in Proceedings of 5th International Conference on Computer Graphics and Artificial Intelligence 3IA'2002, Limoges (France), May 14-15, 2002.

Constraints for 3D Graphics on the Internet

Stephan Diehl and Jörg Keller

FB 14 - Informatik, Universität des Saarlandes,

Postfach 15 11 50, 66041 Saarbrücken, GERMANY

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

Primary contact author: Stephan Diehl, diehl@acm.org

Abstract

In this paper we present a ¤exible extension to support constraints in VRML97, the standard for interactive 3D graphics on the internet. After a short introduction to the dynamic concepts of VRML97, we discuss the design rationale of our extension and explain its implementation. In particular we look at how the extension is integrated into the initialization process and the rendering loop of a VRML browser, the software which loads and displays VRML scenes. So far, we implemented solvers for one-way equational and £nite domain constraints, but our architecture allows to easily add other constraint solvers.

Keywords: computer graphics, world-wide-web, one-way equational constraints, £nite-domain constraints

1 Introduction

VRML has become the standard for providing 3D content on the internet. VRML was designed to be declarative, but the way program code is added does destroy this declarativity. Constraints can remedy this situation and increase the potential for visual authoring and visual programming. Our extension allows to separate constraints and solvers, thus allowing constraints to become part of the VRML £le. The architecture described in this paper makes it possible to add other kinds of constraints, e.g. non-linear constraints, by taking care of the integration into the interactive, real-time graphics framework of VRML. Thus, to constraint programmers it opens up a wide area of emerging applications on the internet including 3D modeling, simulation and gaming as well as 3D e-commerce.

This paper focusses on the implementation aspects of the extension and in particular on the challenging task of properly integrating constraint solving into the event processing and rendering loop. As we use established constraint solving methods these are only described to the extend to which it is required to illustrate the integration. Ideally, the discussion should motivate and enable others to apply their constraint solvers to virtual worlds on the internet.

1.1 Constraints in Computer Graphics

Constraints have been used for user-interfaces since the Sketchpad system [13] in 1963. 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 [10], and there exists a constraint-based system to visually construct 3D animations [9]. In fact many animation techniques in the literature, e.g., inverse kinematics, morphing, ¤ocking, particle systems, are speci£ed 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. Constraints are also a powerful tool for modeling [8, 15] and visual programming [2]. In distributed systems they provide a concise way to encode communication and synchronization [1].

1.2 Extending the VRML97 standard

VRML has evolved from a description language of a static scene graph (VRML 1.0 in 1995) to a modeling language for 3D worlds including behavior, animations, and user interaction (VRML 2.0 in 1996,[3]). A revised speci£cation of VRML 2.0 has become an of£cial ISO Standard called VRML97 [14]. 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++ [5, 6], which extends VRML by classes, inheritance and an improved type system and introduces the concept of dynamic routing. The current paper addresses the design and implementation issues of extending VRML by constraints. Constraints make VRML more expressive. They ease speci£cation of animations and of layout and interaction in user-interfaces.

Applications implemented with our exten-

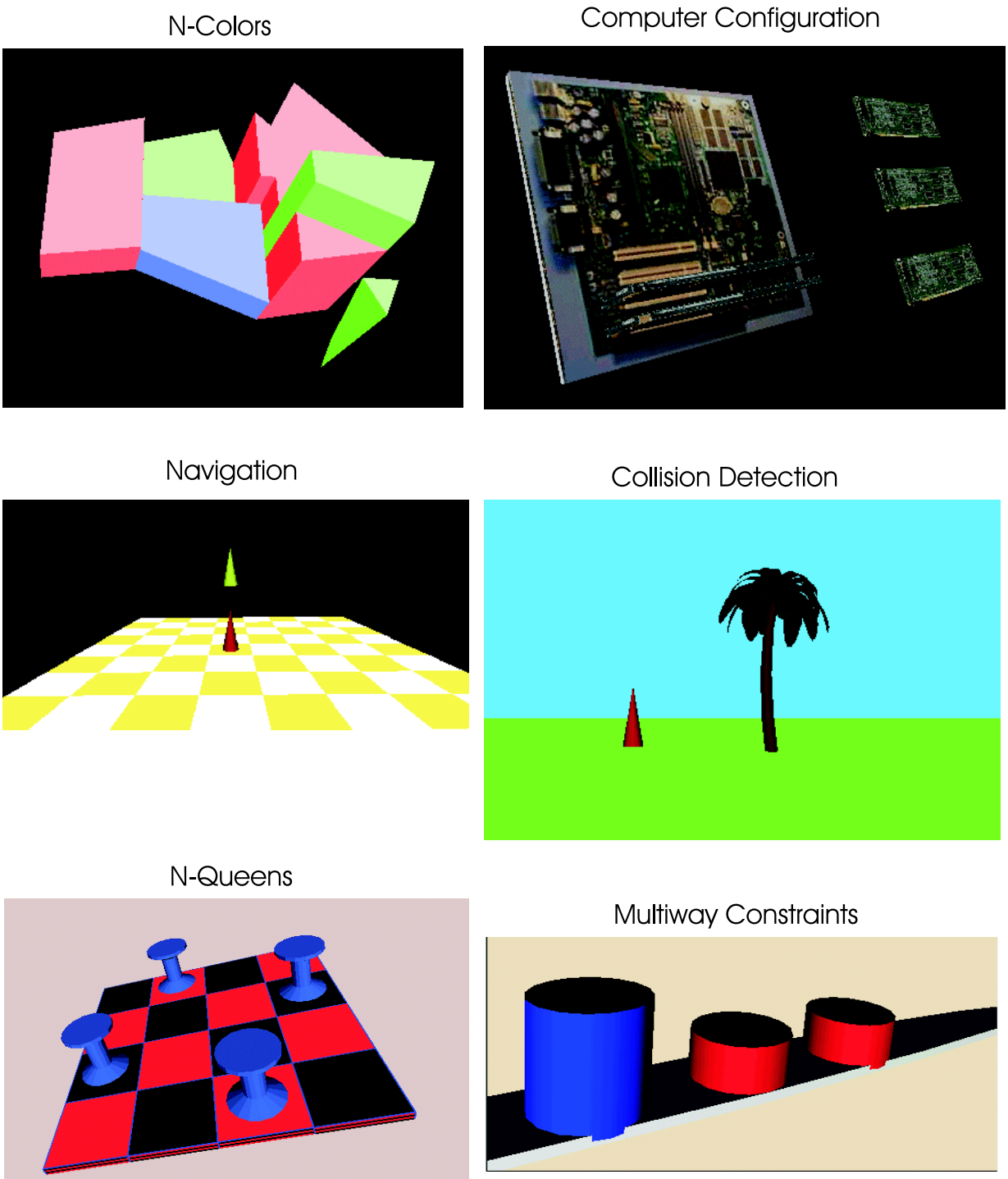

Figure 1: Some applications implemented with VRML and constraints

Figure 2: Flocking

sion include navigation, collision-detection, con£guration and animation (see Figure 1) and have been described in a previous paper [7], which targets a computer graphics audience. As an example of these applications Figure 2 shows an event-based implementation of ¤ocking [11]. Below the de£nition of the constraints using the extension we introduce in this paper is given. Every bird in a ¤ock of n birds gets a constraint encoding its dependency on itself and the remaining $n-1$ birds:

```
DEF constr Constraint {
 startEval TRUE
 inames [ "b1", "b2", "b3" ]
 inodes [ USE bird1,
          USE bird2 ,
          USE bird3 ]
 constraints [
   "b1.translation=
     Follow(b1.translation,
            b2.translation,
            b3.translation)"
   "b2.translation=
     Follow(b2.translation,
            b1.translation,
            b3.translation)"
   "b3.translation=
     Follow(b3.translation,
            b1.translation,
            b2.translation)"
   ] }
```

2 Dynamic Features of VRML: Routes and Scripts

As we want to extend VRML, we £rst have to look at what is there in VRML.

The part of the software which creates an image from a description of a scene, its objects and light sources and the viewer is called the renderer. In most 3D graphics systems the description is represented as a graph, called the scene graph. The renderer traverses this graph and accumulates the information contained in the nodes along each path. This information includes transformations, colors and geometric data. As the scene graph is usually a directed acyclic graph and not just a tree, a node can be reached on different paths, i.e. it has to be evaluated in different contexts.

For computer animations the render loops:

1. Change the scene graph or values in the scene graph.

2. Render the scene.

In classical 3D systems the scene graph was built as part of the program code, that is, it was integrated into the system. Using VRML a scene graph can be speci£ed independently of a speci£c implementation of a graphics system. The constructs of VRML are called nodes, because they specify a scene graph. A node contains £elds, which correspond to the edges of the graph. A node is de£ned by its node type and values for its £elds. These values can be nodes again. There are default values of each £eld and each £eld may only take values of a certain type. The syntax to de£ne a node is as follows:

$$\texttt{Type}\,\{field_1\ value_1\ \ldots\ field_n\ value_n\}$$

For example `Box { size 1 2 1 }` de£nes a node of type `Box` and its £eld `size`

has the value `1 2 1`, i.e. a box with 1m, height 2 m and depth 1m is de£ned.

Actually, there are three kinds of £elds: `field`, `eventIn` and `eventOut`. Pure £elds 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 ef£cient implementation of routes is described in [16].

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 de£ned 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 de£ned 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 VRML97 it is also possible to add programs in languages like JavaScript, VRMLScript or Java to the 3D objects in a £le. Such program code is attached to nodes of type `Script`. Execution of this code is triggered by events. Using JavaScript for each eventIn of a `Script` node a function with the same name as the eventIn has to be de£ned. The function takes the value of the eventIn as 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. The implementation of script nodes using Java differs from the above and is described in the section “Java Scripting API” of the VRML97 speci£cation [14]. In particular, the API provides the methods `getValue()` and `setValue()` to read and write events.

As an alternative to `Script` nodes VRML97 also provides the so-called External Authoring Interface EAI. The EAI allows applets to call methods of the VRML browser and vice versa.

3 Design Rationale

Our £nal goal is that constraints become an integral part of the VRML language. The £rst step to achieve this is to provide an extension

of VRML97 such that we and others can experiment with different ¤avors of constraints.

A £rst effort to implement constraints on top of VRML with the help of a preprocessor turned out to be too restrictive and was heavily depending on the routing mechanism of a particular browser [4]. At a workshop at VRML98 Richard presented an implementation where constraints had been hard coded in a constraint solver in Java to control a VRML scene [12].

In our approach constraints and solver are separated and constraints are part of the VRML £le. To achieve this we implemented a prototype `Constraint`. It encapsulates constraint solvers in a `Script` node. The solvers are programmed in Java. The constraints are passed as strings to the `Script` node.

```
EXTERNPROTO Constraint [
  field MFString inames
  field MFString innodes
  field MFString protoField
  field MFString protoType
  field MFString domains
  field MFString domainDefs
  field MFString userFunctions
  field SFBool startEval
  field SFBool eventFirstPriority
  field MFString constraints
] "ProtoConstraint.wrl#Constraint"
```

Some of the £elds of the `Constraint` prototype are only needed as work arounds for missing functionality of the Java Scripting API, so we do not explain them here, but they are described in [7].

The £elds `domains` and `domainDefs` are used to de£ne the domains for £nite domain constraints.

The £eld `userFunctions` offers the possibility to add any needed function to the constraint solver and use these functions in the constraints. The syntax needed to support this feature is very easy: First, the user has to de£ne the signature of the function and then the function can be de£ned in JavaScript syntax. The signature consists of the return type, the function name and the parameter types.

If the £eld `eventFirstPriority` is `TRUE`, a variable which was changed by an event and triggers the constraint solver can not be set to a different value by the constraint solver.

Finally, the £eld `constraints` contains a list of strings (`MFString`), each string represents a constraint. The constraints are interpreted as one-way equational constraints if `domains` and `domainDefs` are empty and as £nite domain constraints, otherwise.

Constraints are of the form *path relop expression* where *path* identi£es a £eld in the scene graph, *relop* is a relational operator and *expression* is either a constant of primitive type like `MFInt32`, a value of a £eld or a function, see grammar below:

constraint	→	*path relop expression*
relop	→	= | != | <= | >= | < | >
expression	→	*path*
		| *functionname* (*path**)
		| *constant*
path	→	*nodename.tail*
tail	→	*£eldname* | *£eldname* [*int*]
		| *tail.tail*

The following is an example of a constraint, which relates the value of a £eld `radius` of type `SFFloat` to the second value in a £eld `size` of type `SFVec3f`:

```
CAR.children[2].radius
  = Add( SLIDER.size[2], 10)
```

Assume that you want to position an object `A` between two objects `B` and `C` and that you have written a function `middle(p,q)` which computes the mid point between two points `p`

and q. In VRML for the function we have to create a Script node S and then we have to add the following routes:

```
ROUTE B.translation_changed
   TO S.set_p
ROUTE C.translation_changed
   TO S.set_q
ROUTE S.result
   TO A.set_translation
```

Using constraints we can just write:

```
A.translation
  = middle(B.translation,
           C.translation)
```

This is not just shorter, but it is also more readable, accessible and maintainable. More examples including the VRML source code can be found in a previous paper [7].

4 Implementation

Figure 3 shows the scene graph, routes and dependency graph of the £rst example constraint above. The dependency graph is built by a Java program which is encapsulated in the Constraint node in the scene graph. To better understand the relation of the scene graph and the dependency graph as well as how the constraint solvers work, we look at the steps performed by the initialization process and the rendering loop of a VRML browser and what additional steps are added by our extension.

4.1 Initialization

During initialization the VRML browser performs the following three steps:

1. Load VRML £le, parse it and build scene graph including script nodes.
2. Connect all routes.
3. Call method initialize() of each script node.

In the last step also the method initialize() of the script in the Constraint prototype is called. It parses the string contained in its £eld constraints and creates a dependency graph. A node in the dependency graph represents a variable in the constraints and contains an attribute outofdate, which indicates, that its value of this variable is valid or not. It contains a list dependencies of nodes representing variables which depend on this variable and it contains a list constraints of objects representing single constraints. These objects contain the variable, a relational operator and a tree representing the expression in the constraint. Finally, each node in the dependency graph has two ¤ags which are needed because not all £elds referenced in the paths of a constraint must exist at all times. The ¤ag nodeActive is false if the VRML-node to which the £eld belongs does not exist at initialization or whenever it is removed. The ¤ag constraintActive is false, whenever the ¤ag nodeActive of this node or a node it depends on is false. As a result, when a constraint is not active the constraint solver will not try to solve it. But as soon as all variables referenced in the constraint exist, it will become active and thus be solved.

For each variable in the constraints it creates a script node which receives the value of the variable and sends the name of the variable to the script node of the Constraint prototype. For this, additional routes are added to the scene graph. We need an intermediate node for each variable, because in case of fan-in, i.e. if several variables send their modi£ed values to the script, the script node can not identify the sender of each value, it just receives the value.

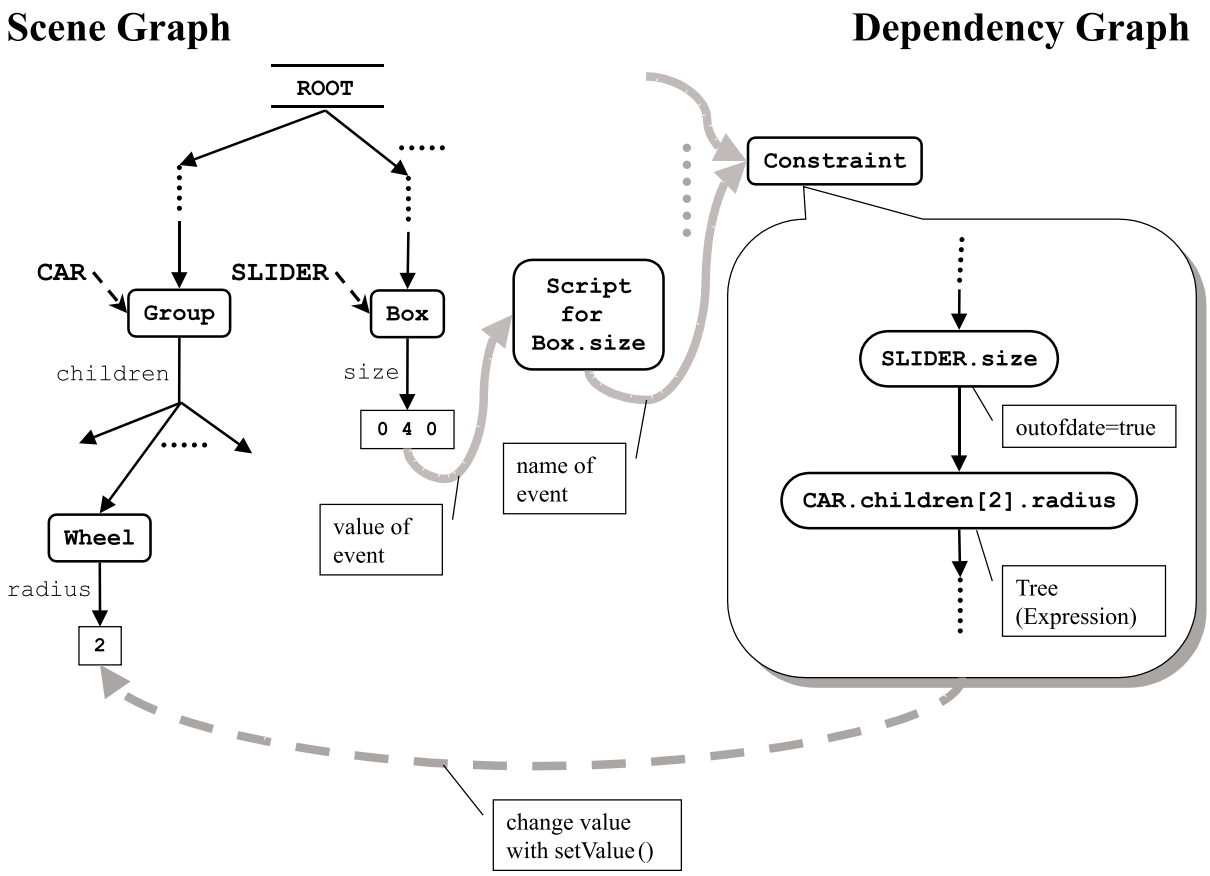

Figure 3: Relation of scene graph and dependency graph

But if the script receives the name of the variable, it can access the value with the method `getValue()` of the Java Scripting API.

For each user-defined function the method `initialize()` of the `Constraint` prototype adds a script node, which encapsulates that function, to the scene graph as we will explain in the next section.

If the field `startEval` of the `Constraint` prototype is `TRUE`, also the constraint solver is invoked during this initialization process.

4.2 Rendering Loop

After initialization the VRML browser executes the following loop [16]:

1. Process events which are caused externally like mouse clicks and clock ticks.
2. Route the messages
3. Evaluate scripts and interpolators and propagate events they cause. By using timestamps or similar techniques the browser postpones the handling of events which are changed more than once during one iteration to the next iteration.
4. Update the scene graph based on the events that occured since the last rendering
5. Render the scene

If the value of a £eld which is a variable in the constraints changes, an event occurs and in step 3 of the above loop the constraint solver is invoked. It gets the names of all variables which changed since the last frame rendered and sets the attribute `outofdate` of these variables in the dependency graph to `false`.

4.2.1 One-way equational constraints

To solve one-way equational constraints we use an extension of the algorithm by Zanden et. al. [17] for constraints with pointer variables. In particular, we added activation and deactivation of constraints to allow dynamic adding and removing of nodes. As in their algorithm the constraint solver has two phases. In the nulli£cation phase all variables are marked as outofdate which depend on a variable which has become outofdate. In the re-evaluation phase the values of all variables are recomputed recursively. Let k be the dependency graph node whose value changed:

```
if (eventFirstPriority==false)
  eval(k)
else
  for each d ∈ k.dependencies
    do eval(d)
```

In the second case only the values of all depending variables are recomputed. Here is the de£nition of the function `eval()`.

```
eval(k) = {
 if (k.isoutofdate
     and k.activeNode
     and k.constraintActive)
   { k.outofdate=false;
     for each c ∈ k.constraints
         do c.solveConstraint();
     for each d ∈ d.dependencies
       do eval(d); } }
```

where `solveConstraint()` computes the value of the expression in the constraint.

4.2.2 Adding and removing nodes

In VRML it is possible that nodes can be added or removed dynamically from the scene graph by using the events `addChildren` and `deleteChildren`. As a result, in a path like `CAR.children[2].radius` the node referenced by `CAR.children[2]` might cease to exist or might be added later. To cope with this situation routes are added from each `children` £eld in a path to the eventIn `childrenControl` of the script in the `Constraint` prototype. The value of the event is the partial path for this £eld, e.g. `CAR.children`. The script can now activate or deactivate the according node in the dependency graph and all nodes depending on it.

4.2.3 User-de£ned functions

To implement user-de£ned functions we had two alternatives: create a script node for each use of such a function in a constraint or create a script node for each function de£ned. The structure and protocol of the £rst solution would have been less complex, but it would have increased the number of script nodes in the scene graph considerably. Thus we chose the second solution. Consider the following example where we implement a function `EqualY(p,q)` which returns `true` if the Y-coordinate of the two points `p` and `q` are equal:

```
Constraint { ...
 userFunctions
  [ "SFBool
      EqualY(SFVec3f, SFVec3f)"
    "function eqy(x1, x2)
      { return x1[1]==x2[1]; }"
  ]
 ... }
```

At initialization the `Constraint` prototype will create the following script node from the above speci£cation:

```
Script {
 eventIn SFVec3f set_p1
 eventIn SFVec3f set_p2
 eventIn SFBool UpdateResult
 field SFVec3f p1 0 0 0
 field SFVec3f p2 0 0 0
 eventOut SFBool ResultUpdated
 eventOut SFBool result
 url [
  "vrmlscript:
    function set_p1(value)
     { p1=value; }
    function set_p2(value)
     { p2=value;
       result=eqy(p1, p2);
       ResultUpdated=true; }
    function UpdateResult(value)
     { ResultUpdated=false; }
    function eqy(x1, x2)
     { return x1[1]==x2[1]; }"
   ]
 }
```

To evaluate an occurrence of the function `EqualY(p,q)` in a constraint, the constraint solver (in the method `solveConstraint()`), will perform the following steps:

1. Set eventIn `UpdateResult` to false.
2. Set eventIn `set_p1` to the value of p.
3. Set eventIn `set_p2` to the value of q. Then the function `set_p2()` performs the test and sets the ¤ag `ResultUpdated` which is used for synchronization of the script node for the function `EqualY()` and the constraint solver, because these are running concurrently.
4. Wait until the ¤ag `ResultUpdated` becomes true, then read the value of the eventOut `result`.

4.2.4 Finite domain constraints

For £nite domain constraints we have to make sure that all constraints are satis£ed before the last step in the rendering loop, such that no inconsistent scene is rendered. For this, we have a copy of the value of each £eld in the dependency graph and after all constraints have been solved these values are written to the scene graph at once. Our FDC solver uses backtracking with domains sorted by size (£rst fail principle).

4.2.5 Technical Remarks

We would have liked to add a keyword `parent` for paths in the constraints. Unfortunately the Java Scripting API does not provide a way to access the parent or one of the parents of a node. The only workaround we came up with was to mirror the whole scene graph and process all events additionally in Java, which would considerably degrade the performance.

Speaking of performance, we found that the Java Scripting API is the major bottleneck in our implementation. The problem is that the value of a £eld is set by method calls instead of assignments. We found that setting the value of an object of class `SFVec3f` with `setValue()` it is about 300 times slower than assigning the value to a ¤oat array of size three which would be an equivalent, but more ef£cient representation. If the `SFVec3f` object is actually a £eld in the scene graph, the access is another 20% slower. Fortunately, we did not use the External Authoring Interface EAI and implement the constraint solver as an applet. For the EAI we measured that calls to `setValue()` are even 20.000 times slower

than assignments to ¤oat arrays.

5 Conclusion

Applications of constraints for 3D computer graphics on the internet include navigation, collision-detection, con£guration and animation. Constraints capture the required knowledge in a declarative way and thus increase maintainability of such applications. We feel that constraints provide a powerful, expressive and natural way to specify dependencies between £elds of different nodes and that they should replace or extend routes in a future VRML standard. We have shown how constraints can be integrated into the initialization process and the rendering loop of a VRML browser. The whole extension was designed such that it is easy to add other constraint solvers as they only work on the dependency graph. There should be no need to change the parser, the communication with the VRML scene and the handling of user-de£ned functions. As part of our future research we want to investigate how to combine our previous work on VRML++ with that on constraints.

References

[1] T. Axling, L. Fahlen, and S. Haridi. Virtual Reality Programming in Oz. In *Proceedings of the 3rd Eurographics Workshop on Virtual Environments*, 1996.

[2] Margaret Burnett and Allen Ambler. Interactive Visual Data Abstraction in a Declarative Visual Programming Language. *Journal of Visual Languages and Computing*, 5(1), 1994.

[3] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Addison Wesley Longman, 1997.

[4] Stephan Diehl. Extending VRML by One-Way Equational Constraints. In *Workshop on Constraint Reasoning on the Internet*, Schloss Hagenberg, Austria, 1997.

[5] 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. IEEE Computer Society Press.

[6] Stephan Diehl. Object-Oriented Animations with VRML++. In *in Proceedings of Virtual Environment 98 Conference and 4th Eurographics Workshop*, Stuttgart, Germany, 1998.

[7] Stephan Diehl and Jörg Keller. VRML with Constraints. In *Proceedings of Fifth Symposium on the Virtual Reality Modeling Language Web3D/VRML 2000, Monterey, CA*. ACM SIGGRAPH, 2000. `http://www.cs.uni-sb.de/~diehl/pubs/web3d2000.pdf`.

[8] Michael Gleicher. Practical Issues in Graphical Constraints. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*. MIT Press, 1995.

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

[10] 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.

[11] Craig W. Reynolds. Flocks, Herds and Schools: A distributed Behaviour Model. *Computer Graphics (Proc. of SIGGRAPH'87)*, 21(4), 1987.

[12] Nadine Richard and Philippe Codognet. Multi-way Constraints for Describing High-Level Object Behaviours in VRML. In *Proceedings of the Workshop on VRML and Object Orientation*, Monterey, 1997.

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

[14] The VRML Consortium. VRML97 International Standard Speci£cation (ISO/IEC 14772-1:1997). `http://www.web3d.org/Specifications`, 1997.

[15] Suresh Thennarangam and Gurminder Singh. Inferring 3-dimensional constraints with DEVI. In Alan Borning, editor, *Proceedings of Second International Workshop on Principles and Practice of Constraint Programming PPCP'94*, volume 874 of *Lecture Notes in Computer Science*. Proceedings, Springer-Verlag, Berlin/Heidelberg, 1994.

[16] Daniel J. Woods, Alan Norton, and Gavin Bell. Wired for Speed: Ef£cient Routes in VRML 2.0. In *Proceedings of the Second Symposium on the Virtual Reality Modeling Language VRML'97*. ACM SIGGRAPH, 1997.

[17] 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.