Extending VRML by One-Way Equational Constraints

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

Abstract

In this paper we discuss how VRML can be extended by one-way equational constraints. We give the syntax of our extension, discuss what constraints can be used for in VRML and by means of example show how constraints can be translated into VRML using scripts and routes.

1 Introduction

VRML has become the standard for providing 3D content on the internet. VRML has evolved from a description language of a static scene graph to a modeling language for 3D worlds including behavior, animations, and user interaction. A revised specification of VRML 2.0 [VAG96, HW96] is currently undergoing the ISO voting procedure and will certainly become the offical ISO Standard VRML'97. In the rest of this paper we use the term VRML as a synonym for both VRML 2.0 and VRML'97. From a programming language designer's point of view VRML lacks many features which have proven useful for specifying algorithms. As VRML was primarily designed with the intention to specify 3D objects and their behavior, we have to be careful when we try to transfer programming language concepts to VRML. Previously we have designed a language called VRML++ [Die97], which extends VRML by classes, inheritance, an improved type system and dynamic routing. The current paper addresses the design and implementation issues of extending VRML by constraints. Constraints make VRML more expressive. They ease specification animations and of layout and interaction in user-interfaces. In distributed systems they provide a concise way to encode communication and synchronization. As we want to extend VRML, we first have to look at what is there in VRML (see the brief introduction to scripts and routes in Section 2). Our final goal is to implement constraints on top of VRML, i.e., it should be possible to translate constraints into pure VRML. This translation is done by a preprocessor, which we are currently implementing. After preprocessing every standard VRML browser can be used to render the 3D scene.

2 Dynamic Features of VRML: Routes and Scripts

In VRML a scene graph consists of nodes. Each node can have several edges. There are three kinds of edges: `field`, `eventIn` and `eventOut`. Fields are used to store data, while an eventIn receives data and an eventOut sends data along a route. A route connects an eventOut to an eventIn. If an eventOut is connected to an eventIn, then the value of the eventIn changes, whenever the value of the eventOut changes. An efficient implementation of routes is described in [WNB97].
In VRML there are some special nodes which produce eventOuts, these include `TouchSensor` nodes which produce an event, when the user clicks at an object and `TimeSensor` nodes, which periodically produce eventOuts.
For example, a node of type `TimeSensor` periodically produces an eventOut named `fraction_changed`.

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

This eventOut can be sent to other nodes, e.g., an interpolator node of type `PositionInterpolator`.

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

Depending on the value received as eventIn `set_fraction` this node computes a position in 3D space as eventOut `value_changed`. This eventOut can be sent as a `set_translation` eventIn to a node of type `Transform`. What eventOut is sent to which eventIn is defined with the `ROUTE` primitive. It statically wires routes 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 values 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.
In VRML 2.0 it is also possible to add programs in languages like JavaScript, VRMLScript or Java to the 3D objects in a file. Such program code is enclosed in nodes of type `Script`. Execution of this code is triggered by events. For each eventIn of a `Script` node a function with the same name as the eventIn is defined. The function gets the value of the eventIn as an argument. Within the function new values can be assigned to eventOuts of the `Script` node or to eventIns of other nodes. These other nodes will then react accordingly to these changes. Note, that in this way it is possible to change an eventIn of a node without using a route.

3 One-Way Equational Constraints

A constraint denotes a relationship among two or more objects. Constraints are declarative, i.e., they state the relationship but not the way to maintain it. Thus constraint-based systems consist of a set of constraints and one or more constraint-solvers. The task of a constraint-solver is to determine how and in what order to satisfy constraints. One-way equational constraints are a weak form of constraints, which have been widely used to implement user-interfaces:

```
lights.on=or(switch1.on,switch2.on)
```

One-way constraints have been used for many purposes including layout, animations and user-interfaces. They can express relations like attachment or noncollision of objects or enforce physical laws. Their success is mainly due to three factors: they are efficient, intelligible and domain-independant.
Still there are some technical problems even with this sort of constraints. Consider the following example of a one-way constraint:

```
radio.size=max(selection.size,radio.size)
```

Here the constraint is cyclic, i.e., the value of `radio.size` depends on its own value.
If the set of objects changes dynamically, objects must be able to reference other objects indirectly. This can be achieved by pointer variables[ZMGS94]. Each object maintains a pointer to its parent and a set of pointers to its children, e.g.:

```
button.color=invert(PARENT.color)
```

Such a pointer variable may reference an object which does not yet exist or does no longer exist. As long as all objects participating in a constraint exist, it must be satisfied. Thus when these objects are created or deleted, the constraint may have to be activated or deactivated.

4 What to use One-Way Constraints for in VRML?

Constraints have been used for user-interfaces in the Sketchpad system[Sut63]. Constraints enforce hidden relations between objects as they are common in layouts or animations. For example several kinds of 2D animations can be expressed as constraints in Amulet [MMMF96], and there exists a constraint-based sytem to visually construct 3D animations[GB95]. In fact many animation techniques in the literature, e.g., inverse kinematics, morphing, flocking, particle systems, are specified by sets of constraints, then one or more variables are manipulated over time. The animation is produced as the constraint solver tries to satisfy the constraints by computing the properties of the graphic objects.
In VRML the value of a field can also be a list of primitive values or nodes. For example the field `children` contains a list of nodes. In the following example the function `invert` seems to have only one argument, but the constraint actually

connects the field `color` of the node containing this constraint to the `color` fields of all nodes in the list referenced by `SWITCHES.children`. So `SELF.color` changes, whenever one of these fields changes.

```
SELF.color=invert(SWITCHES.children.color)
```

In distributed virtual worlds one-way constraints could also be used for communication and synchronization, e.g.:

```
car.speed = max(user1.car.speed, user2.car.speed)
```

In the following example we express a multi-way constraint by a set of one-way constraints. Assume `yes` and `no` are cylinders representing the number of consenting and rejecting votes and `sum` a cylinder representing the total number of votes.

```
sum.height = yes.height + no.height
yes.height = sum.height - no.height
no.height = sum.height - yes.height
```

Note, that these constraints are cyclic. As we use local propagation which is unable to solve cyclic constraints, we have to open the cycle to guarantee termination of evaluation. As we implement local propagation on top of the routing mechanism of VRML, we have to explain what the behavior of the system is. The VRML browser executes the following loop [WNB97]:

1. Processing of events which are caused externally like mouse clicks and clock ticks.
2. Routing of the messages
3. Evaluation of scripts and interpolators
4. Updating the scene graph based on the events that occured since the last rendering
5. Rendering of the scene

By using timestamps or similar techniques the browser postpones the handling of events which are changed twice during one iteration to the next iteration. As a result we can use non-termination as a feature: cyclic constraints can be used to drive animations in VRML.

```
car.translation = plus(car.translation,5,0,0)
```

5 Syntax of Constraints

We extend VRML such that constraints can be used as values of fields. We say, that we attach the constraint to the field. We change the grammar of VRML [VAG96] such that there is an additional case for field values, namely that they have to satisfy a constraint:

$$\begin{array}{lcl} \mathit{fieldvalue} & ::= & \ldots \\ & \mid & \underline{=}\ \mathit{path} \end{array}$$

Such constraints consist of references to fields and events of other nodes and functions which combine the values of these fields and events:

$$\begin{array}{lcl} \mathit{path} & ::= & \mathit{startpath}\ \mathit{restpath} \\ \mathit{startpath} & ::= & \mathit{nodename} \\ & \mid & \underline{\mathbf{SELF}} \\ & \mid & \underline{\mathbf{PARENT}} \\ & \mid & \mathit{functionname}\ \underline{(}\ \mathit{path}^*\ \underline{)}\ (\underline{[}\ \mathit{int}\ \underline{]}) \\ & \mid & \mathit{fieldname}\ (\underline{[}\ \mathit{int}\ \underline{]}) \\ \mathit{restpath} & ::= & \underline{.}\ \mathit{fieldname}\ (\underline{[}\ \mathit{int}\ \underline{]})\ (\mathit{restpath}) \\ & \mid & \underline{.}\ \underline{\mathbf{PARENT}}\ (\mathit{restpath}) \end{array}$$

So the following would be a valid node instantiation:

```
TimeSensor { enabled = if(PARENT.children[2].switch.on,
                          PARENT.clock.switch,
                          GLOBAL.clock.switch).active }
```

The semantics of `SELF` and `PARENT` is most interesting, if they occur in prototype definitions:

```
PROTO Car [ SFFloat carheight 0
            MFNode wheels [ ]
          ] { ... }

PROTO Wheel [ field SFColor color ]
  { Shape {
     geometry Cylinder { height=div(PARENT.PARENT.carheight,20)
                         radius=mult(SELF.height,4) }
     appearance Appearance { material Material = color } } }

Car { carheight 2
      wheels Wheel { color 1 0 0 }
    }
```

In the body of the prototype `Wheel` the path `PARENT.PARENT.carheight` refers to a field which depends on the context where this prototype is instantiated.

The path SELF.height refers to a field of the Cylinder node, whereas the path color refers to the field color of the prototype. As this is an important point and can lead to misconceptions for readers with OO programming background (c.f. this in Java), we emphasize, that in our constraints the name SELF does not refer to the instance of the prototype, but to the inner node it occurs in. As a consequence SELF can be used outside of prototypes.

We also add a new language construct to define those functions used in constraints. In function definitions first the result type, the name of the function and the names and types of its arguments are given. The body of the function definition consists of the definition of local variables and program code written in VrmlScript. The program code must use return to return the result of the function. Local variables are static, which means that they can be used to store information between subsequent calls of the function.

$$
\begin{array}{lll}
\mathit{path} & ::= & \mathit{startpath}\ \mathit{restpath} \\
\mathit{functiondef} & ::= & \underline{\mathbf{FUN}}\ \mathit{typename}\ \mathit{functionname} \\
 & & \underline{[}\ (\mathit{typename}\ \mathit{parametername})^*\ \underline{]} \\
 & & \underline{\{}\ (\mathit{typename}\ \mathit{variablename}\ \mathit{defaultvalue}\ \underline{;})^* \\
 & & \quad \mathit{functionbody}\ \underline{\}} \\
\mathit{functionbody} & ::= & \text{body of a VRMLScript function}
\end{array}
$$

For the first example constraint above we define the function if as follows:

```
FUN SFNode if [ SFBool cond
                SFNode iftrue
                SFNode iffalse ]
 { if (cond) return iftrue; else return iffalse; }
```

6 An Example

Now we look at a more complex example, which encodes a simple animation using our constraints in VRML. In the next section we will then discuss how this example can be converted into VRML.

```
PROTO Dog [ ] { ... }
PROTO Cat [ ] { ... }

DEF DOG1 Transform { children Dog { } }
DEF DOG2 Transform { children Dog { }
                     translation 2 0 0 }

DEF CATS Transform { children [
 Transform { translation = distance(DOG1.translation_changed,
                                    DOG2.translation_changed,
                                    2,0,0)
```

```
             children Cat { } }
 Transform { translation = distance(DOG1.translation_changed,
                                    DOG2.translation_changed,
                                    -4,0,0)
             children Cat { } } ] }

FUN SFVec3f distance [ SFVec3f pos1
                       SFVec3f pos2
                       SFFloat x S
                       FFloat y
                       SFFloat z ]
 { SFVec3f newpos 0 0 0;
   newpos[0]=(pos1[0]+pos2[0])/2+x;
   newpos[1]=(pos1[1]+pos2[1])/2+y;
   newpos[2]=(pos1[2]+pos2[2])/2+z;
   return newpos;
 }
```

The above specification describes a scene with two dogs and two cats. Whenever one or both of the two dogs move the two cats will follow them, such that they keep the same distance to both of the dogs.

7 Translation to VRML 2.0

To illustrate how constraints can be translated into VRML we discuss now the translation of the above example to VRML. The example covers only a small part of all cases, which we have to deal with in our implementation, e.g., there are no paths involving set valued fields and there are no indirectly referenced objects.
First we give names to the nodes in which the constraints occur.

```
DEF CATS Transform { children [
 DEF X1 Transform { children Cat { } }
 DEF X2 Transform { children Cat { } } ] }
```

Then we define a script for every constraint. The script contains the definitions of all functions used in the constraint. In addition it defines for each variable in the constraint an eventIn and a field. Furthermore it defines for every variable a function which processes the associated eventIn and stores the value in the associated field.

```
DEF CONSTR1 Script { eventIn SFVec3f set_arg1
                     eventIn SFVec3f set_arg2
                     field SFVec3f arg1 0 0 0
                     field SFVec3f arg2 0 0 0
```

```
                    eventOut SFVec3f result
                    field SFVec3f newpos 0 0 0
                    url "vrmlscript:
                      function distance(pos1,pos2,x,y,z)
                        { newpos[0]=(pos1[0]+pos2[0])/2+x;
                          newpos[1]=(pos1[1]+pos2[1])/2+y;
                          newpos[2]=(pos1[2]+pos2[2])/2+z;
                          return newpos; }
                      function set_arg1(value)
                        { arg1=value;
                          result=distance(arg1,arg2,2,0,0); }
                      function set_arg2(value)
                        { arg2=value;
                          result=distance(arg1,arg2,2,0,0); }
                      " }

DEF CONSTR2 Script { eventIn SFVec3f set_arg1
                    eventIn SFVec3f set_arg2
                    field SFVec3f arg1 0 0 0
                    field SFVec3f arg2 0 0 0
                    eventOut SFVec3f result
                    field SFVec3f newpos 0 0 0
                    url "vrmlscript:
                      function distance(pos1,pos2,x,y,z)
                        { newpos[0]=(pos1[0]+pos2[0])/2+x;
                          newpos[1]=(pos1[1]+pos2[1])/2+y;
                          newpos[2]=(pos1[2]+pos2[2])/2+z;
                          return newpos; }
                      function set_arg1(value)
                        { arg1=value;
                          result=distance(arg1,arg2,-4,0,0); }
                      function set_arg2(value)
                        { arg2=value;
                          result=distance(arg1,arg2,-4,0,0); }
                      " }
```

Finally the variables in a constraint have to be routed to the eventIns of the script generated for the constraint. And the result of the script has to be routed to the field to which the constraint was attached.

```
ROUTE DOG1.translation_changed TO CONSTR1.set_arg1
ROUTE DOG2.translation_changed TO CONSTR1.set_arg2
ROUTE CONSTR1.result TO X1.set_translation

ROUTE DOG1.translation_changed TO CONSTR2.set_arg1
ROUTE DOG2.translation_changed TO CONSTR2.set_arg2
ROUTE CONSTR2.result TO X2.set_translation
```

8 Implementation

We are currently implementing a preprocessor which translates VRML specifications with constraints into VRML. The preprocessor is entirely written in Java using parser and lexer generation tools for the front end.
Experiments with examples translated by hand suggest that the performance of these constraints is acceptable for user-interfaces but not for animations. Using Java instead of VrmlScript in function definitions and generated `Script` nodes might improve the performance considerably.
The major technical problem, that we still have to solve, has to do with indirectly referenced objects in VRML, because these need not always exist. There are several situations when this can be the case, e.g., the object has not yet been loaded over the internet, it has not yet been put at the referenced position in the graph (`addChildren` event) or it has been removed (`removeChildren` event). Routes from or to events of nodes which do not exist crash the browser. So we have to think about ways to remove and add routes, whenever one of its participating nodes is removed or added.
After finishing this implementation we plan to look into multi-way constraints and constraint hierarchies and see whether they can be implemented in a similar way.

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 and make them more expressive has already been noted by other authors. We have shown that one-way equational constraints can be implemented on top of VRML. If they prove their usefulness in practice, they could be integrated into VRML and thus efficiently implemented constraint-solvers would be provided by VRML browsers. In this sense, we consider this paper as a proposal in which direction VRML should evolve.

Acknowledgements

I want to thank the members of our working group on object-oriented extensions of VRML (http://www.cs.uni-sb.de/~diehl/ooevrml/). The work presented in this paper would not have been possible without the inspiring discussions with Sungwoo Park and Jean-Francis Balaguer.

References

[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 (to appear)*, Beijing, China, 1997.

[GB95] Enrico Gobbetti and Jean-Francis Balaguer. An Integrated Environment to Visually Construct 3D Animations. In *Proceedings of SIGGRAPH'95*, 1995.

[HW96] Jed Hartman and Josie Wernecke. *The VRML 2.0 Handbook - Building Moving Worlds on the Web*. Addison-Wesley, 1996.

[MMMF96] Brad A. Myers, Robert C. Miller, Rich McDaniel, and Alan Ferrency. Easily Adding Animations to Interfaces Using Constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology UIST'96*, Seattle,WA, 1996.

[Sut63] I. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the IFIP Spring Joint Computer Conference*, 1963.

[VAG96] VAG (VRML Architecture Group). *The Virtual Reality Modeling Language Specification – Version 2.0*, 1996. http://vag.vrml.org/VRML2.0/FINAL/.

[WNB97] Danial J. Woods, Alan Norton, and Gavin Bell. Wired for Speed: Efficient Routes in VRML 2.0. In *Proceedings of VRML'97*, 1997.

[ZMGS94] Brad Vander Zanden, Brad A. Myers, Dario Guise, and Pedro Szekely. Integrating Pointer Variables into One-Way Constraint Models. *ACM Transactions on Computer Human Interaction*, 1:161–213, 1994.