VRML with Constraints

Stephan Diehl Jörg Keller

University of Saarland[*]

to appear in Proceedings of ACM Web3D/VRML 2000 Symposium, Monterey, CA

Abstract

In this paper we discuss the benefits of extending VRML by constraints and present a new way based on prototypes and scripting to implement this extension. Our approach is easy-to-use, extensible and it considerably increases the expressivity of VRML. Our implementation supports one-way equational and finite domain constraints. We demonstrate the use of these constraints by means of several examples. Finally we argue that in the long run constraints should become an integral part of VRML.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality; D.3.3 [Programming Languages]: Language Constructs and Features—Constraints

Keywords: VRML, Animation, Programming

1 Introduction

Many problems can be viewed as problems of constraint satisfaction in which the goal is to discover some problem state that satisfies a given set of constraints. Constraints are equality and inequality relations which contain variables of the problem state. This approach has been heavily investigated in artificial intelligence, but it is also applicable to many problems in computer graphics, human computer interaction and virtual reality. A constraint denotes a relationship among two or more objects. Constraints are declarative, i.e., they state a 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.

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++ [2, 3] 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 of animations and of layout and interaction in user-interfaces.

Our implementation works well with CosmoPlayer 2.1. As of this writing, Superscape's Viscape, ParallelGraphics' Cortona, Microsoft's Worldview and Blaxxun's Contact Pro have not yet implemented the JS API completely.

2 Constraints in Computer Graphics

Since 35 years constraints have been used for user-interfaces, e.g. in the Sketchpad system [9]. 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 [6], and there exists a constraint-based sytem to visually construct 3D animations [4]. In fact many animation techniques in the literature, e.g., inverse kinematics, morphing, flocking, particle systems, are specified by sets of constraints, where one or more variables are manipulated over time. The animation is produced as the constraint solver tries to satisfy the constraints and thus changes the properties of the graphic objects involved.

3 VRML with Constraints

Our final goal is that constraints become an integral part of the VRML language. The first step to achieve this is to provide an extension of VRML97 such that we and others can experiment with different flavors of constraints. For this purpose we implemented a prototype `Constraint`. It encapsulates constraint solvers in a `Script` node. The solvers are programmed in Java.

```
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 SF-
Bool   eventFirstPriority
            field MFString constraints
]
"ProtoConstraint.wrl#Constraint"
```

Some of the fields of the `Constraint` prototype are only needed as work arounds for missing functionality of the Java Scripting API. We need the `inames` and `inodes` to bind nodes to names because there is no function `getNode(DefName)` in the JS API as we know it from the EAI.

Furthermore the fields `protoField` and `protoType` map fields of a node, which is an instance of a user defined prototype, to

[*]FB Informatik, University of Saarland, Postfach 15 11 50, 66041 Saarbrücken, GERMANY, diehl@cs.uni-sb.de

their types. This is necessary because we can not get the type of a field via the JS API.

The fields `domains` and `domainDefs` are only used for finite domain constraints. In the `domains` -field the user assigns one of the domains defined in the `domainDefs` -field to the variables used in the constraints.

The field `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 define the signature of the function and then the function can be defined in JavaScript syntax. The signature consists of the return type, the function name and the parameter types.

The value of `startEval` determines whether the constraints will be solved at initalization of the `Constraint` or not. Furthermore, the value of `eventFirstPriority` controls whether values set by an event can be changed by the constraint solver or not. `eventFirstPriority` is `TRUE` by default. In this case, the events from VRML scene graph have highest priority and will not be changed by the solver. As an example consider, that you want to move an object in a room using a `PlaneSensor` . If `eventFirstPriority` is `TRUE` , no collision-handling possible. By collision-handling we mean detection of a collision and automatically moving the object to a non-colliding position. Without collision-handling the object can move through walls or things standing in the room. Therefore, it is necessary to set `eventFirstPriority FALSE` . Now, if the objects approaches the wall, the constraint solver is able to prevent the collison.

Finally, the field `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 finite domain constraints, otherwise.

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

constraint	$\rightarrow$	*path relop expression*
relop	$\rightarrow$	= | != | <= | >= | < | >
expression	$\rightarrow$	*path* | *functionname* (*path**) | *constant*
path	$\rightarrow$	*nodename.tail*
tail	$\rightarrow$	*fieldname* | *fieldname* [*int*] | *tail.tail*

For example, `CAR.children[2].radius=Add(BOX.size[2], 10)` is a constraint, which might relate the value of a field `radius` of type `SFFloat` to the second value in a field `size` of type `SFVec3f`.

3.1 One-Way Equational 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)
```

In the above example the value of `lights.on` is updated whenever the value of `switch1.on` or `switch2.on` changes. In general, the value of the field on the left side of the equality constraint is changed whenever one of the values on the right side changes.

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.

Different algorithms for solving such constraints, even for dynamically changing sets of objects and constraints are described in [10]. In this case constraints have to be activated or deactivated when objects are created or deleted.

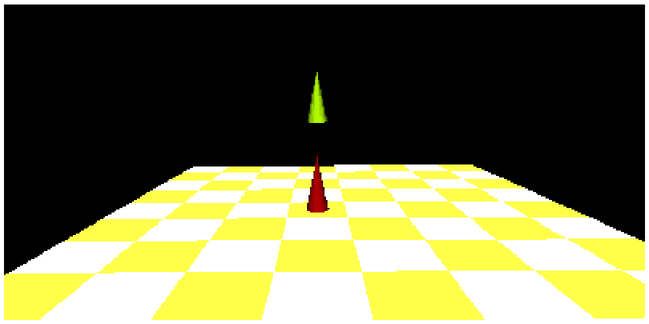

Figure 1: Navigation on a chessboard

Now, we want to show how to use one-way constraints in VRML by means of examples.

Navigation with one-way constraints

The first example in Figure 1 uses constraints to navigate a set of pieces on a chessboard.

With the help of constraints we want to enforce 3 restrictions: The pieces cannot be moved from the board, the pieces are always centered in a square on the chessboard and when moving a piece it cannot pass through another piece. The first 2 restrictions are realized with the user-function `CenterField` . The third restriction is achieved by lifting a piece before it is moved.

```
Constraint {
   startEval TRUE
   eventFirstPriority FALSE
   userFunctions [
      "SFFloat CenterField(SFFloat)"
        "function CenterField(val) {
                    var  help=Math.round(val);
                    if (help>7)  help=7;
                    if (help<0)  help=0;
                    return help; }" ]
  inames [ "PS1" "piece1"
           "PS2" "piece2"   ]
  inodes [ USE PS1,  USE piece1,
           USE PS2,  USE piece2  ]
  constraints [
    "piece1.translation[0]
      =CenterField (PS1.translation_changed[0])"
    "piece1.translation[1]=If(PS1.isActive, 2, 0.5)"
    "piece1.translation[2]
      =CenterField (*(-1, Ps1.translation_changed[1]))"
    "piece2.translation[0]
      =CenterField (PS2.translation_changed[0])"
    "piece2.translation[1]=If(PS2.isActive,2 , 0.5)"
    "piece2.translation[2]
      =CenterField (*(-1, PS2.translation_changed[1]))"
   ]
}
```

Note, that the Y-coordinate of the `PlaneSensor`, that controls the piece is assigned to its Z-coordinate because the chessboard lies in the X-Z-plane.

Collision Detection

The next example in Figure 2 shows how user-defined functions can be used to implement collision detection. In our 3D computer configuration the user can choose some of the cards offered and put

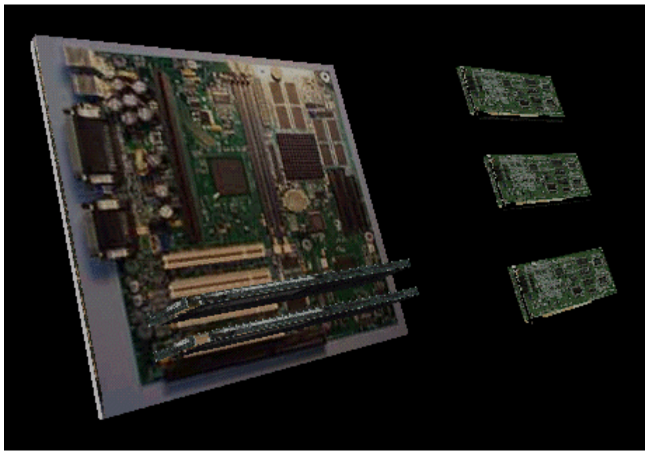

Figure 2: Interactive computer configuration

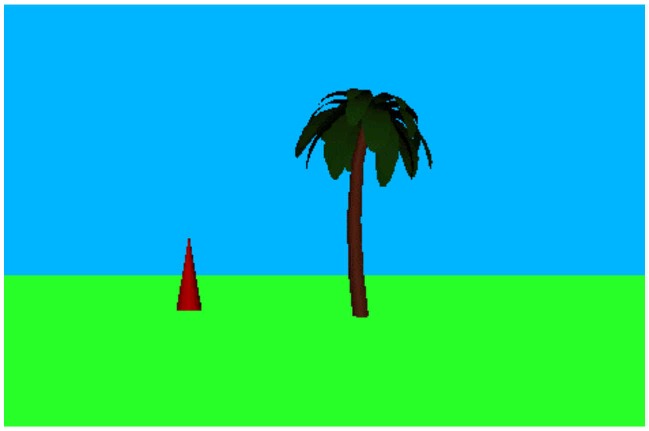

Figure 3: Collision detection and handling

them into a free slot. If he chooses a card and moves it to the board, the constraints automatically detect the collison with the board and put the card into a free slot. Therefore, two things must be changed by the constraint solver: The card must be rotated into a vertical rotation to the board and the translation must be set to the slot's position. In the example, we have 5 cards but only 4 slots. The user can either put Card1 or Card1a into slot1. Therefore, in the case of collision with the board it must be checked if the slot is occupied or free. For each of the cards Card2, Card 3 and Card4 a separate slot is reserved, so only collision with board must be considerd in these constraints.

```
Constraint {
  eventFirstPriority FALSE
  inames [ "Card1" "Card1a" "Card2" "Card3" "Card4"]
  inodes [ USE Card1, USE Card1a, USE Card2,
           USE Card3, USE Card4 ]
  userFunctions [
    "SFBool Collision(SFVec3f, SFVec3f, SFFloat)"
    "function Collision (x1, x2, v) {
                 a=x1[0]-x2[0];
                 b=x1[1]-x2[1];
                 dst=a*a+b*b;
                 if (Math.sqrt(dst)< v ) return true;
                 return false; }"
    "SFBool Equal(SFVec3f, SFVec3f)"
    "function Equal (x1, x2) {
                if (x1[0]==x2[0] && x1[1]==x2[1]
                    && x1[2]==x2[2]) return true;
                return false; }"
   ]
  constraints [
   "Card1.translation
     =If(Collision(-3 -0.6 0, Card1.translation, 3),
         If (Equal(Card1a.translation, -2.7 -0.6 0.4),
             3 3 0,
              -2.7 -0.6 0.4),
         Card1.translation)"

   "Card1.rotation
     =If(Collision(-3 -0.6 0, Card1.translation, 3),
         1 0 0 1.5707,
         0 0 0 0)"

   "Card1a.translation
     =If(Collision(-3 -0.6 0, Card1a.translation, 3),
         If (Equal (Card1.translation, -2.7 -0.6 0.4),
             6 3 0,
             -2.7 -0.6 0.4),
         Card1a.translation)"

    ...

   "Card4.rotation
     =If(Collision(-3 -1.55 0, Card4.translation, 3),
         1 0 0 1.5707,
         0 0 0 0)"
  ]
}
```

Figure 4: Flocking

The example in Figure 3 shows how a piece in a scene avoids collision with an obstacle represented by a tree. The constraint solver will change the position of the piece relative to its position. If the piece is moved in X-direction and it approaches the tree, the Z-position of the piece will be changed. After passing the tree, the old Z-position is restored and the piece moves along its original path.

Flocking

The motion of a flock of birds is one of nature's delights [7]. We implemented an event-based kind of flocking, see Figure 4. Consider a flock of n birds. Every bird gets a constraint encoding its dependency on itself and the remaining n-1 birds:

$$\begin{aligned}&\texttt{bird}_x.\texttt{translation} =\\ &\quad \texttt{Follow}(\ \texttt{bird}_x.\texttt{translation},\\ &\qquad \texttt{bird}_1.\texttt{translation}, \ldots, \texttt{bird}_{x-1}.\texttt{tranlsation},\\ &\qquad \texttt{bird}_{x+1}.\texttt{translation}, \ldots, \texttt{bird}_n.\texttt{tranlsation})\end{aligned}$$

If an event occurs, e.g. the user clicks at a bird and drags it, the function `Follow` computes the distances from the bird itself (first

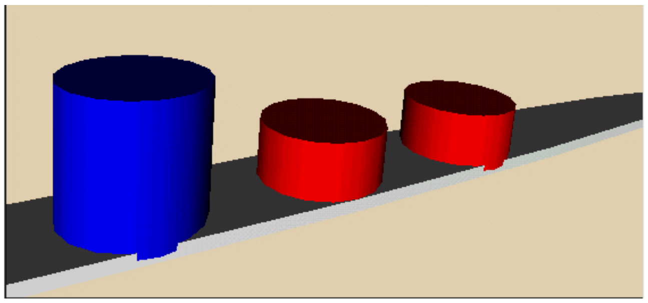

Figure 5: Multi-way constraints

argument) to all other birds. Then, the bird follows that bird with the smallest distance.

Simulation of Multi-Way Constraints

In the following example, shown in Figure 5, 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
```

With the help of our prototype this can be written as:

```
Constraint {
   inames [ "SUM", "YES", "NO" ]
   inodes [ USE SUM, USE YES, USE NO ]
   constraints [
     "NO.translation[1]=Sub(SUM.translation[1],
                            YES.translation[1] )"
     "YES.translation[1]=Sub(SUM.translation[1],
                             NO.translation[1] )"
     "SUM.translation[1]=Add(YES.translation[1],
                             NO.translation[1] )"
   ]
}
```

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. The result of the above constraints is, that whenever the size of one cylinder is changed, e.g. by user interaction, the size of the other two cylinders is adapted accordingly.

3.2 Finite Domain Constraints

In many problems the possible values of variable can take are restricted to a finite set. Such problems include the configuration of systems, schecduling or timetabling. They all have in common that one has to choose amongst a finite number of possibilities. There are a variety of constraint solving algorithms for this case. Currently we have implemented a simple backtracking solver, but we plan to replace it by a more efficient one as soon as possible.

As an example of a finite domain problem consider the problem of coloring a map of australia in Figure 6.

```
Constraint {
  startEval TRUE
  inames [ "WS" "NT" "Q" "SA" "NSW" "V" "T" ]
  inodes [ USE WS_C USE NT_C USE Q_C USE SA_C
           USE NSW_C USE V_C USE T_C ]
```

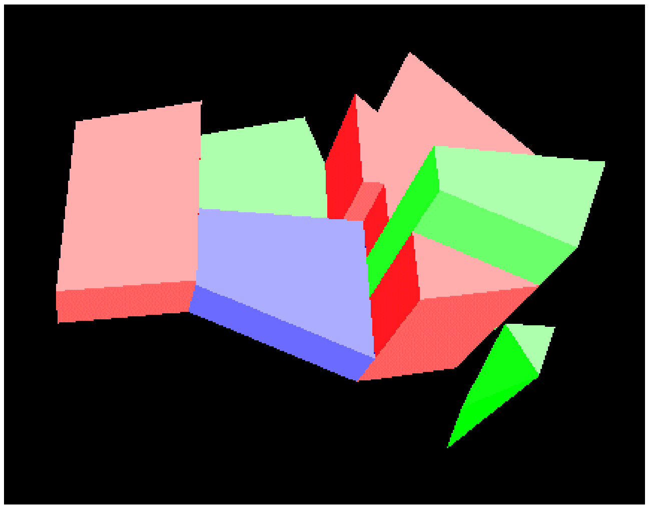

Figure 6: Number of Population in Australia

First we specify that adjacent states are not assigned the same color:

```
  constraints [
    "WS.emissiveColor!=NT.emissiveColor"
    "WS.emissiveColor!=SA.emissiveColor"
    "NT.emissiveColor!=SA.emissiveColor"
    "NT.emissiveColor!=Q.emissiveColor"
    "SA.emissiveColor!=Q.emissiveColor"
    "SA.emissiveColor!=NSW.emissiveColor"
    "SA.emissiveColor!=V.emissiveColor"
    "Q.emissiveColor!=NSW.emissiveColor"
    "NSW.emissiveColor!=V.emissiveColor"
    "V.emissiveColor!=T.emissiveColor"
    ]
```

Now we define three domains `M6`, `M4` and `M2` as finite sets of colors

```
  domainDefs [
    "MFColor M6 { 0 0 1, 1 0 0, 0 1 0, 1 0 1,
                  1 1 0, 0 1 1 }"
    "MFColor M4 { 0 0 1, 1 0 0, 0 1 0, 1 0 1  }"
    "MFColor M2 { 1 1 0, 0 1 1 }"
    ]
```

and for earch field we restrict its possible value to be in one of these sets.

```
  domains [
    "WS.emissiveColor=M6"
    "NT.emissiveColor=M6"
    "Q.emissiveColor=M4"
    "SA.emissiveColor=M4"
    "NSW.emissiveColor=M4"
    "V.emissiveColor=M2"
    "T.emissiveColor=M2"
   ]
 }
```

Figure 6 shows a map of australia, the height of each column indicates the number of inhabitants in that part of the country.

Another problem that is often discussed as an example for finite domain constraints is the N-Queens problem. We have also used it as a test case for our extension, see Figure 7.

4 Implementation

Most of the information in the fields of the `Constraint` prototype are provided as strings. The Java program parses these strings

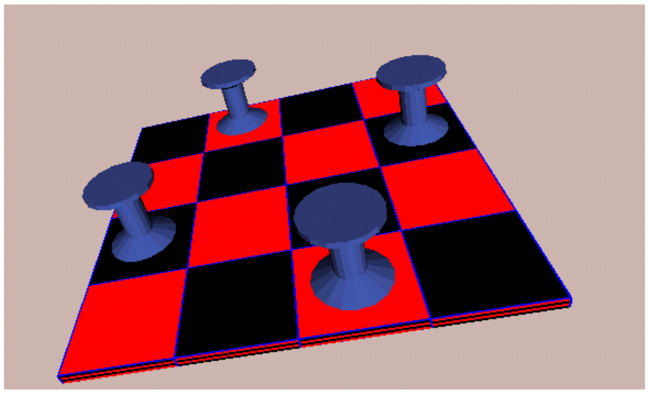

Figure 7: N-Queens problem

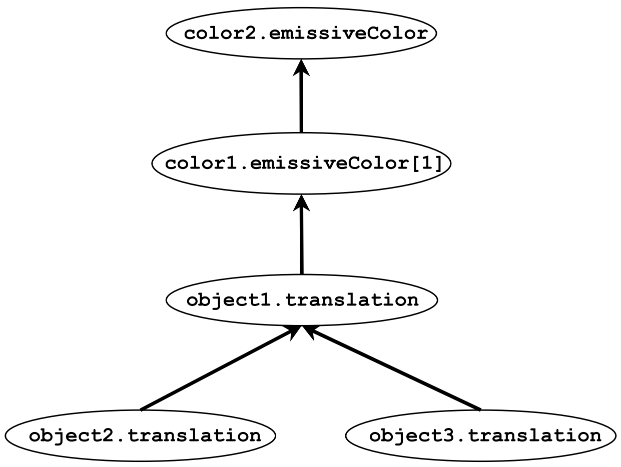

Figure 8: Dependency graph

and constructs a constraint graph. This graph makes explicit on which variables and constraints a variable depends.

The algorithm used to solve the constraints is described in [10]. This algorithm works in two phases: The nullification phase and the re-evaluation phase. If a value of a variable changes, e.g. as result of an event `e` , all variables that directly or indirectly depend on this changed variable are marked out-of-date (nullification phase). Then, recursively the values of all variables are computed, that depend on the changed variables (reevaluation phase). If a variable has no successor or all successors are up-to-date, the recursion will stop. Notice, that the value of the event `e` will also be re-evaluated in case of `eventFirstPriority FALSE` .

Consider the following Example:

```
 Constraint {
  inames [ "object1" "object2" "color1" "color2 ]
  inodes [ USE object1 USE ob-
ject2 USE color1 USE color2 ]
  constraints [
   "object1.translation=Sub(object2.translation,
                            object3.translation)"
   "color1.emissiveColor[1]=Div(object1.translation[1],
                                10)"
   "color2.emissiveColor=color1.emissiveColor[1] "
   ]
}
```

Figure 8 shows the dependency-graph of this example.

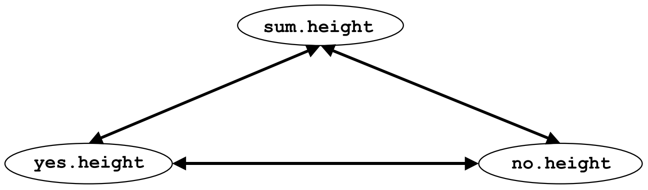

Figure 9: A cyclic dependency graph

Now consider the previously discussed example with the simulation of the multi-way constraints:

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

Figure 9 shows the cyclic dependency-graph of this example.

If now for example the value of `yes.height` changes, `sum.height` and `no.height` will be set out-of-date (nullification phase). But consider, that `sum.height` and `no.height` depend both on `yes.height` but also on each other. It is not important, which variable will be computed first. Assume that `sum.height` will be computed first. It will be marked up-to-date and by computing its constraint, `yes.height` and `no.height` will be requested. `Yes.height` is up-to-date, but `no.height` is still marked out-of-date. So, `no.height` will be computed first and after this, the solver computes the constraint of `sum.height` (re-evaluation phase). Note, that the constraints should be consistent. Otherwise, one or more constraints will not be satisfied.

By allowing the user to define own functions the constraints become more expressive and flexible. To implement this feature the constraint solver creates a new Script-node by the JS API-function `createVRMLFromString()` . For each function parameter an EventIn-Field will be created. To return the result, we simply use an EventOut. We also allow to use such a function several times in the constraints. Solving a constraint with a user function will be done in 3 steps: First, all EventIn's are updated by `setValue()` . Then, the JavaScript-Function will compute the return-Value. Last, the constraintSolver gets the result by `getValue()` . These 3 steps efficiently allow the solver to use only one Script-node for a function but to use the function several times in the constraints.

5 Related Work

A first 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 certain browser [1]. At a workshop at VRML98 Nadine Richard presented a first Java-based implementation of constraints [8]. The constraints had been hard coded in the constraint solver in Java. In this paper we have shown, how constraints and solver can be separated and thus, how constraints become part of the VRML file.

6 Future Work

Our current implementation is working, but it is relatively slow. There are two directions which we will investigate to make it more efficient. First the number of Java Scripting API calls has to be reduced, second more efficient finite domain solving techniques must be included.

In programming languages constraints turned out to be quite expressive when it comes to communication and synchronization in distributed virtual worlds [5]. It will be interesting to investigate how this approach could be combined with the LivingWorlds proposal or similar multi-user extensions.

7 Conclusions

We have presented constraints as an extension of VRML and described our experimental implementation. We feel that constraints provide a powerful, expressive and natural way to specify dependencies between fields of different nodes and that they should replace routes in a future VRML standard.

References

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

[2] 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.

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

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

[5] Seif Haridi, Peter van Roy, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 3(16), 1998.

[6] 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.

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

[8] 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.

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

[10] 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.