Metis
An Object-Oriented Toolkit for Constructing Virtual Reality Applications

Russell Turner[1], Song Li[1], Enrico Gobbetti[2]

1. Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore MD 21250 USA

2. Center for Research, Development, and Advanced Studies in Sardinia
Via Sauro 10
09123 Cagliari, Italy

E-mail
turner@cs.umbc.edu
sli2@cs.umbc.edu
gobbetti@crs4.it

***Abstract*:** Virtual reality systems provide realistic look and feel by seamlessly integrating three-dimensional input and output devices. One software architecture approach to constructing such systems is to distribute the application between a computation-intensive simulator back-end and a graphics-intensive viewer front-end which implements user interaction. In this paper we discuss *Metis*, a toolkit we have been developing based on such a software architecture, which can be used for building interactive immersive virtual reality systems with computationally intensive components. The Metis toolkit defines an application programming interface on the simulator side, which communicates via a network with a standalone viewer program that handles all immersive display and interactivity. Network bandwidth and interaction latency are minimized, by use of a constraint network on the viewer side that declaratively defines much of dynamic and interactive behavior of the application.

1. Introduction

Metis is an object-oriented toolkit for 3D interactive simulation. The goal is to create a simple, flexible high-performance software architecture that enables the rapid construction of immersive virtual reality applications for simulation, utilizing highly interactive techniques such as 3D direct manipulation and virtual tools (or "3D widgets"). Possible areas of application include robotics simulation, 3D character animation, surgical simulation, small-scale multi-user shared environments, and any inherently 3D tasks which require highly interactive user interfaces.

Metis is intended for use in virtual reality applications with varying levels of immersivity, using display techniques ranging from stereo glasses to head-mounted displays, and input devices such as 3D mice and data gloves. For such applications to function properly without inducing user fatigue and motion sickness, they must reliably respond to input and update the display with high frame rates and low latency, all of which require high-performance rendering and simulation capabilities. Metis was designed with a client-server software architecture intended to support these kinds of performance requirements. It also specifies high-level input and output device models so that applications can be developed independently of specific virtual reality hardware configurations, and provides an architecture for constructing interactive virtual tools or "3D widgets". One particular such configuration, which we have been using to test the toolkit, is a form of immersive desktop virtual reality known as "fishtank" VR in which the position of the user's head is tracked as he views the screen through stereo glasses. By altering the projection of the three-dimensional scene in response to the user's head position, objects can be made to appear fixed in space in front of the screen. Using a three-dimensional pointing device, the user can then interact with these virtual objects directly using virtual tools.

Metis is also intended for implementing applications that allow small numbers of users to simultaneously interact in real-time with a computationally intensive simulation. This is achieved using a client-server architecture in which multiple users, each interacting via a local *viewer* client, may connect remotely to a single *simulation* server. This also has the advantage of allowing the simulation portion of the application to be run on a separate machine, which may be better suited for computation, than the viewing portion, which may be better suited for interactive 3D graphics. This type of architecture is well-suited for implementing time-critical rendering and computation, and constant-time rendering capabilities are expected to be added to future versions of Metis.

While the purpose of Metis is not to support large-scale distributed virtual environments or internet-based applications, it could be used to implement Virtual Reality Modeling Language (VRML) browsers. Metis' run-time object structures have been designed to conform as closely as possible to the VRML2.0 standard [SGI96], and it is expected that the toolkit will eventually be able to import VRML2.0 files directly.

2. Design Overview

The Metis toolkit provides several key functional components necessary for implementing most virtual reality applications. These take the form of a renderer for immersive display, a high-level virtual input device model, and a one-way constraint maintenance system, all integrated together using a object-oriented client-server software architecture. The Metis software itself consists of an application program interface, called the Metis *API*. and a standalone interactive 3D program called the Metis *Viewer.* Metis application programmers use the API to construct their own applications, sometimes referred to as *simulators*, which can be viewed and interacted with locally or remotely using the viewer.

2.1 Client-Server Architecture

This division of functionality takes the form of a networked client-server architecture (see Fig. 1). The application simulator, built on top of the API, resides on the server side, while the renderer, constraint maintenance system and input device model reside in the viewer on the client side. When the simulator is started, it waits for a viewer to connect to it via a network connection, through which the simulator may send graphics commands and receive user input.

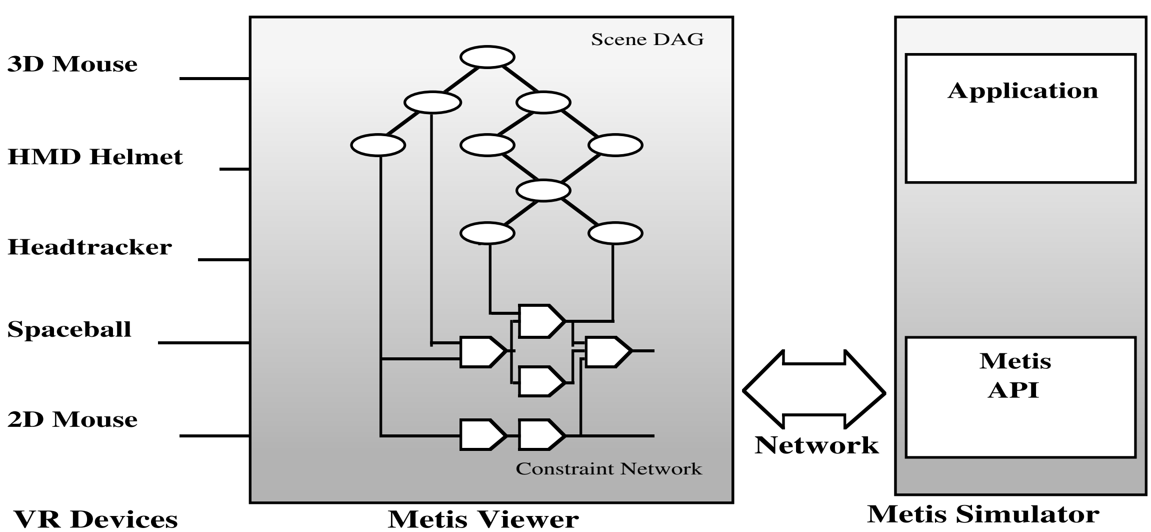

Fig. 1. Metis Client-Server Architecture

This architecture is analogous to the client-server design of X-Windows. The Metis viewer is analogous to an X server in that it provides an interactive graphical interface to the user, while providing output graphics resources and user input data to the application. A Metis application is analogous to an X client in that it provides the application functionality and is implemented on top of the Metis API, which is analogous to the Xlib or Xt API for X. Unlike the X server, however, the Metis viewer provides inherently three dimensional graphics resources, a high-level input device model geared towards virtual reality devices, and a constraint system which allows more interactive functionality to be implemented within the viewer itself.

This kind of client-server architecture, based on communication of objects and constraints, has several important advantages. First of all, the simulation and the viewing process are decoupled: they can therefore run at different rates and can profit from parallel processing, increasing the efficiency of the application. This is similar to modern virtual reality toolkits such as MR [Shaw92], AVIARY [West92], DIVE [Carlsson93] and VIPER [Torguet95]. In contrast to these systems, however, the definition of the behavior on the viewer side is not accomplished using procedural techniques, but rather by using a declarative approach, where the Metis simulator communicates to the viewer not only the virtual world's appearance but also its behavior, specified as a set of time-dependent constraints. Thus, the need for communication between simulator and viewer is less than that of a static scene description, since no communications are needed while the constraints on the viewer's side remain valid. The Cognitive Coprocessor Architecture [Robertson89], and later the TBAG [Elliot94] and VB2 [Gobbetti93] systems, introduced the idea of making time-dependent functions part of the description of graphical primitives in a 3D software architecture in order to declaratively describe simple animated and interactive behaviors. Metis exploits this technique to reduce communication bandwidth and improve load balancing in the context of a client-server VR architecture. Furthermore, interaction latency is minimized, since interaction code can be executed directly in the viewer in the form of constraints. A standard interface between simulators and viewers is defined, specifying the node and constraint types understood by the viewer. This makes it possible to provide a standard generic viewer suitable for a large number of application, allowing programmers to concentrate purely on the simulator side.

2.2 Object-Oriented Design

A Metis application specifies scenes directly in three-dimensional space by creating a scene data structure to specify the geometry, appearance and hierarchical structure, as well as some of the dynamic and interactive behavior, of the virtual environment. This is done from the simulator API by issuing scene creation and linking commands to the viewer, which instantiates the scene structure locally in the viewer. The scene can be accessed from the simulator side by using the instance names of its components, which are simply strings that represent their corresponding real objects on the viewer side. Fig. 2 gives an overview of the main classes in the current viewer architecture.

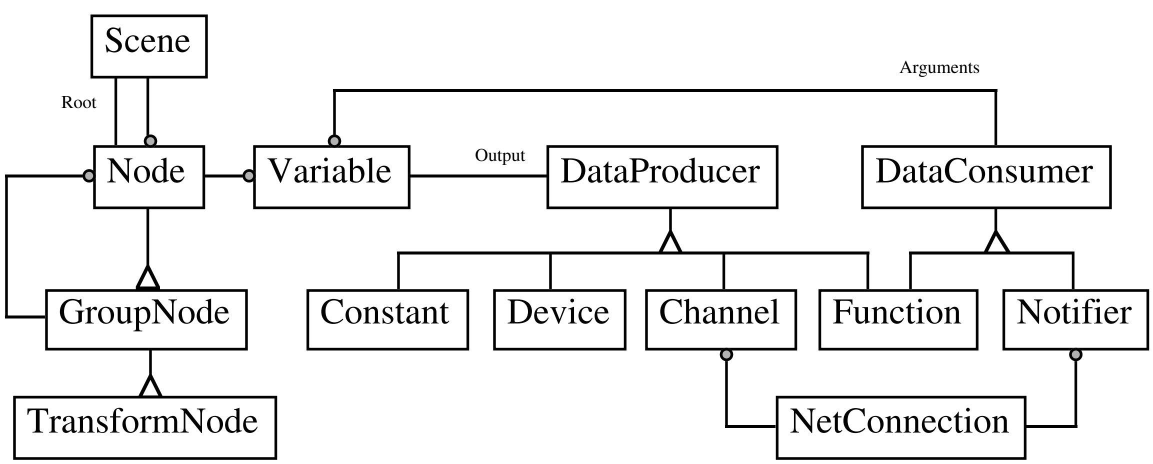

Fig. 2: OMT Diagram of Metis Viewer Scene Classes

The Metis API consists of a collection of C++ classes. Some of these can be extended through subclassing to implement application functionality. Most classes, however, are used to instantiate various components which are assembled through object composition into the simulator-side scene graph which "shadows" the actual scene graph on the viewer side. Overloaded operators, e.g. <<= and =, are used to provide an intuitive syntax. A sample program that uses the Metis API can be found in Appendix A.

3. Metis Scene Architecture

The Metis scene graph itself consists of a directed acyclic graph of nodes, which represents the static graphical state of the scene, and a constraint graph, which maintains dependency relationships between the nodes and implements dynamic behavior. Unlike VRML 2.0 [SGI96], which relies on procedural script nodes to extend dynamic and interactive behavior, Metis uses a purely declarative approach that constructs a network of pre-defined constraints.

3.1 Nodes

A node is an encapsulated data unit that represents some related properties that can be used to describe a static component of a scene. For example, a Sphere node denotes a visible sphere in space, and a Material node specifies color and other visual surface properties of an object. The Metis node system is based on the VRML2.0 node architecture. All externally visible state of a Node is represented by its *variables*, which are active objects containing data of various types. Metis variables can be connected together in dependency relationships using the constraint system. Currently, Metis variables can be one of the following types: Integer, Real, Vector, Quaternion, Color, Matrix. The Integer, Real, Vector, and Color types are standard data types used in computer graphics, while the Quaternion represents a rotation about an arbitrary axis and the Matrix type is a 4x4 homogeneous transformation matrix that is usually used to represent a coordinate system transformation. A Material node, for instance, is defined as follows:

```
Material {
    Real     intensity;
    Color    diffuse;
    Color    emissive;
    Real     shininess;
    Color    specular;
    Real     transparency;
}
```

In Metis, some nodes are distinguish from others by maintaining a group of subnodes. These nodes are called *grouping nodes*. One of the most important amongst them is the Transform node, which separates all its subnodes from the rest of the scene DAG by defining a coordinate system and placing the subnodes in it. The following C++ prototype defines the Transform node, which contains, in addition to the transformation parameters, a list of references to the subnodes.

```
Transform {
    NodeList      children;
    Quaternion    rotation;
    Vector        scale;
    Vector        translation;
    Matrix        matrix;
}
```

One advantage of Metis over other similar systems, e.g. VRML [SGI96], is that it supports a large collection of real three-dimensional devices. In addition to conventional devices like 2D mouse and keyboard, Metis supports a spaceball, headtracker, 3D mouse and stereo glasses. Every physical device available on the system corresponds to a *device node* in Metis, which encapsulates all system-dependent programming details of the particular devices. For example, the *Spaceball* node is defined as follows:

```
Spaceball {
    Quaternion    rotation;
    Vector        translation;
    Integer       button;
}
```

As in VRML2.0, nodes in a Metis scene can be shared, i.e. multiply-referenced, so that the scene data structure forms a directed acyclic graph rather than a simple hierarchy. By sharing a node, it is possible to save a memory usage and create complex structures more easily. However, multiple referencing creates some difficulties in maintaining transform constraints for shared nodes. Metis attempts to solve these problems and make node sharing as efficient as possible by using the constraint system to maintain transformation inheritance relationships. Fig. 3 is an example of a scene graph with multiple referencing. On the screen, three cubes will be shown, and two of those share the same material properties. The Metis API code for constructing such a scene graph can be found in the sample program in Appendix A.

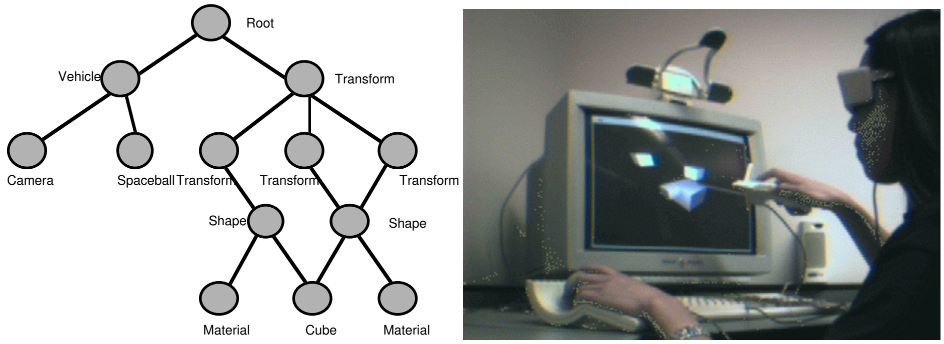

Fig. 3. A Scene DAG Viewed Schematically and by the User

3.2. Constraints

While standard graphics formats such as VRML 1.0 [Bell95] simply provide static scene rendering descriptions, Metis allows much of the interactive and dynamic behavior of the virtual environment to be defined in the scene graph itself. This is useful for implementing simulation-based VR applications where the interaction and simulation event response loops need to be decoupled. For example, consider a surgical simulation application. The user of such a program can control a virtual scalpel with a 3D input device, which directly controls the three-dimensional position and orientation of a virtual scalpel. Another tracker attached to the head changes the virtual point of view as the user looks at the scene from different positions. In the meantime, the elastic deformation of the tissue is simulated continuously using a differential equation solver.

Unlike other procedural systems such as MR [Shaw92], Metis allows much of this functionality to be implemented on the viewer side using a declarative method based on the constraint network. Constraint systems have long been used for maintaining relationships in 2D graphics systems (see [Sannella92] for a survey). In Metis, constraints are not used to fully program the application behavior, which is defined, using application specific techniques, in the simulation process, but to reduce the complexity of the communication between simulator and viewer. By using constraints, application programmers state declaratively a relation that is to be maintained between values of objects instantiated in the viewer, and are thus not required to write procedures to maintain the relation themselves on the simulator side. The number of degrees of freedom under direct simulator control and the bandwidth required for client-server communication are thus reduced, multiprocessing capabilities are exploited, and the computation load can be balanced between simulator and viewer processes. Given this particular use of constraints, we have decided to use one-way local propagation constraints, because they provide a good compromise between generality and efficiency [VanderZanden96]. Complex update schemes (such as multi-way constraints or physical simulations) can be implemented on the server side by changing at run time the set of active one-way constraints based on the system's state.

A constraint in Metis is unidirectional and it is analogous to a function, with one output and a number of inputs. Every type of constraint does a particular task very well. For example, one constraint calculates trigonometric functions, another one performs matrix transformation, etc. By linking certain types of constraints together and attaching their inputs and outputs to variables in the node subsystem, we can build a constraint network that establishes function dependency relationships between any two variables in the scene DAG. The Metis constraint solver, which resides in the Metis viewer, evaluates the constraint network and ensures that values of variables everywhere in the scene conform to the desired dependency relationships. In general, there may be many interrelated constraints in a given application. While the Metis application programmer must specify the constraints explicitly, it is left up to Metis constraint solver to maintain the constraint relationships and determine how and when to evaluate them.

3.3 Metis Constraint Types

To make it possible to program complex virtual environment behaviors using this declarative approach, a large collection of different types of constraints is necessary There are several categories of constraint types available in Metis that function very distinctively. A typical Metis application will build a constraint network containing constraints from the following categories.

- **Function Constraint**. A function constraint computes its single output value from its input variable values. For example, an Equality constraint simply duplicates its input value as the output. A Matrix Transformation constraint, a more complex example, takes its three input variables, for translation, scale and rotation, respectively, and outputs their product as a homogeneous matrix.

- **Channel Constraint.** When the simulator program running on the server needs to control some variables in the scene DAG, it does this through *channel* constraints. As the only way to affect the state of the Metis viewer from the simulator, a channel in Metis is actually the abstract of the socket by which the simulator transfers data to the Metis viewer via the TCP/IP network. Channels, along with variables in device nodes, are the only external data sources in the whole constraint network. Every variable in the network is either directly or indirectly dependent on them.

- **Notifier Constraint.** While channels offer the Metis viewer a way to get input from the simulator, notifiers report changes in specified variables back to the simulator. By linking variables to a notifier constraint, the Metis viewer can automatically notify the simulator that some device variable, or an arbitrary function of that variable, has changed.

While a static Metis scene is defined by a single directed acyclic graph, or scene DAG, the entire constraint network forms a second directional acyclic graph embedded within the scene DAG. The inputs of this constraint DAG consist of device variables and channels, while the outputs consists of notifiers and the node variables that determine the visual appearance of the scene.

Channel	Allows the simulator to update variable values.
FunctionNotifier	Notifies the simulator whenever the value of a variable is changed.
TimerNotifier	Notifies the simulator whenever a timer is alarmed.
Constant	Provides a constant value.
Equal	Makes values of two variables equal.
SineWave	Performs trigonometric computation.
Transform	Generates a homogeneous matrix from translation / rotation / scale.
MatInverter	Computes the inverse of a homogeneous matrix.
MatMultiplier	Computes the product of two homogeneous matrices.
MatRotator	Computes the product of a matrix and a quaternion.
MatTranslator	Computes the product of a matrix and a vector.

Table 1. Predefined Constraints in Metis

Instead of allowing users to define arbitrary constraint functions at run time, we have taken the alternative approach of coding a fixed set of primitive constraints and only allowing arbitrary constraint composition at run-time. The list of constraint functions currently available in Metis is given in Table 1. With this approach, constraints can be coded so that their instances require only a minimum amount of space at run-time, since for each constraint the type and parameters need to be chosen only from a limited set of possibilities. Efficient coding techniques, such as those introduced by Hudson and Smith for their ultra-lightweight constraints [Hudson96] can thus be employed. The reduced memory overhead and the reduced communication bandwidth of this solution outweigh in our opinion the reduction in expressiveness with respect to the use of general user-definable constraints.

3.4 Constraint Evaluation

In order to be efficient enough to support interaction, the Metis constraint solver needs to optimize the way it evaluates the constraint network. Our algorithm for doing this uses lazy evaluation with a provision for eager evaluation of notifier constraints. This can significantly reduce the number of constraint evaluations performed since, unless notifier constraints are involved, evaluation of data from channels and devices is not necessary until the moment of rendering.

The algorithm requires two boolean flags, *candelay* and *outofdate*, for each constraint instance. The *candelay* flag will be set to false if and only if the constraint is on a path to a notifier. In this case, immediate evaluation is required to give the simulator instant feedback. Otherwise, the algorithm simply marks the constraint out-of-date by setting the *outofdate* flag to true, indicating this constraint needs to be reevaluated when its value is needed. Thus, unless the *candelay* flag is false, the actual computation can be delayed until it is actually needed, i.e. the time of rendering. At that point, the constraint solver recursively traces back through the constraint network and performs the evaluation, until it finds a source (i.e. a channel or a device) or a constraint whose value is up-to-date. The evaluation phase can be further optimized to avoid unneeded computations when a constraint produces the same value as the one produced by previous evaluation, as in Hudson's incremental attribute evaluator [Hudson91][Hudson93].

3.5 Constraints vs. Scripts

While Metis uses a declarative constraint network to accommodate the need for building a dynamic virtual world, some systems deploy procedural methods instead. VRML 2.0, for example, uses events and script nodes for this purpose. The author of VRML 2.0's virtual world develops short programs in a Java-like syntax, called scripts, and puts them in special nodes called script nodes. A script is activated and then executed whenever the node receives an event, fired by a device or other scripts. According to the content stored in the event, as well as how the script itself is written, the script may fire events to other script nodes, which handle the event in a similar fashion.

While this procedural approach gives virtual world authors much flexibility to meet their various needs, it has several disadvantages. Compared to the constraint solution in Metis, a VRML 2.0 browser is more difficult to develop because a script language interpreter must be implemented. For the same reason, the virtual world authors are required to learn a programming language. Also, procedural solutions such as scripts make it much more difficult to apply efficient evaluation algorithms, such as lazy evaluation, in order to gain a performance improvement. Thus, constraint-based systems such as Metis are very likely to outperform their procedural counterparts.

4. Scene Examples

4.1 Spatial Input Devices

Three-dimensional devices are represented in Metis as device nodes, whose variable outputs can be linked to constraint inputs. Whenever a device node's variable changes, such as when a mouse pointer changes its position, dependent constraints in the constraint network get re-evaluated or marked out of date. In this way, we can efficiently control many aspects of a node's interactive behavior without directly handling events in the simulator.

A simple example involves direct manipulation of a geometric object using the spaceball. The spaceball and the geometric object are represented in Metis by a Spaceball node and a Transform node, respectively. By linking the Spaceball's rotation variable directly to the Transform's rotation variable using an Equality constraint, we can use a spaceball to control the orientation of all subnodes in the coordinate system represented by that Transform node. Similarly, when we link the Spaceball's translation and Transform's translation together, we can control both its orientation and its position. If we wish to control a translation variable using the orientation of the spaceball, all we need to do is to replace the Equality constraint mentioned above with a RotationToTranslation constraint, as in the following diagram:

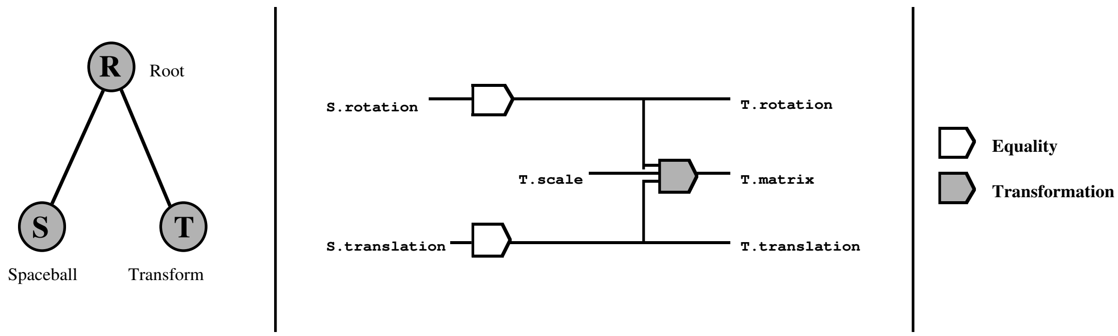

Fig. 4. Constraint Network for a Spaceball

4.2 Coordinate System Transformations

A more sophisticated application of a Metis constraint network is the maintenance of interrelated coordinate system transformations. Again, we will discuss the example of controlling a geometric object, represented by a Transform node, with a spaceball. In a fishtank virtual reality system, where the position of the head is tracked, the virtual camera position must be maintained with respect to a separate “vehicle” coordinate system, which represents the reference frame of the graphics screen itself in the virtual world. The position and orientation of the vehicle can be controlled in a manner analogous to driving a vehicle through the scene. Since the various 3D input devices reside in the same physical space as the screen (they are attached to the "vehicle"), their tracking data is normally interpreted as being in the vehicle coordinate system. The scene DAG in Fig. 5 shows how we represent this in Metis by placing a the Spaceball node as a child of the Transformation node representing the vehicle coordinate system. If we wish to directly manipulate Transform node T_3 using the spaceball, we need to convert the rotation and translation input values of the Spaceball, which are in vehicle coordinates, into the corresponding translation and rotation values of Transform node T_3 which are in T_3's parent coordinate system. We can solve this problem mathematically by converting each node's local coordinates into global coordinates and setting them to be equal, as expressed in the following matrix equation:

$$VS = T_1 T_2 T_3$$

where V, S, T_1, T_2, and T_3 are 4x4 homogeneous transformation matrices for the corresponding nodes. We can determine the transformation matrix, M, to convert the Spaceball's local rotation and translation values into Transform node T_3's local rotation and translation values,

$$MS = T_3$$

by rewriting the equation as follows:

$$T_2^{-1} T_1^{-1} VS = T_3$$
$$M = T_2^{-1} T_1^{-1} V$$

We can use the Metis constraint system to dynamically calculate the value of M and then use it to transform the separate rotation and translation components.

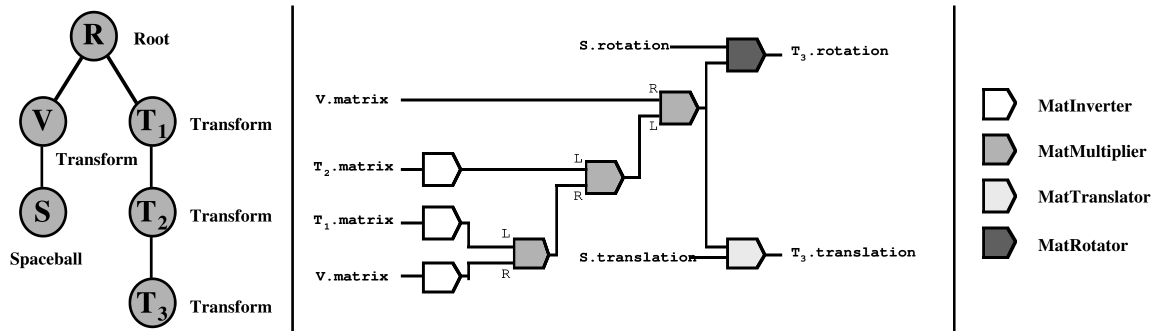

Fig. 5. Constraint Network for Maintaining Coordinate System Relationships

There are at least four types of constraints that are required to accomplish this. Among them, we need a MatInverter constraint, which gives the inverse of the input matrix; a MatMultiplier, which calculates the product of two matrices, a MatRotator and a MatTranslator, which can transform a quaternion or a vector respectively by a matrix. Fig. 5 shows how these constraints can be connected to form a constraint path from the Spaceball to the Transform node that automatically performs this matrix calculation in the Metis viewer without any processing by the simulator program. Furthermore, since it uses the Metis constraint solver with lazy evaluation, these calculations will be performed efficiently.

4.3 Three-Dimensional Widgets

In most 2D graphical software, in order to perform 2D operations, such as creating a polygon, moving it around, changing its color, etc., the user manipulates a 2D device, usually a mouse, to control a 2D cursor, or 2D widget. Often we would like 3D software to work in a similar fashion, that is, directly manipulating 3D widgets using real 3D devices, in order to make the user interface more intuitive. In Metis, this can be done by dynamically constructing a constraint network.

Suppose we want to control a three-dimensional cursor with a spaceball. When the cursor is pointing to an object in the scene, we push and hold down the button to drag it around. After moving it to the desired position, we release the button. A simple example of how such a three-dimensional cursor widget can be implemented in Metis is shown in Figures 6 and 7, where C and T are the Transform nodes for the cursor and object respectively. Fig. 6 shows the constraint network before we click the button. We set C.translation equal to the S.translation output, and we link two notifiers, one for the spaceball button and one for the position of the cursor, to let the simulator know the current state of the cursor.

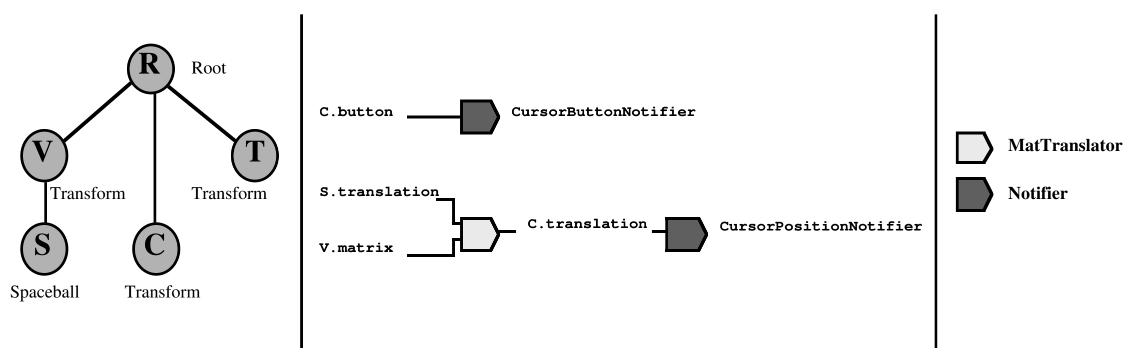

Fig. 6. Constraint Network for a 3D widget: Phase 1

Whenever the button is clicked, the simulator will be notified by the CursorButtonNotifier. It will then check whether any object is selected by examining the position of the cursor, which is obtained through CursorPositionNotifier. If an object, say T, is really selected, the simulator will send requests to the viewer to change the constraint system as in Fig. 7, where T.translation is forced to be equal to C.translation. Thus, the object represented as T will be dragged by the cursor, which in turn is controlled by the spaceball.

Similarly, when the button is released, the simulator is notified again and it will unlink T.translation and restore the constraint network to the state depicted in Fig. 6.

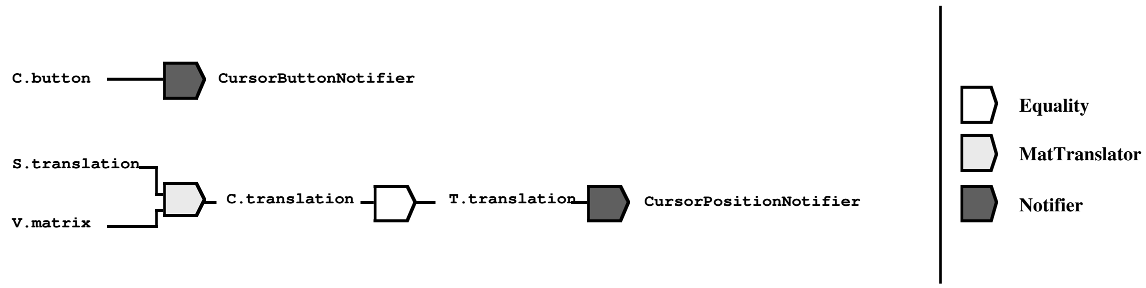

Fig. 7. Constraint Network for a 3D widget: Phase 2

5. Implementation

Metis is being developed on an SGI Onyx Reality 2 Workstation. The current version of the viewer and server API has approximately 17,000 lines of code. Several small demonstration applications such as those described in Section 4 have already been implemented. Currently Metis uses a Spacetec Spaceball and Logitek trackers for 3D input and head-tracking and CrystalEyes shutter glasses for stereo viewing. The viewer front-end is implemented using C++ and the OpenGL graphical library. However, it could be easily ported to other hardware and graphical libraries, such as PHIGS. The simulator API uses C++, but other object-oriented language bindings, e.g. Java and Eiffel, are also planned.

6. Summary and Future Work

The Metis toolkit defines a client/server architecture for building virtual reality applications in which an application programming interface on the server side communicates via a network with a standalone viewer program on the client side that handles all immersive display and interactivity. Network bandwidth and interaction latency are minimized, by use of constraint network on the viewer side that declaratively defines much of dynamic and interactive behavior of the application, freeing the server-side to devote its resources to computationally intensive simulations. We believe that this constraint network can work very effectively for a variety of virtual reality applications, and that the client-server and object-oriented architecture of Metis gives VR application programmers the ability to construct high-performance VR applications with computationally intensive simulation. The current implementation of Metis has demonstrated the usefulness of this approach. Future enhancements planned for Metis include the addition of constant-time rendering and a time-based constraint system. We also plan to develop more demonstration programs and to use it to implement a physically-based animation system.

Acknowledgments

We would like to thank the reviewers for their many helpful suggestions. This work was supported by National Science Foundation Interactive Systems grant number IRI-9503093.

References

[Bell95] Bell G., Parisi A., Pesce M. (1995) *The Virtual Reality Modeling Language Specification, Version 1.0.*

[Carlsson93] Carlsson C, Hagsand O (1993) DIVE - A Platform for Multi-User Virtual Environments. Computers & Graphics 17(6).

[Elliot94] Elliot C, Schechter G, Yeung R, Abi-Ezzi S (1994) TBAG: A High-Level Framework for Interactive, Animated 3D Graphics Applications. Proc. SIGGRAPH: 421-434.

[Gobbetti93] Gobbetti E, Balaguer JF (1993) VB2: A Framework for Interaction in Synthetic Worlds. Proc. UIST: 167-178.

[Hudson91] Hudson S (1991) Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. ACM Transactions on Programming Languages and Systems 13(3): 315-341.

[Hudson93] Hudson S (1993) A System for Efficient and Flexible One-Way Constraint Evaluation in C++. Technical Report GVU-93-15, Graphics, Visualization, and Usability Center, Georgia Tech.

[Hudson96] Hudson S, Smith I (1996) Ultra-Lightweight Constraints. Proc. UIST: 147-155.

[Robertson89] Robertson GGm Mackinlay JD, Card SK (1989) The Cognitive Coprocessor Architecture for Interactive User Interfaces. Proc. UIST: 10-18.

[SGI96] Silicon Graphics, Inc. (1996) *The Virtual Reality Modeling Language Specification, Version 2.0.*

[Shaw92] Shaw C, Liang J, Green M, Sun Y (1992) The Decoupled Simulation Model for Virtual Reality Systems. Proc SIGCHI: 321-328.

[Torguet95] Torguet P, Caubet R (1995) VIPER - A Virtual Reality Application Design Platform. Proc. EG Workshop on Virtual Environments.

[VanderZanden96] Vander Zanden BT, Venckus SA (1996) An Empirical Study of Constraint Usage in Graphical Applications. Proc. User Interface Software Technology: 137-146.

[West92] West AJ, Howard TLJ, Hubbold RJ, Murta AD, Snowdnon D, Butler DA (1992) AVIARY - A Generic Virtual Reality Interface for Real Applications. Proc. Virtual Reality Systems.

Appendix A: A Sample Metis Simulator Program

```
MeTransform      root;
MeTransform      vehicle;
MeTransform      cursor;
MeTransform      object1, object2;
MeCamera         camera;
MeShape          shape;
MeBox            box;
MeAppearance     app;
MeMaterial       mat;
MeShape          curshape;
MeSphere         curpointer;
MeSpaceball      spaceball;
MeMatTranslator  matxlate;

ButtonNotifier   curbutnotifier;
CursorNotifier   curposnotifier;



root.children <<= vehicle.children <<= camera;
vehicle.children <<= spaceball;
root.children <<= cursor.children <<= curshape.geometry <<= curpointer;
root.children <<= object1;
root.children <<= object2;
object1.children <<= shape;
object2.children <<= shape;
shape.geometry <<= box;
shape.appearance <<= app.material <<= mat;


mat.specular  = MeColor (0.7, 0.5, 0.5);
mat.diffuse   = MeColor (0.2, 0.2, 0.2);
mat.shininess = 0.9;


vehicle.rotation = MeQuaternion (MeVector (0,0,1), -0.75*PI) *
                   MeQuaternion (MeVector (1,0,0), -racos(1.0/rsqrt(3.0)));
vehicle.translation = MeVector (6, 6, 6);


matxlate.matrix <<= vehicle.matrix;
matxlate.translation <<= spaceball.translation;
curbutnotifier.source <<= spaceball.button;
curposnotifier.source <<= cursor.translation <<= matxlate;
curpointer.radius = 0.3f;

setRoot (root);

cout << "Scene creation done." << endl;
cout << endl;
```