VRML++: A Language for Object-Oriented Virtual-Reality Models

Stephan Diehl
FB 14 - Informatik, Universität des Saarlandes,
Postfach 15 11 50, 66041 Saarbrücken, GERMANY
Email: diehl@cs.uni-sb.de, WWW: http://www.cs.uni-sb.de/~diehl

in Proceedings of the 24th International Conference on Technology of Object-Oriented Languages and Systems TOOLS Asia, Bejing, 1997

Abstract

We present a new object-oriented language called VRML++ which extends the Virtual Reality Modeling Language (VRML 2.0), a specification language for interactive three-dimensional scenes on the internet. The new features of VRML++ are classes and inheritance, an improved type system, and dynamic routing. As a net result we get type-safe inclusion polymorphism and dynamic binding. We argue, that these features are essentials of object-oriented programming languages. Furthermore using these new features it is possible to define abstractions of routing structures which we call connection classes. VRML++ increases reuseability, readability, and extensibility of specifications while reducing run-time errors. Finally we discuss our implementation of VRML++.

1 Introduction

The Virtual Reality Modeling Language (VRML) is a data format to describe interactive, three-dimensional objects and scenes which are interconnected via the world wide web. Like the HyperText Markup Language(HTML), VRML is an open standard. The involvement of major software companies in the design and standardization process indicates that VRML will become a key technology for many future internet applications like multi-user environments, computer aided cooperative work or 3D games. In VRML 1.0 it was possible to reuse nodes by instantiating them (`DEF/USE`) and then applying transformations and changes of field properties. In VRML 2.0 prototypes provide a more powerful mechanism to define node types and create instances of these node types. By introducing scripts, events and routes VRML 2.0 added programming language concepts and thus behavior to VRML scenes. We would like to reuse and parameterize such behavior. This is possible to some extent in VRML 2.0, but when we look at programming languages, reuse of code was greatly simplified by the development of object-oriented programming languages (OOPLs). Huge libraries of classes are common to object-oriented languages and make programming an easier task. In software-engineering object-oriented analysis and design is propagated as the way to manage the development of large scale applications. Cox [Cox,1990] predicts a revolution in software industry by reuseable, reliable, abstract software components which can be plugged together to create new applications. At any time a component can be replaced by a more efficient one, which provides the same interface. In a similar way the emerging virtual reality industry could benefit from reuseable, reliable, abstract virtual models and behaviors. This paper describes the design and implementation of VRML++, a language which integrates key concepts of object-oriented programming languages into VRML. In Section 2 we give a short introduction to VRML 2.0 focussing on its abstraction mechanisms. In Section 3 we discuss what we think are essential features of object-oriented programming languages and in Section 5 we identify those present in VRML 2.0. Next our language VRML++ and its implementation are described in Sections 5 and 6. In Section

7 some directions for future work are discussed and Section 8 concludes this paper.

1.1 VRML++ and Authoring Virtual Worlds

The object-oriented features of VRML++ and its improved type system increase reuseability of specifications and prevent run-time errors in animated VRML scenes. VRML++ provides a better way to structure huge libraries of objects and behaviors. Classes can be used to abstract common properties of objects. We get more complex classes by inheriting from other classes and adding new or overriding inherited properties. Instances of classes communicate with the rest of the world via their interface, the implementation details are hidden (encapsulation). In VRML++ it is also possible to abstract routing structures by defining connection classes. These connection classes greatly increase readability.

1.2 VRML++ and Shared Virtual Worlds

We expect, that adding object-orientation to VRML will ease the design of shared virtual worlds in a similar way. Moreover using class libraries can help to solve one of the main problems in shared virtual worlds – the bandwidth problem. Using classes we can reduce the amount of data to be sent between browsers. Similar to the Java class library we would have the same library of objects and behaviors on each client. Thus messages would contain new class definitions or instantiations of classes.

1.3 VRML++ and Animation

The combination of orthogonal animations can be expressed by inheritance. As an example we can combine movement, change of orientation and color by inheriting such behaviors. Furthermore the new dynamic, multiple routing feature of VRML++ provides a powerful means to propagate events to different parts of an animated object.

2 Abstraction in VRML 2.0

For those unfamiliar with VRML, we review in this section those concepts of VRML relevant in the context of this paper. Consequently we do not discuss its graphics primitives, but only look at the way abstractions are defined and "messages" are sent. For more information on VRML 2.0 see [Hartman and Wernecke,1996, VAG,1996].

In VRML a scene graph is defined by nested instantiations of nodes. A node has a type and several fields. Fields have values. Values can be primitive values like floating point numbers or 3D coordinates, but also instances of nodes can be used as values. Many fields also accept lists of values. The instantiation of a node of type t with values $v_1, \ldots, v_n$ for its fields $f_1, \ldots, f_n$ is written t `{` f_1 v_1 ... f_n v_n `}`.

For example, a cylinder of radius 5 and height 10, which is placed 10 meters to the right of the origin of the local coordinate system is instantiated by:

```
Transform { children
             Cylinder { radius 5
                        height 10 }
            translate 10 0 0 }
```

Such an instance can be bound to a name for later reference:

```
DEF CYL Transform { ... }
```

Instead of using standard node types like `Transform` and `Cylinder` one can define new types, which are called prototypes:

```
PROTO BigCylinder
 [ field SFFloat bigheight 10 ]
 { Cylinder { radius 5
              height IS bigheight }
 }
```

The fields and events of the new type are declared in `[ ... ]`. The value of a field or event can be of primitive type (e.g. `SFFLOAT`, `SFBOOL` or `SFTIME`), a single node (denoted by the type `SFNODE`) or a list of nodes (denoted by the type `MFNODE`). In this example a new type `BigCylinder` is defined with a field `bigheight`. In the body of the definition enclosed by `{ ... }` the value of `bigheight` is used as the value for the field `height`. The radius of an object of type `BigCylinder` is always 5, but its height can be provided as a value of the field `bigheight`:

```
BigCylinder { bigheight 20 }
```

In addition to fields a node can have events. The value of an event can change while the scene is rendered, it depends on the passing of time or user interactions. There are input and output events. For example, a node of type `TimeSensor` periodically produces an output event named `fraction_changed`.

```
DEF CLOCK TimeSensor { ... }
```

This event can be sent to other nodes,e.g., a node of type `PositionInterpolator`.

```
DEF PI PositionInterpolator { ... }
```

Depending on the value received as input event `set_fraction` this node computes a position in 3D space as output event `value_changed`. This output event can be sent as a `set_translation` input event to a node of type `Transform`. What output event is sent to which input event is defined with the `ROUTE` primitive. It statically wires one-way communication channels between nodes. Note, that the names `CLOCK`, `PI` and `CYL` have been bound to instances using `DEF` above. At run-time the scene graph can be changed by sending messages along the routes.

```
ROUTE CLOCK.fraction_changed
   TO PI.set_fraction
ROUTE PI.value_changed
   TO CYL.set_translation
```

In our example, the objects defined as children of the `Transform` node change their position in the course of time. What we just described is the basic mechanism to program animations in VRML.

3 Essential Features of Object-Oriented Programming Languages

In VRML 2.0 we can use OOPLs within a `Script` node. The goal of our research is to lift object-orientiation into VRML. To achieve this we first have to identify key ideas of OOPLs. The concepts of OOPLs we will deal with in this paper are:

Objects: Objects encapsulate state and behavior. The state is contained in variables and the behavior in methods. To invoke a method or to read or change the value of a variable a message is sent to an object. Computation in an object-oriented system consists in sending messages between objects.

Classes: Classes are sometimes called object-factories. A class is a scheme, which describes the objects of this class and is used to create such objects. Objects are also called instances of a class. And the process of creating such an object is called instantiation.

Inheritance: A new class is defined by specifying one (single inheritance) or more (multiple inheritance) superclasses. The new class inherits all variables and methods from its superclasses. In addition it can add new variables and methods or even override the definition of an inherited variable or method.

Dynamic Binding: Assume, we have a variable `obj`, and it can be inferred statically,i.e., at compile-time, that the object contained in `obj` is a member of class `K`. Now the method `m` of `obj` is called. Assume, we run the program and `obj` is bound to an object of class `S` and `S` is a subclass of `K`. If static binding is used, then the code of method `m` of `K` is executed. Otherwise, if we use dynamic binding, then the code of method `m` of `S` is executed. Thus dynamic binding makes sure, that at run-time the most specific method is used.

Polymorphism: In a polymorphic language arguments of operators and functions can have more than one type. Cardelli and Wegner [Cardelli and Wegner,1985] distinguish coercion, overloading, inclusion and parametric polymorphism. Due to inheritance, inclusion polymorphism is common to object-oriented languages: If an argument of a function is required to be an object of class `K`, then the function also accepts objects of every subclass of `K` as the value of this argument.

4 How Object-Oriented is VRML ?

In [Matsuda *et al.*,1996] Matsuda, Honda and Lea point out, that objects in VRML have properties, state variables and behaviors. Using standard terminology of the object-oriented programming community, this can only be considered object-based. The extension suggested by Park [Park,1997] is to replace ROUTEs and Scripts by Eventhandlers. He calls the resulting language OO-VRML. But his extension does not increase object-orientation. The work of Curtis [Beeson,1997] shows that there is a need for object hierarchies or even better class hierarchies when it comes to implementing simulations involving behaviors in VRML. He tries to use VRML 2.0 and Java to this end, but because of the lack of inheritance in VRML his implementation becomes complicated.
In our view the concepts present in VRML 2.0 correspond roughly to those of OOPLS as follows:

- Prototypes are classes without inheritance. Instantiation is done by creating a copy of a prototypical instance.
- Nodes of the scene graph are objects.
- Events and `Script` nodes, which process events, are methods.
- Fields are variables.

But VRML lacks inheritance, the essential feature of object-orientation. Furthermore in VRML there is no elaborate type system, no dynamic binding and no inclusion polymorphism.

5 VRML++

VRML++ provides all language constructs of VRML 2.0 but in addition it provides constructs to define classes, express type restrictions and specify dynamic routing. In the following sections we explain these extensions of VRML++.

5.1 Inheritance

If we define a new class `B` to be a subclass of another class `A`, then it has all the events and fields of `A`. But `B` can change some of these events or fields or add new ones. We say, `B` inherits from `A`. The problem is, that if we program this in VRML 2.0 using prototypes, we also have to list all fields and events which remain unchanged (see [Beeson,1997]). This makes the specifications hard to read and maintain. To solve this problem, we extend the syntax of VRML 2.0 and use a preprocessor, which converts the extended syntax into standard VRML 2.0 syntax.

5.2 Classes

The following example shows how we use classes in VRML++. Assume we have a prototype `Robot` which provides the input events `walk` and `jump`. Now we want to define a new prototype `MyRobot` which only differs from `Robot` in that it provides a different implementation of the event `walk`.

```
CLASS Robot [eventIn SFTime walk
             eventIn SFTime jump]
 { ...
   Script
    { field SFNode self USE SELF
      eventIn SFTime walker IS walk
      url "vrmlscript:
             function walker(value)
            ...."
    }

   Script
    { field SFNode self USE SELF
      eventIn SFTime jumper IS jump
      url "vrmlscript:
             function jumper(value)
            ...."
    }
 }

CLASS MyRobot [eventIn SFTime walk]
   EXTENDS Robot
 {
   Script
    { eventIn SFTime runner IS walk
      url "vrmlscript:
             function runner(value)
            ...."
    }
 }
```

Note, that the `jump` event, which was not changed is passed on to `Robot`.
In a class definition `SELF` denotes the instance of the class, when it is instantiated. More precisely: if the class inherits only from `SFNode`, then `SELF` denotes the first node in the class definition. Otherwise, it denotes an instance of the first superclass. A class can inherit from other classes, from other prototypes or from standard VRML nodes like `Sphere` or `Transform`. The top class of the inheritance hierarchy is always `SFNode`.

5.3 Multiple Inheritance

In VRML++ it is also possible for a class to have several superclasses. In this case every field and event is only inherited from the first class in the list of superclasses which supports it. In other words the values of fields and events are only propagated to the first superclass providing it. Assume we defined a class `Zoomer` which has the two `SFTime` events `closer` and `farther`. The first event moves the current viewpoint closer to the object and the second farther from the object. Now we can add this zooming behavior to our robot:

```
CLASS Zoomer
      [ eventIn SFTime closer
        eventIn SFTime farther ]
{ Script
   { field SFNode self USE SELF
     eventIn SFTime closer IS closer
     url "vrmlscript:
           function closer(value)
          ...."
   }

  Script
  { field SFNode self USE SELF
    eventIn SFTime closer IS closer
    url "vrmlscript:
          function closer(value)
         ...."
  }
}

CLASS Zobot [ ]
 EXTENDS Robot, Zoomer {  }
```

5.4 Improved Type System and Inclusion Polymorphism

The lack of a powerful type system in VRML 2.0 can lead to many errors at run-time. Usually these errors are reported when a node is instantiated which tries to add a route to an event not supported by a node. The situation becomes worse when we create new nodes at run-time or allow dynamic routing (see below).
To make sure at compile-time, that a node passed to another node has a certain event or field, we add user defined types to VRML. As a result we make inclusion polymorphism and dynamic binding type-safe.

```
CLASS MoveAble
      [ eventIn SFTime move ]
 EXTENDS SFNode
 { ...
 }

CLASS MyRobot
      [ eventIn SFTime walk
        field MoveAble legs
        ... ]
 EXTENDS Robot
 { ...
   ROUTE walk TO legs.move
 }
```

In this example we require, that `legs` is of type `MoveAble` and not just of type `SFNode` as in VRML. Since by definition all instances of `MoveAble` have an event `move` there can not occur a run-time error like:

> *"Error: Cannot route to a node which has no* `EventIn move` *!"*

5.5 Dynamic Routing and Dynamic Binding

In an animated, interactive virtual world routes can become obsolete and new routes have to be established depending on the user's interaction. What we would like to do, is to get a node at instantiation- (as a value of a field) or run-time (as a value of an event or exposedField) and invoke one of its methods. In VRML 2.0 we can define nodes in a prototype and create new routes between these nodes when instantiating the prototype.

```
PROTO Example []
 {
   DEF EX1 node{}
   DEF EX2 node{}
   ROUTE EX1.out TO EX2.in
 }
```

The following two examples show the dynamic routing features of VRML++. Here the nodes which the prototype creates a route between are not known until an instance of the prototype is created and the nodes are passed as arguments to the prototype. In `Example2` these arguments can be lists (`MFNode`) of nodes and a route is created from each element of (`node1`) to each element of (`node2`).

```
PROTO Example1
      [ field SFNode node1 ...
        field SFNode node2 ... ]
 { ...
   ROUTE node1.out TO node2.in
 }

PROTO Example2
      [ field MFNode node1 ...
        field MFNode node2 ... ]
 { ...
   ROUTE node1.out TO node2.in
 }
```

Such dynamic routing can be implemented by using a `Script` node and the function `addRoute()` of the browser script interface (see 4.7.10 in the specification of VRML 2.0 [VAG,1996]).
With dynamic routing we get dynamic binding, i.e., what method (Script implementing an eventIn) is actually called depends on the type at run-time. For example, consider the classes `Robot` and `MyRobot` defined above. If we define a class `WalkRobot` as follows and instantiate it, it is not known until runtime whether the function `walker` or `runner` is invoked to process the eventIn `walk`:

```
CLASS WalkRobot
    [ field TimeSensor trigger NULL
      field Robot robot NULL ]
 EXTENDS SFNode
 { ROUTE trigger.time TO robot.walk
 }
```

If the value of `robot` is an instance of class `Robot` then `walker` is called, if it is an instance of `MyRobot` then `jumper` is called. The type system of VRML++ makes dynamic binding safer by catching potential run-time error at compile-time.
By using the keyword `UNROUTE` instead of `ROUTE` routes can also be dynamically deleted. As an example an object can be statically routed to a touch sensor. If the user clicks at it, the object is dynamically routed to some other node and by some other event, e.g. when clicking at another sensor node, this route is deleted again.

5.6 Connection Classes

Using dynamic routing we can define connection classes. A connection class abstracts a routing structure, i.e., a connection class is a generic set of routes which can be instantiated.
For example, given a TimeSensor as an argument the class `FanOut` defined below propagates the event `fraction_changed` to a set of other nodes, e.g. Interpolators:

```
CLASS FanOut
    [ field TimeSensor trigger NULL
      field MFNode targets [] ]
 EXTENDS SFNode
 { ROUTE trigger.fraction_changed
      TO targets.set_fraction
 }

DEF O1
    OrientationInterpolator { ... }
DEF O2
    OrientationInterpolator { ... }
DEF TS TimeSensor{ ... }

FanOut { trigger USE TS
         targets [ USE O1 USE O2 ]
       }
```

Another typical connection class is `Filter` which routes different source nodes like sensors to a filter, e.g. an Interpolator, and routes the result of this filter to several target nodes.

```
CLASS Filter
    [ field MFNode sources []
```

```
      field OrientationInterpolator
            filter NULL
      field MFNode targets [] ]
 EXTENDS SFNode
 { ROUTE source.fraction_changed
      TO filter.set_fraction
   ROUTE filter.value_changed
      TO targets.set_rotation
 }
```

The following example shows that one can use a connection class (here: `Mover`) to add an interface to another class. We first define a moving ball similar to the way it would be defined in VRML 2.0:

```
#VRML++ draft utf8

CLASS MoveBall
      [ field TimeSensor from NULL
        field PositionInterpolator
              to NULL ]
  EXTENDS SFNode
  { DEF BALL
        Transform {
         children
          Shape {
           geometry Sphere { } } }
    ROUTE from.fraction_changed
       TO to.set_fraction
    ROUTE to.value_changed
       TO BALL.set_translation
  }

DEF PI PositionInterpolator
  { key [ 0, 0.5, 1 ]
    keyValue [ -1 0 0,
                1 0 0,
               -1 0 0 ]
  }

DEF TS TimeSensor { startTime 1
                    stopTime 0
                    loop TRUE }

MoveBall { timer USE TS
           interpol USE PI }
```

Now we can divide the object moved and the underlying routing structure. Here `SELF` is a special node name denoting the current instance of this class:

```
#VRML++ draft utf8

CLASS Ball [] EXTENDS SFNode
 {  Transform {
     children
      Shape { geometry Sphere {}}}
 }

CLASS MoveBall
      [ field TimeSensor timer NULL
        field PositionInterpolator
              interpol NULL ]
  EXTENDS Ball
  {
    ROUTE timer.fraction_changed
       TO interpol.set_fraction
    ROUTE interpol.value_changed
       TO SELF.set_translation
  }
```

But we can even go a step further and abstract the underlying routing structure, which can be reused to define other moving objects:

```
#VRML++ draft utf8

CLASS Ball [] EXTENDS SFNode
 {  Transform {
     children
      Shape { geometry Sphere {}}}
 }

CLASS Mover
      [ field TimeSensor timer NULL
        field PositionInterpolator
              interpol NULL
      ]
  EXTENDS SFNode
  {
    ROUTE timer.fraction_changed
       TO interpol.set_fraction
    ROUTE interpol.value_changed
       TO SELF.set_translation
  }
```

```
CLASS MoveBall [ ]
 EXTENDS Ball, Mover {  }
```

5.7 Abstract Classes

As a simple example of how inheritance would make important concepts more explicit in VRML, consider that all grouping nodes like `Transform`, `LOD` and `Group` could inherit from an abstract class like `GroupingNode`. By using such an abstract class to provide type information for events and fields, constraints which are currently verbally expressed in the VRML 2.0 specification can be made explicit and enforced by the type system, e.g. that the value of the field `appearance` must be a node of type `Appearance`. Instead of

```
PROTO Shape
 [ exposedField
     SFNode appearance NULL
   ...
 ] { }
```

we write in VRML++:

```
CLASS Shape
 [ exposedField
     Appearance appearance NULL
   ...
 ] { }
```

6 Implementation

We have implemented a preprocessor which translates VRML++ files into VRML 2.0 files. By using such a preprocessor VRML++ becomes very portable and can be used with every VRML 2.0 browser. Currently these browsers have to support JavaScript or VRMLScript.

The preprocessor is able to translate class definitions with multiple superclasses and dynamic routing based on SFNode/MFNode fields and events. It also performs static type checking on the basis of the improved type system.

You can download the source code and executables for Sparc stations (Sun OS) and PCs (DOS) of the current version of our preprocessor from

http://www.cs.uni-sb.de/
  ~diehl/vrml++/content.html

The VRML 2.0 code generated by the preprocessor was tested with the WorldView, CosmoPlayer and Live3D browsers. The preprocessor was written in C++ using GNUs g++, the standard template library, bison and flex. So it should be fairly portable. For dynamic routing we use `Script` nodes with functions in VRMLScript or JavaScript code. As we only use the browser interface, we could generate Java source code as well.

7 Future Work

To prove the claims and design goals of VRML++ in practice we will have to write object and behavior libraries and combine these with authoring tools. Another interesting project would be to integrate VRML++ into an implementation of a shared virtual world to see to what extent it can help to reduce the network traffic. Currently we investigate how VRML++ can be used to program animations such that parts of an animation can be reused. Encouraged by the positive feedback after releasing VRML++ we have initiated a working group officially recognized by the VRML Consortium. The goal of the group is to develop object-oriented extensions for VRML 2.0 or a future standard, see

http://www.cs.uni-sb.de/~diehl/ooevrml/

8 Conclusion

VRML has moved from a pure specification language of a static scene graph to a specification language for objects, behaviors and animations. The need for better ways to structure these specifications has already been noted by other authors. We consider this paper as a proposal in which direction VRML should evolve. We are convinced that VRML++ provides the right extensions for VRML to become object-oriented and thus benefit from object-oriented methodologies developed in the programming language and software-engineering communities.

References

[Beeson,1997] Curtis A. Beeson. An Object Oriented Approach to VRML Development. In *Proceedings of VRML'97*, 1997.

[Cardelli and Wegner,1985] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[Cox,1990] Brad J. Cox. Planning the Software Industrial Revolution: The Impact of Object-Oriented Technologies. *IEEE Software*, 1990.

[Hartman and Wernecke,1996] Jed Hartman and Josie Wernecke. *The VRML 2.0 Handbook - Building Moving Worlds on the Web*. Addison-Wesley, 1996.

[Matsuda *et al.*,1996] K. Matsuda, Y. Honda, and R. Lea. Sony's approach to behavior and scripting aspects of VRML: an Object-Oriented perspective. Technical report, http://www.csl.cony.co.jp/project/vs/proposal/behascri.html, 1996.

[Park,1997] Sungwoo Park. Object-Oriented VRML for Multi-user Environments. In *Proceedings of VRML'97*, 1997.

[VAG,1996] VAG (VRML Architecture Group). *The Virtual Reality Modeling Language Specification – Version 2.0*, 1996. http://vag.vrml.org/VRML2.0/FINAL/.