A Geometric Constraint Library for 3D Graphical Applications

Hiroshi Hosobe
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
hosobe@nii.ac.jp

ABSTRACT

Recent computer technologies have enabled fast high-quality 3D graphics on personal computers, and also have made the development of 3D graphical applications easier. However, most of such technologies do not sufficiently support layout and behavior aspects of 3D graphics. Geometric constraints are, in general, a powerful tool for specifying layouts and behaviors of graphical objects, and have been applied to 2D graphical user interfaces and specialized 3D graphics packages. In this paper, we present Chorus3D, a geometric constraint library for 3D graphical applications. It enables programmers to use geometric constraints for various purposes such as geometric layout, constrained dragging, and inverse kinematics. Its novel feature is to handle scene graphs by processing coordinate transformations in geometric constraint satisfaction. We demonstrate the usefulness of Chorus3D by presenting sample constraint-based 3D graphical applications.

Keywords

geometric constraints, constraint satisfaction, geometric layout, 3D graphics, scene graphs

1. INTRODUCTION

Recent advances in commodity hardware have enabled fast high-quality 3D graphics on personal computers. Also, software technologies such as VRML and Java 3D have made the development of 3D graphical applications easier. However, most of such technologies mainly focus on rendering aspects of 3D graphics, and do not sufficiently support layout and behavior aspects.

Constraints are, in general, a powerful tool for specifying layouts and behaviors of graphical objects. It is widely recognized that constraints facilitate describing geometric layouts and behaviors of diagrams in 2D graphical user interfaces such as drawing editors, and therefore *constraint solvers* for this purpose have been extensively studied [3, 7, 8, 9, 11, 12, 13, 17, 18]. Also, many specialized 3D graphics packages enable the specification of object layouts and behaviors by using constraints or similar functions.

It is natural to consider that various 3D graphical applications can also be enhanced by incorporating constraints. It might seem sufficient for this purpose to modify existing 2D geometric constraint solvers to support 3D geometry. It is, however, insufficient in reality because of the essential difference between the ways of specifying 2D and 3D graphics; typical 2D graphics handles only simple coordinate systems, whereas most 3D graphics requires multiple coordinate systems with complex relations such as rotations to treat *scene graphs*. It means that we need to additionally support coordinate transformations in 3D geometric constraint solvers.

In this paper, we present Chorus3D, a geometric constraint library for 3D graphical applications. The novel feature of Chorus3D is to handle scene graphs by processing coordinate transformations in geometric constraint satisfaction. We have realized Chorus3D by adding this feature to our previous 2D geometric constraint library Chorus [13].

Another important point of Chorus3D is that it inherits from Chorus the capability to handle "soft" constraints with hierarchical *strengths* or preferences (i.e., constraint hierarchies [7]), which are useful for specifying default layouts and behaviors of graphical objects. It determines solutions so that they satisfy as many strong constraints as possible, leaving weaker inconsistent constraints unsatisfied.

Chorus3D also inherits from Chorus a *module mechanism* which allows user-defined kinds of geometric constraints. This feature enables programmers to use geometric constraints for various purposes including the following:

Geometric layout: A typical use of Chorus3D is to lay out graphical objects. For example, it allows putting objects parallel or perpendicular to others without requiring predetermined positioning parameters. Also, it provides constraint-based general graph layout based on the spring model [14].

Constrained dragging: Chorus3D enables dragging objects with positioning constraints. For example, it can constrain a dragged object to be on the surface of a sphere. Constrained dragging is important for 3D graphics because it provides a sophisticated way to ac-

commodate ordinary mouse dragging to 3D spaces.

Inverse kinematics: Chorus3D is applicable to inverse kinematics, which is a problem of finding desired configurations of "articulated" objects [1, 20]. It allows the specification of articulated objects by using coordinate transformations, and can automatically calculate the parameters of the transformations that satisfy constraints. This method is also applicable to camera control by aiming at a possibly moving target object.

In this paper, we demonstrate the usefulness of Chorus3D by presenting sample constraint-based 3D graphical applications.

This paper is organized as follows: We first present our approach to the use of constraints for 3D graphics. Second, we describe our basic framework of constraints. Next, we present a method for processing coordinate transformations in our framework. We then provide the implementation of Chorus3D, and demonstrate examples of using constraints in 3D graphics. After giving related work and discussion, we mention the conclusions and future work of this research.

2. OUR APPROACH

In this research, we integrate geometric constraints with 3D graphics. Basically, we realize this by extending our previous 2D geometric constraint solver Chorus [13] to support 3D geometry. However, as already mentioned, it is not a straightforward task because 3D graphics typically requires handling scene graphs with hierarchical structures of coordinate systems, which is not covered by the 2D version of the Chorus constraint solver.

To support hierarchies of coordinate systems, we introduce the following new model of constraints:

Point variables: Each point variable (which consists of three real-valued constrainable variables) is associated with one coordinate system, and its value is expressed as local coordinates.

Geometric constraints: Geometric constraints on point variables are evaluated by using the world coordinates of the point variables (they can also refer to 1D variables for, e.g., distances and angles by using their values directly). A single constraint can refer to point variables belonging to different coordinate systems.

Coordinate transformations: Parameters of coordinate transformations are provided as constrainable variables, and the solver is allowed to change the parameters of transformations to appropriately satisfy given constraints.

With this model, we can gain the benefit of the easy maintenance of geometric relations by using constraints, as well as the convenience of modeling geometric objects by employing scene graphs.

In our actual implementation, we provide the following three elemental kinds of coordinate transformations:

Translation: A translation transformation is characterized with three variables t_x, t_y, and t_z, and specifies the translation of vector (t_x, t_y, t_z).

Rotation: A rotation transformation is parameterized with four variables r_x, r_y, r_z, and r_w, and specifies the rotation of angle r_w about the axis (r_x, r_y, r_z).

Scale: A scale transformation is represented with three variables s_x, s_y, and s_z, and specifies the axis-wise scale (s_x, s_y, s_z) about the origin.

We can express many practically useful transformations by using such elemental ones. In fact, any transformations represented with `Transform` nodes in VRML can be realized by combining these kinds of transformations [4].

3. CONSTRAINT FRAMEWORK

In this section, we briefly describe our framework for handling constraints. We base it on the framework for the 2D version of the Chorus constraint solver. See [13] for further detail.

3.1 Problem Formulation

We first present the mathematical formulation for modeling constraints and constraint systems. In the following, we write $\boldsymbol{x}$ to represent a variable vector $(x_1, x_2, \ldots, x_n)$ of n variables, and also $\boldsymbol{v}$ to indicate a variable value vector $(v_1, v_2, \ldots, v_n)$ of n real numbers (v_i expresses the value of x_i).

To support various geometric constraints in a uniform manner, we adopt *error functions* as a means of expressing constraints. An error function $e(\boldsymbol{x})$ is typically associated with a single arithmetic constraint, and is defined as a function from variable value vectors to errors expressed as non-negative real numbers; that is, $e(\boldsymbol{v})$ gives the error of the associated constraint for $\boldsymbol{v}$. An error function returns a zero if and only if the constraint is exactly satisfied. For example, $e(\boldsymbol{x}) = (x_i - x_j)^2$ can be used for the constraint $x_i = x_j$. We assume that, for each $e(\boldsymbol{x})$, its gradient is known:

$$\nabla e(\boldsymbol{x}) = \left(\frac{\partial e(\boldsymbol{x})}{\partial x_1}, \frac{\partial e(\boldsymbol{x})}{\partial x_2}, \ldots, \frac{\partial e(\boldsymbol{x})}{\partial x_n} \right).$$

In the same way as constraint hierarchies [7], constraint systems in our framework can be divided into levels consisting of constraints with equal strengths. Constraints with the strongest preference are said to be required (or hard), and are guaranteed to be always satisfied (if it is impossible, there will be no solution). By contrast, constraints with weaker preferences are said to be preferential (or soft), and may be relaxed if they conflict with stronger constraints.

Solutions to constraint systems are defined as follows: let $e_{i,j}(\boldsymbol{x})$ be the error function of the j-th constraint ($1 \leq j \leq m_i$) at strength level i ($0 \leq i \leq l$); then solutions $\boldsymbol{v}$ are determined with the optimization problem

$$\underset{\boldsymbol{v}}{\text{minimize}} \;\; E(\boldsymbol{v}) \;\; \text{subject to} \;\; e_{0,j}(\boldsymbol{v}) = 0 \;\; (1 \leq j \leq m_0)$$

where E is an objective function defined as

$$E(\boldsymbol{x}) = \sum_{i=1}^{l} \sum_{j=1}^{m_i} w_i e_{i,j}(\boldsymbol{x})$$

in which w_i indicates the weight associated with strength i, and the relation $w_1 \gg w_2 \gg \cdots \gg w_l$ holds. In this formulation, level 0 corresponds to required constraints, and the others to preferential ones. Intuitively, more weighted (or stronger) preferential constraints should be more satisfied.

Our framework simulates constraint hierarchies. Particularly, if the squares of constraint violations are used to compute error functions, a system in our framework will obtain approximate solutions to the similar hierarchy solved with the criterion least-squares-better [3, 17]. The largest difference is that a system in our framework slightly considers a weak constraint inconsistent with a stronger satisfiable one in computing its solutions, while the similar hierarchy would discard such a weak one.

Our actual implementation of the Chorus3D constraint solver provides four external strengths `required`, `strong`, `medium`, and `weak` as well as two internal strengths `very strong` (used to approximately handle `required` nonlinear or inequality constraints) and `very weak` (exploited to make new solutions as close to previous ones as possible). It typically assigns weights 32^4, 32^3, 32^2, 32^1, and 1 to strengths `very strong`, `strong`, `medium`, `weak`, and `very weak` respectively. These weights were determined according to the precision of the actual numerical algorithm (described in the next subsection). To know how much these weights affect solutions, suppose a system of `strong` constraint $x = 0$ and `medium` one $x = 100$. Then the unique solution will be obtained as $x = 3.0303 \cdots (= 100/33)$. Thus the difference of strengths is obvious. According to our actual experience, this precision allows us to discriminate constraint strengths in most graphical applications.

3.2 Algorithm

To actually find solutions to constraint systems presented above, we need to solve their corresponding optimization problems. For this purpose, we designed a constraint satisfaction algorithm by combining a numerical optimization technique with a genetic algorithm. It uses numerical optimization to find local solutions, while it adopts a genetic algorithm to search for global solutions.

For numerical optimization, we mainly use the quasi-Newton method based on Broyden-Fletcher-Goldfarb-Sahnno updating formula [2, 6], which is a fast iterative technique that exhibits superlinear convergence. Since it excludes fruitless searches by utilizing its history, it is usually faster than straightforward Newton's method.

We introduced a genetic algorithm to alleviate the problem that some kinds of geometric constraints suffer from local optimal but global non-optimal solutions [11, 16]. Generally, a genetic algorithm is a stochastic search method that repeatedly transforms a population of potential solutions into another next-generation population [10, 15]. We typically necessitate it only for computing initial solutions; in other words, we can usually re-solve modified constraint systems without the genetic algorithm, only by applying numerical optimization to previous solutions.

4. PROCESSING COORDINATE TRANSFORMATIONS

In this section, we propose a method for integrating coordinate transformations with our constraint framework.

As already mentioned, we use world coordinates of points to evaluate 3D geometric constraints. A naive method for this is to duplicate point variables in all ancestor coordinate systems, and then to impose required constraints that represent coordinate transformations between the point variables. However, this method requires an optimization routine supporting required nonlinear constraints, which limits the availability of actual techniques (in fact, we cannot use the quasi-Newton method for this purpose). Also, this method tends to yield many variables and constraints, and therefore requires an extra amount of memory.

Below we propose a more widely applicable method for handling coordinate transformations. Its characteristic is to hide transformations from optimization routines, which is realized by embedding transformations in error functions.

4.1 Model

To begin with, we introduce another variable vector $\boldsymbol{x}' = (x'_1, x'_2, \ldots, x'_n)$, which is created by replacing variables for local coordinates of 3D points in $\boldsymbol{x}$ with the corresponding ones for world coordinates (1D variables remain the same). We can mathematically model this process as follows: Consider the sequence of the s transformations

$$\boldsymbol{y}_0\,(=\boldsymbol{x}) \xrightarrow{\boldsymbol{t}_0} \boldsymbol{y}_1 \xrightarrow{\boldsymbol{t}_1} \cdots \xrightarrow{\boldsymbol{t}_{s-2}} \boldsymbol{y}_{s-1} \xrightarrow{\boldsymbol{t}_{s-1}} \boldsymbol{y}_s\,(=\boldsymbol{x}')$$

where $\boldsymbol{y}_0$ and $\boldsymbol{y}_s$ are equal to $\boldsymbol{x}$ and $\boldsymbol{x}'$ respectively, each $\boldsymbol{y}_k$ $(1 \le k \le s-1)$ is an "intermediate" vector, and each $\boldsymbol{t}_k$ $(0 \le k \le s-1)$ is a function that transforms $\boldsymbol{y}_k$ into $\boldsymbol{y}_{k+1}$. Intuitively, $\boldsymbol{t}_k$ corresponds to a coordinate transformation, and transforms related point variables from its source coordinate system into its destination system. It should be noted that, although transformations are, in general, hierarchical (or tree-structured), we can always find such a linear sequence by "serializing" them in an appropriate order.

By using such transformations, we can compute $\boldsymbol{x}'$ as follows:

$$\boldsymbol{x}' = \boldsymbol{t}_{s-1}(\boldsymbol{t}_{s-2}(\cdots(\boldsymbol{t}_1(\boldsymbol{t}_0(\boldsymbol{x})))\cdots)) \equiv \boldsymbol{t}(\boldsymbol{x})$$

where $\boldsymbol{t}$ is defined as the composition of all the elemental transformations. In the following description, we write $y_{k,i}$ to denote the i-th element of $\boldsymbol{y}_k$, and also $t_{k,i}$ to represent the i-th element of $\boldsymbol{t}_k$; that is,

$$\begin{aligned}\boldsymbol{y}_{k+1} &= (y_{k+1,1}, y_{k+1,2}, \ldots, y_{k+1,n}) \\ &= (t_{k,1}(\boldsymbol{y}_k), t_{k,2}(\boldsymbol{y}_k), \ldots, t_{k,n}(\boldsymbol{y}_k)) = \boldsymbol{t}_k(\boldsymbol{y}_k).\end{aligned}$$

4.2 Method

Geometric constraints are evaluated by using world coordinates of points, which means that their error functions are

defined as $e(\boldsymbol{x}')$. Using the composed transformations, we can evaluate them as

$$e(\boldsymbol{x}') = e(\boldsymbol{t}(\boldsymbol{x})).$$

Importantly, we can efficiently realize this computation by applying only necessary transformations to actually used variables.

We also need to compute the gradient of $e(\boldsymbol{t}(\boldsymbol{x}))$, i.e.,

$$\nabla e(\boldsymbol{t}(\boldsymbol{x})) = \left(\frac{\partial e(\boldsymbol{t}(\boldsymbol{x}))}{\partial x_1}, \frac{\partial e(\boldsymbol{t}(\boldsymbol{x}))}{\partial x_2}, \ldots, \frac{\partial e(\boldsymbol{t}(\boldsymbol{x}))}{\partial x_n} \right).$$

Basically, we can decompose each partial derivative $\partial e(\boldsymbol{t}(\boldsymbol{x}))/\partial x_i$ into primitive expressions by repeatedly using the chain rule. However, we should avoid the simple application of the chain rule since it would result in a large number of expressions.

Instead, we perform a controlled way of decomposing such partial derivatives; it appropriately arranges the chain rule to restrict the computation to only necessary components. First, we decompose $\partial e(\boldsymbol{t}(\boldsymbol{x}))/\partial x_i$ as follows:

$$\begin{aligned}
\frac{\partial e(\boldsymbol{t}(\boldsymbol{x}))}{\partial x_i} &= \sum_{j'} \frac{\partial e(\boldsymbol{x}')}{\partial x'_{j'}} \frac{\partial t_{s-1,j'}(\boldsymbol{y}_{s-1})}{\partial x_i} \\
&= \sum_{j'} \frac{\partial e(\boldsymbol{x}')}{\partial x'_{j'}} \sum_{j_{s-1}} \frac{\partial t_{s-1,j'}(\boldsymbol{y}_{s-1})}{\partial y_{s-1,j_{s-1}}} \frac{\partial t_{s-2,j_{s-1}}(\boldsymbol{y}_{s-2})}{\partial x_i} \\
&= \sum_{j_{s-1}} \left\{ \sum_{j'} \frac{\partial e(\boldsymbol{x}')}{\partial x'_{j'}} \frac{\partial t_{s-1,j'}(\boldsymbol{y}_{s-1})}{\partial y_{s-1,j_{s-1}}} \right\} \frac{\partial t_{s-2,j_{s-1}}(\boldsymbol{y}_{s-2})}{\partial x_i} \\
&= \sum_{j_{s-1}} \frac{\partial e(\boldsymbol{x}')}{\partial y_{s-1,j_{s-1}}} \frac{\partial t_{s-2,j_{s-1}}(\boldsymbol{y}_{s-2})}{\partial x_i}.
\end{aligned}$$

Note that each $\partial e(\boldsymbol{x}')/\partial x'_{j'}$ is given by the definition of the geometric constraint, and also that each $\partial t_{s-1,j'}(\boldsymbol{y}_{s-1})/\partial y_{s-1,j_{s-1}}$ is a partial derivative in the gradient of a single coordinate transformation $\boldsymbol{t}_{s-1}$. Thus we can obtain each $\partial e(\boldsymbol{x}')/\partial y_{s-1,j_{s-1}}$. Also, by repeating this process, we can compute, for each k,

$$\frac{\partial e(\boldsymbol{t}(\boldsymbol{x}))}{\partial x_i} = \sum_{j_k} \frac{\partial e(\boldsymbol{x}')}{\partial y_{k,j_k}} \frac{\partial t_{k-1,j_k}(\boldsymbol{y}_{k-1})}{\partial x_i}$$

and finally achieve

$$\frac{\partial e(\boldsymbol{t}(\boldsymbol{x}))}{\partial x_i} = \sum_{j_1} \frac{\partial e(\boldsymbol{x}')}{\partial y_{1,j_1}} \frac{\partial t_{0,j_1}(\boldsymbol{x})}{\partial x_i}$$

where each $\partial t_{0,j_1}(\boldsymbol{x})/\partial x_i$ is a component of the gradient of $\boldsymbol{t}_0$. Therefore, $\partial e(\boldsymbol{t}(\boldsymbol{x}))/\partial x_i$ is now determined.

Furthermore, we can considerably reduce the number of the computations of $\partial e(\boldsymbol{x}')/\partial y_{k,j_k}$ in practice. We can make the following observations about the above computation:

- For each variable $x_{j'}$, $\partial e(\boldsymbol{x}')/\partial x'_{j'}$ can be non-zero only if $x_{j'}$ is actually needed to evaluate the designated constraint.
- If x_i is originated in the coordinate system associated with $\boldsymbol{t}_k$ (that is, x_i is either a local coordinate or a parameter of the coordinate transformation), we have $y_{k,i} = x_i$, which means that we have $\partial t_{k,j}(\boldsymbol{y}_k)/\partial x_i$. Therefore, we can compute $\partial e(\boldsymbol{x}')/\partial x_i$ immediately.

These observations reveal that we need to transfer a partial derivative $\partial e(\boldsymbol{x}')/\partial y_{k,j}$ to the next step only when x_j represents a really necessary coordinate that has not reached its local coordinate system. Also, since we can handle each necessary point independently, we can implement this process with a linear recursive function that hands over only three derivatives $\partial e(\boldsymbol{x}')/\partial y_{k,j}$ at each recursive call.

5. IMPLEMENTATION

We implemented the proposed method by developing a constraint solver called Chorus3D, which is a 3D extension to our previous 2D geometric constraint solver Chorus [13]. We constructed Chorus3D as a C++ class library, and also developed a native method interface to make it available to Java programs.

Chorus3D allows programmers to add a new kind of arithmetic constraints (e.g., Euclidean geometric constraints) by constructing a new constraint class with a method that evaluates their error functions. Also, programmers can introduce a new kind of non-arithmetic (or pseudo) constraints (for, e.g., general graph layout) by developing a new evaluation module which computes an "aggregate" error function for a given set of constraints.

Chorus3D currently provides linear equality, linear inequality, edit (update a variable value), stay (fix a variable value), Euclidean geometric constraints (for, e.g., parallelism, perpendicularity, and distance equality), and graph layout constraints based on the spring model [14]. Linear equality/inequality constraints can refer to only 1D variables (including elements of 3D point variables), while edit and stay constraints can be associated with 1D and 3D point variables. Euclidean geometric constraints typically refer to point variables although they sometimes require 1D variables for angles and distances. Each graph layout constraint represents a graph edge, and refers to two point variables as its associated graph nodes. As stated earlier, constraints on such point variables are evaluated by using world coordinates of the points. Also, a single constraint can refer to point variables belonging to different coordinate systems.

The application programming interface of Chorus3D is a natural extension to that of Chorus, which provides a certain compatibility with a recent linear solver called Cassowary [3]; in a similar way to Cassowary and Chorus, Chorus3D allows programmers to process constraint systems by creating variables and constraints as objects, and by adding/removing constraint objects to/from the solver object. In addition, Chorus3D handles coordinate transformations as objects, and presents an interface for arranging them hierarchically.

6. EXAMPLES

In this section, we present three examples to demonstrate how to incorporate geometric constraints into 3D graphics by using the Chorus3D constraint solver. All the examples are implemented in Java by using Java 3D as a graphics

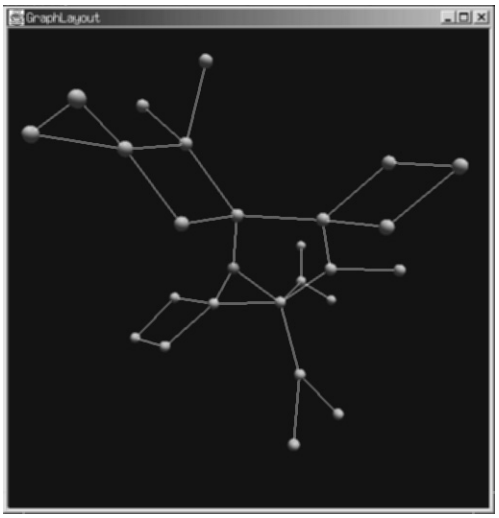

Figure 1: A 3D geometric layout of a general graph structure.

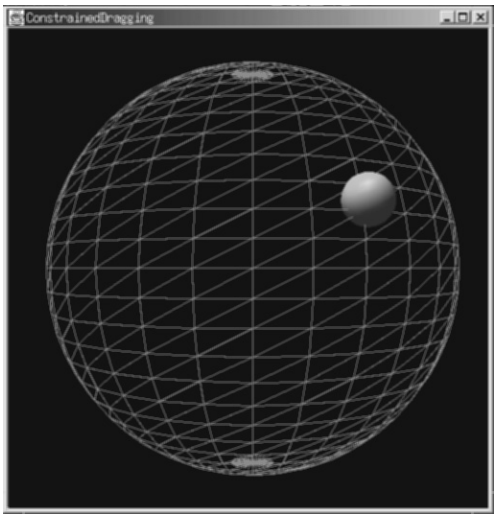

Figure 2: Dragging an object constrained to be on a sphere.

programming interface as well as the native method interface with Chorus3D. We also provide computation times taken for constraint satisfaction in these examples.

6.1 Graph Layout

The first example is an application which lays out a set of points with a general graph structure in a 3D space as shown in Figure 1. This application also allows a user to drag graph nodes with a mouse.[1] The used graph layout technique is based on a 3D extension to the spring model [14]. This kind of 3D graph layout is practically useful to information visualization, and has actually been adopted in a certain system [19].

The constraint system of this graph layout consists of 26 point variables (i.e., 78 real-valued variables), 31 graph layout constraints, and three linear equality constraints for fixing one of the point variables at the origin. When executed on an 866 MHz Pentium III processor running Linux 2.2.16, Chorus3D obtained an initial solution in 456 milliseconds. It performed constraint satisfaction typically within 250 milliseconds to reflect the user's dragging a graph node.

6.2 Constrained Dragging

The second example is an application which allows a user to drag an object constrained to be on another spherical object. Figure 2 depicts this application, where the smaller solid spherical object is constrained to be on the surface of the larger wireframe one. The application declares a `strong` Euclidean geometric constraint which specifies a constant distance between the centers of these objects. When the user tries to drag the smaller object with a mouse, the application imposes another `medium` Euclidean constraint which collinearly locates the viewpoint, the 3D position of the mouse cursor (which is considered to be on the screen), and

[1]Unlike constrained dragging in the next example, this mouse operation is simply implemented with Java 3D's `PickMouseBehavior` classes.

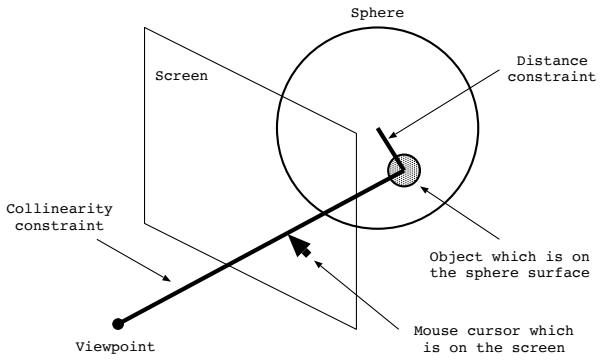

Figure 3: Implementation of constrained dragging.

the center of the dragged object as shown in Figure 3. This collinearity constraint reflects the motion of the mouse in the position of the dragged object. Since the collinearity constraint is weaker than the first Euclidean constraint, the user cannot drag the smaller object to the outside of the larger sphere.

The application initially declares one Euclidean geometric constraint on two point variables, and solved it in 1 millisecond on the same computer as the first example. When the user tries to drag the smaller object, it adds another Euclidean constraint as well as two edit constraints for the viewpoint and mouse position. The solver maintained this constraint system usually within 2 milliseconds.

6.3 Inverse Kinematics

The final example applies inverse kinematics to a virtual robot arm by using constraints. Unlike the previous examples, it takes advantage of coordinate transformations to express its constraint system.

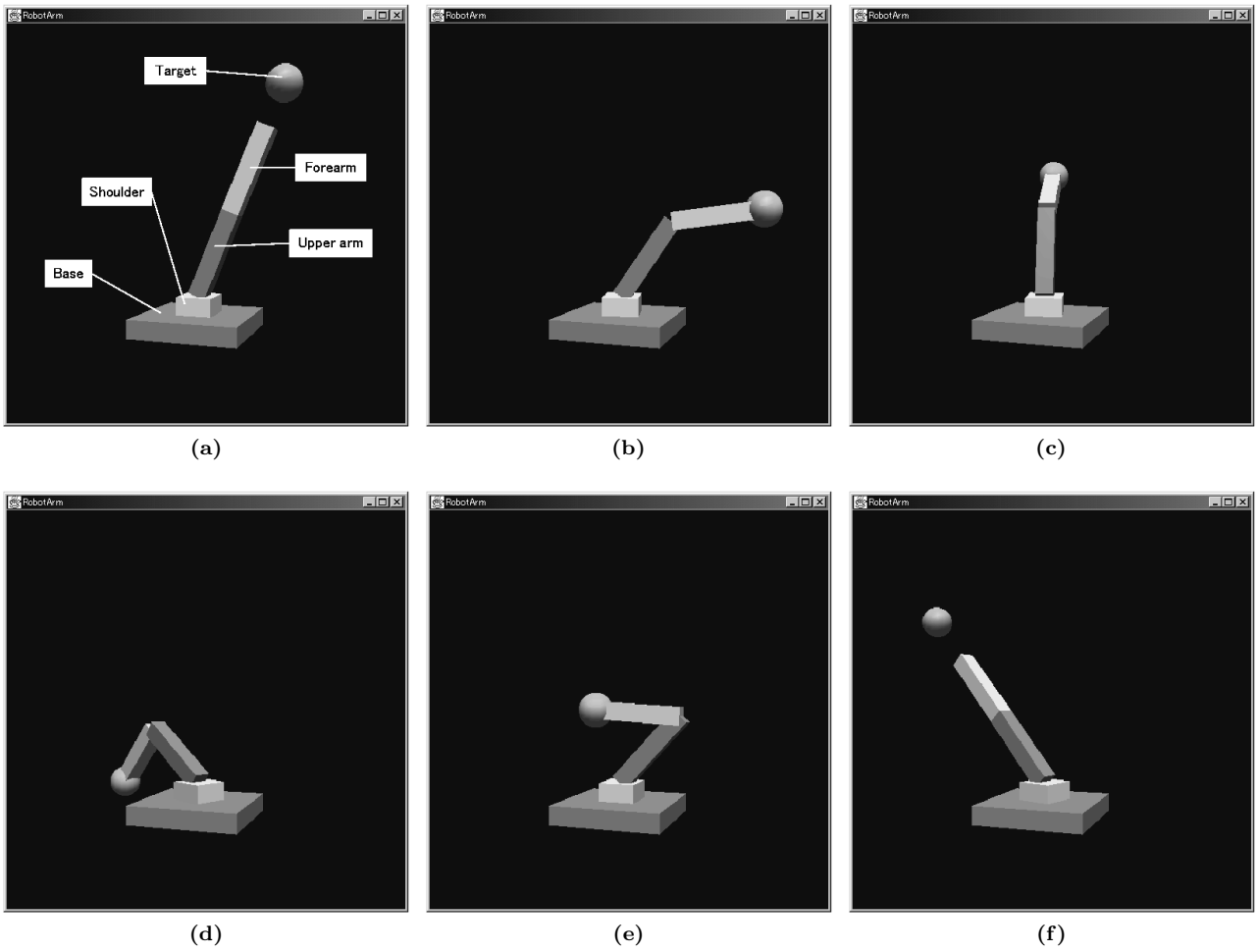

Figure 4: A robot arm application which performs inverse kinematics.

As illustrated in Figure 4(a), the robot arm consists of four parts called a base, a shoulder, an upper arm, and a forearm. Constraint satisfaction for inverse kinematics is performed to position its hand (the end of the forearm) at the target object if possible, or otherwise to make it maximally close to the target. Figures 4(b)–(f) show the movement of the robot arm. In Figures 4(b)–(e), its hand is positioned at the exact location of the target by using appropriate angles of its joints. By contrast, in Figure 4(f), the hand cannot reach the target, and therefore the arm is extended toward the target instead.

Figure 5 describes the constraint program used in the robot arm application. After constructing a constraint solver `s`, it creates six coordinate transformations `shldrTTfm`, `shldrRTfm`, `uarmTTfm`, `uarmRTfm`, `farmTTfm`, and `farmRTfm`. Here the rotation angle parameters of the rotation transformations `shldrRTfm`, `uarmRTfm`, and `farmRTfm` will actually work as variables that can be altered by the solver. Next, it generates a point variable `handPos` to represent the position of the hand, and then suggests the target position to the hand by using a preferential edit constraint `editHandPos`. Finally, executing the solver, it obtains the desired angles `shldrAngle`, `uarmAngle`, and `farmAngle` of the rotation transformations. These angles will be passed to the Java 3D library to render the properly configured robot arm.

This program generates a constraint system which contains three translation and three rotation transformations, one explicit point variable as well as six point variables and three 1D variables for coordinate transformations, and one edit constraint. The solver found an initial solution to this system in 18 milliseconds, and obtained each new solution for a frame update typically within 10 milliseconds.

7. RELATED WORK AND DISCUSSION

There has been work on integrating constraints or similar functions with 3D graphics languages to facilitate the specification of graphical objects. For example, we can view the event routing mechanism in VRML [4] as a limited form of one-way propagation constraints. Also, there is an attempt to extend VRML by introducing one-way propagation and finite-domain combinatorial constraints [5]. However, they cannot handle more powerful simultaneous nonlinear constraints such as Euclidean geometric constraints.

Although many constraint solvers have been developed in

```
// constraint solver
s = new C3Solver();
// translation transformation for the shoulder: fixed to (0, .1, 0)
shldrTTfm = new C3TranslateTransform(new C3Domain3D(0, .1, 0));
s.add(shldrTTfm); // shldrTTfm is parented by the world coordinate system
// rotation transformation for the shoulder: axis fixed to (0, 1, 0); angle ranging over [−10000, 10000]
shldrRTfm = new C3RotateTransform(new C3Domain3D(0, 1, 0), new C3Domain(-10000, 10000));
s.add(shldrRTfm, shldrTTfm); // shldrRTfm is parented by shldrTTfm
// translation transformation for the upper arm: fixed to (0, .1, 0)
uarmTTfm = new C3TranslateTransform(new C3Domain3D(0, .1, 0));
s.add(uarmTTfm, shldrRTfm); // uarmTTfm is parented by shldrRTfm
// rotation transformation for the upper arm: axis fixed to (0, 0, 1); angle ranging over [−1.57, 1.57]
uarmRTfm = new C3RotateTransform(new C3Domain3D(0, 0, 1), new C3Domain(-1.57, 1.57));
s.add(uarmRTfm, uarmTTfm); // uarmRTfm is parented by uarmTTfm
// translation transformation for the forearm: fixed to (0, .5, 0)
farmTTfm = new C3TranslateTransform(new C3Domain3D(0, .5, 0));
s.add(farmTTfm, uarmRTfm); // farmTTfm is parented by uarmRTfm
// rotation transformation for the forearm: axis fixed to (0, 0, 1); angle ranging over [−3.14, 0]
farmRTfm = new C3RotateTransform(new C3Domain3D(0, 0, 1), new C3Domain(-3.14, 0));
s.add(farmRTfm, farmTTfm); // farmRTfm is parented by farmTTfm
// variable for the hand's position, associated with farmRTfm and fixed to (0, .5, 0)
handPos = new C3Variable3D(farmRTfm, new C3Domain3D(0, .5, 0));
// medium-strength edit constraint for the hand's position
editHandPos = new C3EditConstraint(handPos, C3.MEDIUM);
s.add(editHandPos);
// suggest the hand being located at the target's position
editHandPos.set(getTargetWorldCoordinates());
// solve the constraint system
s.solve();
// get solutions
double shldrAngle = shldrRTfm.rotationAngle().value();
double uarmAngle = uarmRTfm.rotationAngle().value();
double farmAngle = farmRTfm.rotationAngle().value();
```

Figure 5: Constraint program for the robot arm application.

the field of graphical user interfaces [3, 7, 11, 12, 13, 17, 18], most of them do not provide special treatment for 3D graphics. In general, the role of nonlinear geometric constraints is more important in 3D applications than in 2D interfaces. Most importantly, 3D graphics usually requires rotations of objects which are rarely used in 2D interfaces. The main reason is that we often equally treat all "horizontal" directions in a 3D space even if we may clearly distinguish them from "vertical" directions. Therefore, nonlinear constraint solvers are appropriate for 3D applications. In addition, coordinate transformations should be supported since they are typically used to handle rotations of objects.

Gleicher proposed the differential approach [8, 9], which supports 3D geometric constraints and coordinate transformations. In a sense, it shares a motivation with Chorus3D; in addition to support for 3D graphics, it allows user-defined kinds of geometric constraints. However, it is based on a different solution method from Chorus3D; it realizes constraint satisfaction by running virtual dynamic simulations. This difference results in a quite different behavior of solutions as well as an interface for controlling solutions. By contrast, Chorus3D provides a much more compatible interface with recent successful solvers such as Cassowary [3].

Much research on inverse kinematics has been conducted in the fields of computer graphics and robotics [1, 20]. However, inverse kinematics is typically implemented as specialized software which only provides limited kinds of geometric constraints.

Chorus3D has two limitations in its algorithm: one is on the precision of solutions determined by preferential constraints; the other is on the speed of the satisfaction of large constraint systems. These limitations are mainly caused by the treatment of multi-level preferences of constraints in addition to required constraints (i.e., constraint hierarchies). Although many numerical optimization techniques have been proposed and implemented in the field of mathematical programming [2, 6], most of them do not handle preferential constraints. To alleviate the limitations of Chorus3D, we are pursuing a more sophisticated method for processing multi-level preferential constraints.

We implemented Chorus3D as a class library which can be exploited in C++ and Java programs. However, more high-level authoring tools will also be useful for declarative approaches to 3D design. One possible direction is to extend VRML [4] to support geometric constraints. Standard VRML requires scripts in Java or JavaScript to realize complex layouts and behaviors. By contrast, constraint-enabled VRML will cover a wider range of applications without such additional scripts.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented Chorus3D, a geometric constraint library for 3D graphical applications. It enables programmers to use geometric constraints for various purposes

such as geometric layout, constrained dragging, and inverse kinematics. Its novel feature is to handle scene graphs by processing coordinate transformations in geometric constraint satisfaction.

Our future work includes the development of other kinds of geometric constraints to further prove the usefulness of our approach. In particular, we are planning to implement non-overlapping constraints [13] in Chorus3D so that we can use it for the collision resolution of graphical objects. Another future direction is to improve Chorus3D in the scalability and accuracy of constraint satisfaction.

9. REFERENCES

[1] Badler, N. I., Phillips, C. B., and Webber, B. L. *Simulating Humans: Computer Graphics, Animation, and Control*. Oxford University Press, Oxford, 1993.

[2] Bertsekas, D. P. *Nonlinear Programming*, 2nd ed. Athena Scientific, 1999.

[3] Borning, A., Marriott, K., Stuckey, P., and Xiao, Y. Solving linear arithmetic constraints for user interface applications. In *Proc. ACM UIST*, 1997, 87–96.

[4] Carey, R., Bell, G., and Marrin, C. The Virtual Reality Modeling Language (VRML97). ISO/IEC 14772-1:1997, The VRML Consortium Inc., 1997.

[5] Diehl, S., and Keller, J. VRML with constraints. In *Proc. Web3D-VRML*, ACM, 2000, 81–86.

[6] Fletcher, R. *Practical Methods of Optimization*, 2nd ed. John Wiley & Sons, 1987.

[7] Freeman-Benson, B. N., Maloney, J., and Borning, A. An incremental constraint solver. *Commun. ACM 33*, 1 (1990), 54–63.

[8] Gleicher, M. A graphical toolkit based on differential constraints. In *Proc. ACM UIST*, 1993, 109–120.

[9] Gleicher, M. A differential approach to graphical manipulation (Ph.D. thesis). Tech. Rep. CMU-CS-94-217, Sch. Comput. Sci. Carnegie Mellon Univ., 1994.

[10] Herrera, F., Lozano, M., and Verdegay, J. L. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artif. Intell. Rev. 12*, 4 (1998), 265–319.

[11] Heydon, A., and Nelson, G. The Juno-2 constraint-based drawing editor. Research Report 131a, Digital Systems Research Center, 1994.

[12] Hosobe, H. A scalable linear constraint solver for user interface construction. In *Principles and Practice of Constraint Programming—CP2000*, vol. 1894 of *LNCS*, Springer, 2000, 218–232.

[13] Hosobe, H. A modular geometric constraint solver for user interface applications. In *Proc. ACM UIST*, 2001, 91–100.

[14] Kamada, T., and Kawai, S. An algorithm for drawing general undirected graphs. *Inf. Process. Lett. 31*, 1 (1989), 7–15.

[15] Kitano, H., Ed. *Genetic Algorithms*. Sangyo-Tosho, 1993. In Japanese.

[16] Kramer, G. A. A geometric constraint engine. *Artif. Intell. 58*, 1–3 (1992), 327–360.

[17] Marriott, K., Chok, S. S., and Finlay, A. A tableau based constraint solving toolkit for interactive graphical applications. In *Principles and Practice of Constraint Programming—CP98*, vol. 1520 of *LNCS*, Springer, 1998, 340–354.

[18] Sannella, M. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *Proc. ACM UIST*, 1994, 137–146.

[19] Takahashi, S. Visualizing constraints in visualization rules. In *Proc. CP2000 Workshop on Analysis and Visualization of Constraint Programs and Solvers*, 2000.

[20] Zhao, J., and Badler, N. I. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Trans. Gr. 13*, 4 (1994), 313–336.