Object-Oriented Animations with VRML++

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 Proc. of Virtual Environments Conference and 4th Eurographics Workshop, Stuttgart, 1998

Abstract

In this paper we show how VRML++ can be used to specify animations. VRML++ extends the Virtual Reality Modeling Language VRML97 by classes and inheritance, an improved type system, and dynamic routing. Using these new features it is possible to define abstractions of routing structures which we call connection classes. Such connection classes are a powerful tool to define generic animation class. The example discussed in detail in this paper demonstrates that VRML++ increases reuseability, readability, and extensibility of specifications while reducing run-time errors.

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. By introducing scripts, events and routes VRML 2.0 and now VRML97 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 VRML97, but when we look at programming languages, reuse of code was greatly simplified by the development of object-oriented programming languages. In a similar way the emerging virtual reality industry could benefit from reuseable, reliable, abstract virtual models and behaviors. In [Die97] we introduced VRML++, a language which integrates key concepts of object-oriented programming languages into VRML, and discussed its design and implementation. The current paper presents a case study. The specification of an animation written in VRML++ is presented and the underlying VRML++ concepts are discussed in detail. The animation uses generic animation classes which get sensors and interpolators as parameters.

2 Related Work

In [MHL96] Matsuda, Honda and Lea point out, that objects in VRML have properties, state variables and behaviors. Using standard terminology of the object-

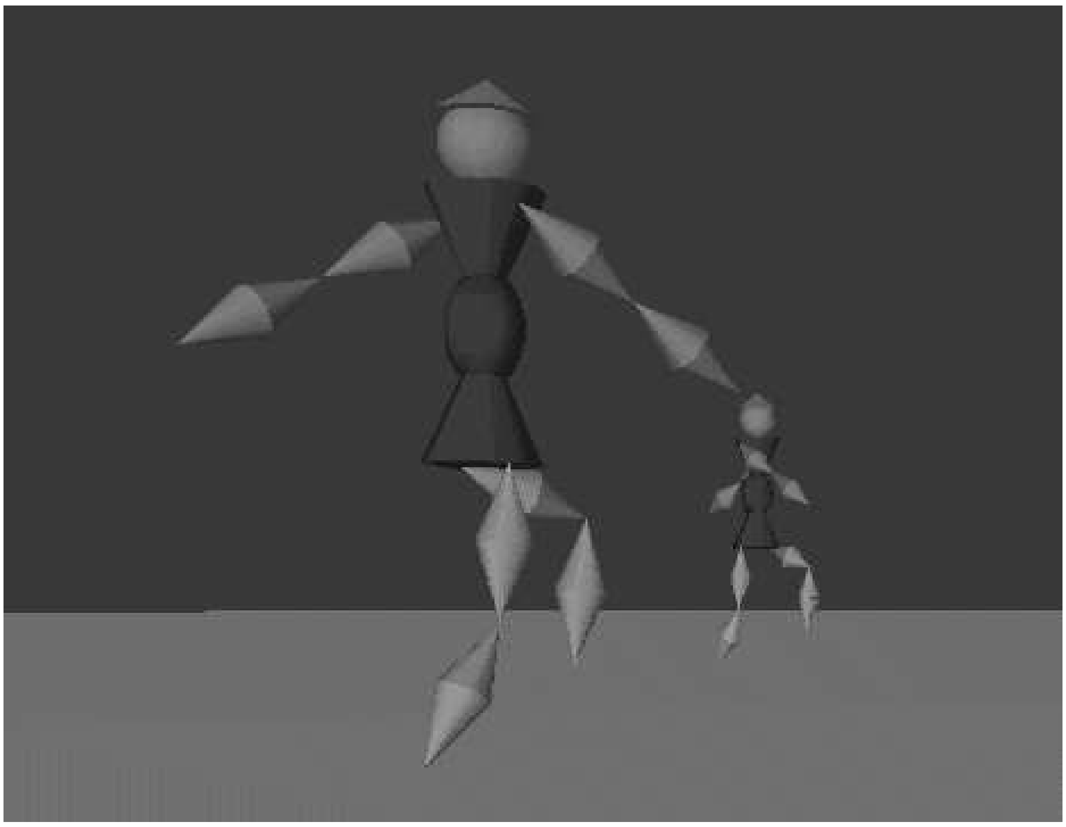

Figure 1: Animation of two persons walking

oriented programming community, this can only be considered object-based. The extension suggested by Park [Par97] 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 [Bee97] 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 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 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.

3 VRML++: The bare bones

In this section we give a very brief introduction to VRML++, for more details see [Die97] or the VRML++ web site.

Classes: Assume you have a prototype which has all fields and events of an existing one, but in addition it should have some new fields and events. In VRML you would have to include the original prototype in the new one and associate explicitly (using IS) each field and event of that instance with the field and events of the new prototype. In the OO literature this implementation technique is called containment. This simple strategy to extend prototypes does usually not suffice to extend geometries, but it is a powerful way to extend behaviors.

```
PROTO Ball [ field SFFloat size 1.0 ]
 { Shape { geometry Sphere { radius IS size }
           appearance ...
         }
 }

PROTO RotatingBall [ field SFFloat size 1.0
                     field SFTime slowdown 1.0 ]
{ DEF SELF Ball { size IS size }
  DEF OI OrientationInterpolator { ... }
  DEF CLOCK TimeSensor { cycleInterval IS slowdown }
  ROUTE C.fraction_changed TO OI.set_fraction
  ROUTE OI.value_changed TO SELF.set_fraction
}
```

In VRML++ this can be written in a more readable and maintainable way, because fields are inherited automatically:

```
PROTO RotatingBall [ exposedField SFTime slowdown 1.0 ] EXTENDS Ball
{ DEF OI OrientationInterpolator { ... }
  DEF CLOCK TimeSensor { cycleInterval IS slowdown }
  ROUTE C.fraction_changed TO OI.set_fraction
  ROUTE OI.value_changed TO SELF.set_fraction
}
```

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`.

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.

Types: The VRML type system only discriminates primitive types like integers, floats and boolean values and all composed data are considered to be of type `Node`. Furthermore the value of a field can be a single value (indicated by the prefix SF of the type) or a list of values (indicated by the prefix MF of the type). In addition, in VRML++ the names of standard nodes, as well as those of user-defined prototypes and classes can be used as type restrictions for fields and events. A major reason for errors in VRML scenes are routes to nodes which do not have the field referred to in the ROUTE. This problem becomes even more important, when nodes and routes are created at runtime in Scripts (via the Browser Interface) or Applets (via the External Authoring Interface) or in VRML++ via dynamic routing. By using our improved type system constraints which are currently expressed verbally in the VRML97 specification can be made explicit and enforced by type checking, For example, in the specification we can read, that the value of the field `appearance` must be a node of type `Appearance`. This can be directly expressed in VRML++ as:

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

Dynamic Routing: In VRML routes can only be declared between DEF'ed nodes or added and deleted at runtime by programs. In contrast, VRML++ offers dynamic routing. By dynamic routing we mean that routes are created between nodes which are passed as field values to a prototype, e.g.

```
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
 }
```

We call such a class, which abstracts a routing structure, a connection class. A connection class contains a generic set set of routes which can be instantiated. The above example demonstrates a typical connection class, a `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.

4 Case Study: An Animation in VRML++

VRML++ provides all language constructs of VRML97 but in addition it provides constructs to define classes, express type restrictions and specify dynamic routing.

In this section we discuss the specification of an animation of a walking person (see Figure 1) using these new language concepts.

```
#VRML++ draft utf8

Background { groundColor [ 0.3 0.5 0.3 , 0.1 0.4 0.1 ]
             groundAngle 1.57
             skyColor 0 0 0.6
           }

PROTO ArmPart [exposedField SFColor color 1 0 0]
{ ... }
```

In VRML++ we can instantiate and define prototypes using VRML97 syntax.

```
PROTO Arm
      [ exposedField SFRotation elbowrotation 0 0 0 0
        exposedField SFRotation shoulderrotation 0 0 0 0
        exposedField SFColor color 1 0 0
      ]
{ Transform {
   children [ ArmPart { color IS color }
              Transform { children ArmPart { color IS color }
                          translation 0 0.40 0
                          rotation IS elbowrotation }
            ]
   rotation IS shoulderrotation }
}
```

To model an `Arm` of a person we use two `ArmPart` objects. The arm can be animated by rotating its joints (shoulder and elbow). We also define prototypes for `BodyPart` and `Head`.

```
CLASS Leg [ ] EXTENDS Arm { }
```

In VRML++ a class can inherit from other classes and prototypes. The new class has all the events and fields of its superclasses, but the new class can also change some of these events or fields or add new ones. The topmost class of the inheritance hierarchy of VRML++ is `SFNode`. Next we define a subclass `Man` of `SFNode` which gets as parameters objects representing the different body parts of a person.

```
CLASS Man [ exposedField MFNode head [ ]
            exposedField MFNode body [ ]
            exposedField MFNode rightarm [ ]
            exposedField MFNode leftarm [ ]
            exposedField MFNode rightleg [ ]
            exposedField MFNode leftleg [ ]
          ] EXTENDS SFNode
{
```

```
Transform { children [
  Transform { children IS body },
  Transform { children IS head
              translation 0 0.35 0
           },
 Transform { children IS rightarm
             translation 0.15 0.30 0
             rotation 0 0 1 3.14
           },
 ...
 ### similar for leftarm, rightleg and leftleg

 ]
 }
}
```

Note that an instance of this class is still a static object. To animate such an object, we define a subclass `Walkman` of `Man`. Some of the fields of `Walkman` are interpolators which animate the joints of the object. In VRML++ names of classes and prototype can be used as types of fields and events. These types can be used to prevent run-time errors as we know them from VRML97. Usually these errors are reported when a node is instantiated which tries to add a route to an event not supported by a node.
In our example we define a class `JointInterpolator`, which is only used as a type restriction in class `Walkman`. In Java such classes, which can not be instantiated but are used as intermediate classes in the inheritance hierarchy and for similar typing purposes are called abstract classes.

```
CLASS JointInterpolator
      [ eventIn SFFloat set_fraction
         eventOut SFRotation value_changed ]
EXTENDS SFNode { } # abstract class

CLASS ArmShoulderInt [ ]
EXTENDS OrientationInterpolator, JointInterpolator { }

#similar definitions for ArmElbowInt, LegShoulderInt
#and LegElbowInt
```

A concrete subclass of `JointInterpolator` could implement its events with a script. But we simply use an `OrientationInterpolator` to define the concrete subclass `ArmShoulderInt`. The above definition is tricky. It uses multiple inheritance, i.e., several superclasses for the new class. As a result instances of the new class satisfy the type restriction `JointInterpolator` but get their implementations from the first superclass `OrientationInterpolator`.
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 route from or to one of its events. This is not possible in VRML97. In VRML97 one can only route from and to nodes which have been bound to a name with DEF. The following example shows 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. 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.

```
CLASS Walkman
      [ field TimeSensor clock NULL
        field JointInterpolator armrightelbow NULL
        field JointInterpolator armrightshoulder NULL
        ...
        ### similar for armleftelbow, armleftshoulder, legleftelbow,
        ### legleftshoulder, legrightelbow and legrightshoulder
      ] EXTENDS Man
{
 # MOVE RIGHT ARM
 ROUTE clock.fraction_changed TO armrightelbow.set_fraction
 ROUTE clock.fraction_changed TO armrightshoulder.set_fraction

 ...
 ### similar for armleftelbow, armleftshoulder, legleftelbow,
 ### legleftshoulder, legrightelbow and legrightshoulder
}
```

Note in the above example that clock in the routes is not a DEF'ed name, but a field name. In VRML++ it is also possible that the value of such a field is a list of nodes. In this case routes from or to every node in the list are created.
So far an instance of Walkman would move its limbs, but the object would stay at the same position. Next we define a very useful generic animation class which extends the standard prototype Transform. The new class Move gets a clock and an interpolator as field values, routes the clock to the interpolator and the value of the interpolator to the translation field, which it inherited from Transform. Thus Move abstracts the classical animation mechanism used in VRML.

```
CLASS Move [ field TimeSensor clock NULL
             field PositionInterpolator moveInt NULL
           ] EXTENDS Transform
{ ROUTE clock.fraction_changed TO moveInt.set_fraction
  ROUTE moveInt.value_changed TO SELF.set_translation
}
```

Now we have defined all body parts and animation classes for our example and it remains to show how to use them. First we instantiate interpolators for the joints. Note that the two interpolators below use the same key values but their keys are reversed. This way we achieve that one arm goes up, when the other arm goes down.

```

DEF ArmRightShoulderInt
    ArmShoulderInt {
     key [ 0 , 0.2, 0.8, 1 ]
     keyValue [ 0 0 0 0, 1 0 0 1.4, 1 0 0 -1.4, 0 0 0 0]
    }
DEF ArmLeftShoulderInt
    ArmShoulderInt {
     key [ 1, 0.8, 0.2, 0 ]
     keyValue [ 0 0 0 0, 1 0 0 1.4, 1 0 0 -1.4, 0 0 0 0]
    }
...

# dito for elbow joint of arm and joints of leg
```

We also want to animate the position of the person. We divide this in two animations. One animation changes the horizontal position of the person, the other animation the vertical position. The second animation is necessary, because if the person bends its knees its distance to the ground has to be decreased.

```
DEF PosInt
    PositionInterpolator {   # run 10 meters
     key [ 0,  1 ]
     keyValue [ 0 0 0, 0 0 20 ]
    }

DEF HeightInt
    PositionInterpolator {   # adjust vertical position
     key [ 0 , 0.5, 1 ]
     keyValue [ 0 0 0, 0 -0.2 0, 0 0 0 ]
    }
```

For our animations we use two clocks. A fast one which controls the joints and a slow one which controls the position of the person.

```
DEF CLOCK  TimeSensor { loop TRUE }

DEF SLOWCLOCK TimeSensor { cycleInterval 20 loop TRUE }
```

Finally the walking person can be instantiated.

```
DEF WALKER
Move {
 clock USE SLOWCLOCK
 moveInt USE PosInt
 children
  Move {
   clock USE SLOWCLOCK
   moveInt USE HeightInt
   children
     Walkman
      { body BodyPart { color .14 .14 .15 }
        head Head { color 0.4 0.4 0.5
                    hatColor 0.6 0.4 0.5 }
        rightarm Arm { color 0.4 0.4 0.45 }
        leftarm Arm { color 0.4 0.4 0.45 }
        rightleg Leg { color 0.5 0.5 0.55 }
        leftleg Leg { color 0.5 0.5 0.55 }
        clock USE CLOCK
        armrightelbow USE ArmRightElbowInt
        armrightshoulder USE ArmRightShoulderInt
        ...
        ### similar for armleftelbow, armleftshoulder, legleftelbow,
        ### legleftshoulder, legrightelbow and legrightshoulder
      }
    }
}
```

5 Implementation

We have implemented a preprocessor which translates VRML++ files into VRML97 files. By using such a preprocessor VRML++ becomes very portable and can be used with every VRML97 browser. Currently these browsers have to support JavaScript or VRMLScript.
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 VRML97 code generated by the preprocessor was tested with the WorldView, CosmoPlayer and Live3D browsers. But all these browsers are moving targets, so don't hesitate to contact the author if you have problems with viewing VRML++ worlds with your browser.

6 Future Work

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 VRML97 or a future standard, see

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

The ideas brought up in this group still have to be integrated into one consistent proposal. Then we intend to implement this proposal reusing parts of the implementation of VRML++.

7 Conclusion

In this paper we gave a case study of how one can write animations in VRML++. The specifications are more readable then similar VRML97 specifications. Because of inheritance the specifications are also more reuseable and extensible. Finally the type system of VRML++ describes the interface of an object to the rest of the world more precisely and helps to prevent run-time errors. 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

[Bee97] Curtis A. Beeson. An Object Oriented Approach to VRML Development. In *Proceedings of VRML'97*, 1997.

[Die97] Stephan Diehl. VRML++: A Language for Object-Oriented Virtual Reality Models. In *Proceedings of the 24th International Conference on Technology of Object-Oriented Languages and Systems TOOLS Asia'97*, Beijing, China, 1997.

[MHL96] 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.

[Par97] Sungwoo Park. Object-Oriented VRML for Multi-user Environments. In *Proceedings of VRML'97*, 1997.