Adapting VRML For Free-form Immersed Manipulation

Randy Stiles Sandeep Tewari
Mihir Mehta Laurie McCarthy

Lockheed Martin Advanced Technology Center*
http://vet.parl.com/~vet/

Abstract

We have extended the VRML standard to allow free-form manipulation of objects while immersed. Our driving goal was immersed training for equipment operations and maintenance, and to this end we developed a sensor that allows 6DOF manipulation, a cooperating sensor that allows snapping objects into place as part of an assembly, and a two-handed manipulation approach for these sensors.

CR Categories and Subject Descriptors: I.3.6 [Computer Graphics]: Methodology and Techniques, Interaction techniques, standards. I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism, virtual reality; H.5.2 [Information Interfaces and Presentation]: User Interfaces - Interaction Styles; I.3.4 [Computer Graphics]: Graphics Utilities, Virtual device interfaces.

1. INTRODUCTION

This paper describes a suggested extension to the VRML standard that allows for free-form manipulation of objects while immersed in a virtual environment. We discuss our implementation of TransformSensors and SnapSensors, for both single and two-handed manipulation of objects. Free-form manipulation of objects is a necessary prerequisite to our work in applying virtual environments to training. We are supporting operations and maintenance training on CAD-derived shipboard equipment, where it is a common task to pull objects out, assemble them, and snap or plug them into assemblies.

The set of sensor nodes already in VRML are useful for many cases - we find the CylinderSensor very useful for manipulating valves, doors, dials, selector switches, etc. while immersed [11]. But no existing VRML sensor couples both position and orientation output, a necessary precondition for picking an object up and positioning it somewhere else. Similarly, there are no sensors or other mechanisms in VRML that allow snapping other sets of sensor outputs to position and orientation. It is probably possible to get a snapping effect using sensor values routed to a script node and then out to transforms, but given the necessity of explicitly routing items, there is no mechanism to state that the snapping action works with all other sensors, or sensors of a given type.

The quality of interaction is very important when using a virtual environment for training in equipment operations and maintenance. Tasks to operate equipment always involve manipulating controls, and maintenance almost always involves assembling, disassembling, and replacing equipment. Researchers in the computer graphics field, such as Mark Mine, Fred Brooks and Carlo Sequin, recently pointed out that *virtual environments lack a unifying framework for interaction*, and elaborate that *knowledge on how to manipulate objects or controls cannot be "stored in the world"* [5]. By adapting VRML 2.0 for immersive use, we have realized the start of a unifying framework for interaction, that allows storing knowledge of how to manipulate objects and controls "in the world".

The forms of interaction for single and two-handed immersed manipulation of the TransformSensor and SnapSensor have been implemented in our Vista software. Vista is a Performer-based [9] immersive display software that works with other software components to accomplish immersive training for one or more students, and its VRML capability has been developed for the Virtual Environments for Training effort. Vista is a software component of our virtual environment training system, a simulation framework for training called the Training Studio [10, 3]. The other primary components are Vivids, an intelligent tutoring and simulation system [7], and Steve, a pedagogical agent that provides task training as a mentor or missing member of a team [8].

2. INTERACTION FOR TRAINING

To support equipment maintenance and operations training, it is necessary to let people tear apart objects and put them back together, to replace parts or open them to inspect them. For realistic training and effective evaluation of skills, a level of freedom during performance is often required; i.e., a single "correct" path cannot be pre-defined or, therefore, pre-authored. Multiple solution paths can exist for reasons relating to both procedures and the object itself. Procedural differences are common to real world behavior and can be due to a trainee's reordering of sub-tasks that are independent, and not strictly hierarchical. Issues involving free manipulation also arise when authoring the behaviors of models used for equipment maintenance and operations training. Often assembly of equipment involves several objects that may or may not be functionally interchangeable, but are physically similar.

The primary techniques we selected as critical to support real-world manipulation during training include 6DOF manipulation, two-handed manipulations, and snap locations

*O/H1-43, B255, 3251 Hanover St., Palo Alto, CA 94304
{stiles,tewari,mehta,mccarthy}@ict.atc.lmco.com

for object placement. Allowing for 6DOF manipulation is particularly important for providing realistic interactions within the environment. More restrictive manipulations would probably suffice for operations at consoles or panels, but maintenance often requires more complex interactions. Support for combined translation and rotation of objects is essential for tasks such as part replacement or component assembly.

Provision for two-handed manipulations is important if the experience is to extend to real world interactions. For example, to remove a large panel or other piece of equipment, the object must sometimes be pulled straight out or at an angle, which requires two hands on a single object. For this type of manipulation, each hand plays a role in positioning and orienting the object.

Two-handed manipulation is also needed for simultaneous manipulation of multiple objects. In the example of disassembling an oil pump (Figures 1,3), three rings are removed from a shaft previously removed from the pump. The shaft must be held with one hand while the other removes the rings. Manipulation techniques must be available for both hands, each controlling a different object.

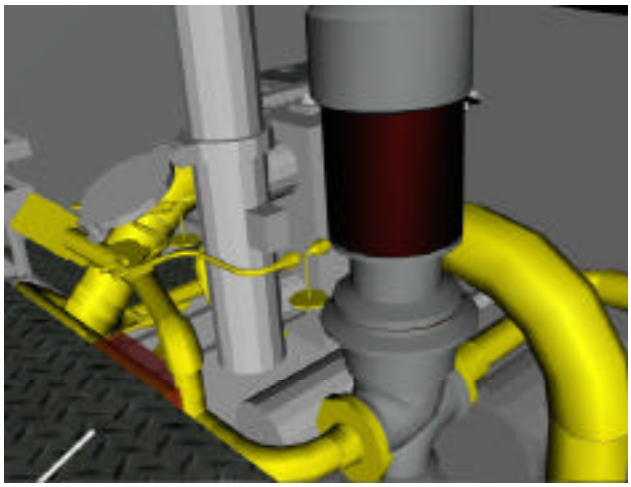

Figure 1. Fuel Oil Pump and CAD Piping

Another issue involves connecting objects within the immersed environment. This is essential for equipment operations since the assembly of components requires joining objects, for example, screwing in a bolt, joining two pipes, or inserting alignment screws. These are tasks that require precise object alignment, which is difficult in the immersed environment. Requiring such precision in an environment that lacks tactile feedback might affect the training process since the focus shifts from the training content to the training environment. Instead, we simplify matters so that if the trainee places the appropriate type of object in a location that is authored to accept, the object will automatically snap into place.

3. TRANSFORMSENSOR

3.1 Description

A TransformSensor (see Table 1) is used to designate an object as being moveable in all six degrees of freedom; i.e., by changing translation and rotation concurrently, such as is possible with a 6DOF position sensor (Ascension, Logitech, Polhemus, etc.). The changed rotation and translation can be routed to a Transform node, Script node, etc.

```
PROTO TransformSensor [
   exposedField SFBool enabled  TRUE
   eventOut SFBool isActive
   exposedField SFBool autoOffset  TRUE
   exposedField MFString snapTypes [  ]
   exposedField SFVec3f translation
     0 0 0
   exposedField SFRotation rotation
     0 0 1 0
   exposedField SFRotation rotationOffset
     0 0 1  0
   exposedField SFVec3f translationOffset
     0 0 0
   ]
{}
```

Table 1. TransformSensor Fields

The TransformSensor, when active, simultaneously outputs events for both translation and rotation. For immersed use, a person selects an object with a manipulator (tied to a position sensor) and, if associated with a TransformSensor in the scene graph hierarchy above, the TransformSensor outputs values of the manipulator relative to the selected object. If the TransformSensor output is sent to the object's Transform node, the object will move in a corresponding manner. This is immersed, direct manipulation. The effect achieved is as if one were holding the object in one's hand.

The TransformSensor belongs in the set of VRML Geometric Sensors (Plane, Cylinder and Sphere). Therefore the rules for finding TransformSensor(s) attached to an object are the same as for other Geometric Sensors associated with a selected object.

When the TransformSensor is enabled it tracks user input and sends output events. Unlike other Geometric Sensors, it keeps track of user input whenever it is selected, but does not necessarily send output events. The TransformSensor keeps track of user input when it is used in conjunction with a SnapSensor. If it is snapped at the time, it tracks user input until it is unsnapped, and then routes any further results from user input out as output events. Upon activation of the pointing device an *isActive* TRUE eventOut is generated. Upon deactivation of the pointing device an *isActive* FALSE eventOut is generated.

For the common case where the VRML scene is displayed on a flat-screen with 2D mouse-based flat interaction, the TransformSensor could be manipulated by using a mechanism like the universal manipulator in Inventor, where the planes of a box surrounding the object allow you to move the object in a given plane, and the axis displayed with the object allows you to rotate the object. At most, a person can manipulate only 2 degrees of freedom at a time in this manner, but the functionality is complete to accomplish the same end results as the immersed case.

The fields *rotationOffset* and *translationOffset* are used to maintain history so that translations and rotations are relative to the initial values. When an *isActive* FALSE eventOut is generated, the current translation and rotation values are stored as the offsets so that these can be used when the object is picked next time. Relative translations and rotations can be disabled by setting the *autoOffset* flag to FALSE, making all motion absolute. The storing of rotation and translation offsets is a little different when a TransformSensor is used in

conjunction with a SnapSensor. When *autoOffset* is TRUE and the TransformSensor becomes inactive while within range of a SnapSensor it can snap to (i.e., it has snapped), then the position and orientation of the SnapSensor is used as the offset for future interaction with the TransformSensor. If the TransformSensor becomes inactive while out of range of all SnapSensor(s) it can snap to then its current position and orientation is used as the offset for future interaction.

During immersion, rotation values output by the TransformSensor are computed by rotating the initial orientation of the pointing device (when the TransformSensor became active) into the current orientation. To prevent the object from jumping suddenly to the pointing device's orientation a correction matrix is applied that takes the initial orientation of the pointing device into that of the object (to which the TransformSensor is attached). In flat-screen mode, rotation values are output exactly the way they are done for a SphereSensor.

Translation values that are output by the TransformSensor when immersed are the difference between the position of the pointing device when the TransformSensor became active and the current position of the pointing device. In flat-screen mode, translation values are output exactly the way they are done for a PlaneSensor.

When the TransformSensor receives a *set_translation* event it sends this value out as a *translation_changed* event. When it receives a *set_rotation* value it sends this value out as a *rotation_changed* event.

The *snapTypes* field is used to store the named types of SnapSensors that a TransformSensor can snap to. *snapTypes* are user-defined strings. This is described in the section for the SnapSensor.

4. SNAPSENSOR

4.1 Description

The SnapSensor is used to designate certain locations in the scene graph as snap locations (see Table 2). By using a SnapSensor, the content author can specify that a given type of object will fit into this location. The SnapSensor holds the position and orientation of the object when it snaps. This is useful for designating a location where a nut will fit into a bolt, a shaft will fit into a casing, etc. Most importantly, snapping allows the author to overcome imprecision in movements that is common in immersed systems that have no physical feedback. Our technique for snapping is based on range checks.

```
PROTO SnapSensor [
   exposedField  SFString snapType
     "defaultSnap"
   exposedField SFFloat snapWidth 0.05
   exposedField SFFloat snapHeight 0.10
   exposedField  SFFloat snapAngle 0
   exposedField SFVec3f center 0 0 0
   exposedField SFRotation orientation
      0 0 1 3.14
   exposedField SFBool enabled  TRUE
   eventOut  SFBool isActive
   ]
{}
```

Table 2. SnapSensor Fields

When an object comes within range of a SnapSensor, an isActive TRUE eventOut is sent. If the object is already within the SnapSensor's range the object is not moved until the Geometric Sensor's (used to move the object) translation and rotation values are outside the SnapSensor's range. The effect achieved is that the object will pop out from the SnapSensor's position and orientation to the pointing device's position and orientation. From that point the object can be seen to follow the moving sensors values. When an object is moved out of range of a SnapSensor it sends an isActive FALSE eventOut. While a SnapSensor is active, i.e., if an object has snapped to it, other objects cannot snap to this SnapSensor. Allowing multiple objects to snap at the same time onto a SnapSensor can lead to objects getting lost. Allowing one object to snap at a time onto a SnapSensor is more practical and useful.

Typically, Geometric Sensors (Transform, Plane, Cylinder, Sphere) are used to manipulate (translate/rotate) objects. The SnapSensor is not a Geometric Sensor but its behavior is defined as a result of interaction with Geometric Sensors. SnapSensors are similar to Proximity Sensors. Proximity Sensors continuously check if the user's viewpoint is within range. SnapSensors selectively check to see if Geometric Sensors (Plane, Transform, Sphere, Cylinder) are within range. Instead of making range checks for every frame, the SnapSensor makes range checks whenever a Geometric Sensor that can snap to it becomes active. This means that range checks are done only against those Geometric Sensor(s) that are actively being used, and can snap to a SnapSensor.

The *snapType* is used to determine which Geometric Sensors can snap onto a SnapSensor. This is a user-defined string. When a TransformSensor becomes active it matches its *snapTypes* with the *snapType* of the SnapSensor to determine if it can snap to this SnapSensor.

Since the SnapSensor has both position and orientation, a cylinder is the most natural thing to use for range checks. Also, using a cylinder is useful if long column-like objects have to be snapped into place inside a pipe, casing, or other cylinder-like area. A cylindrical range check allows for a tighter fit than possible with a sphere for most cases involving equipment.

The *orientation* defines the orientation of the SnapSensor and consists of an axis and an angle (twist or roll) about this axis. The *snapWidth* and *snapHeight* define the radius and the half-length of a cylinder which is used for range checks. Henceforth we will refer to this cylinder as the rangeCylinder (Figure 2 shows a visual display of a rangeCylinder). The axis of the rangeCylinder is defined by the axis in the orientation field of the SnapSensor. The angle in the orientation field of the SnapSensor is ignored when defining the orientation of the rangeCylinder. The center field is the center of the rangeCylinder. If the difference between the orientation of the object and the orientation of the SnapSensor is less than the *snapAngle* then the object can snap. The *snapAngle* is very useful since it reproduces behavior where an object must be twisted to lock it into place.

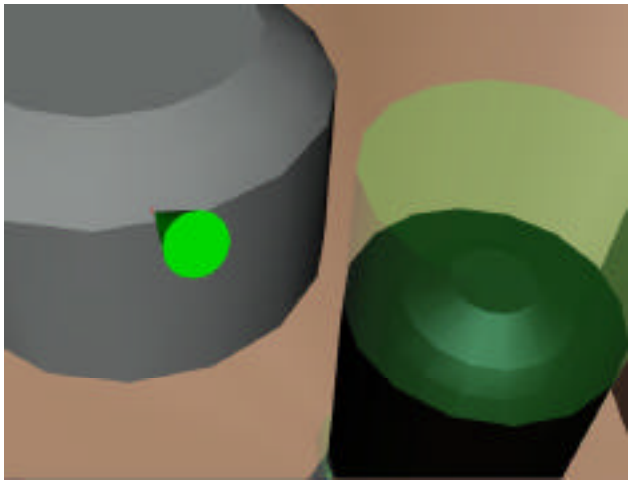

Figure 2. Green Range Cylinder Shown For SnapSensor Atop Pump

Determining when a Geometric Sensor is within snapping range of a SnapSensor should be done in the following way:

- Check if the center of the Geometric Sensor is within the rangeCylinder of the SnapSensor.
- If so, find a orientation (axis, angle) that will take the Geometric Sensor's current orientation into the SnapSensor's orientation. If the roll or twist in this orientation is less than or equal to the snapAngle of the SnapSensor then the Geometric Sensor is within range. If the value of the snapAngle is π radians then there is no restriction on the orientation of the object before it can snap. If the orientation of the snapAngle is 0 radians, then the object must be perfectly aligned before it can be snapped to the SnapSensor.

4.2 Usage With TransformSensor

We have implemented the interaction between the SnapSensor and the TransformSensor and describe this interaction later in this section. Other Geometric Sensors (Plane, Cylinder and Sphere) can be very easily extended to incorporate this behavior.

When a TransformSensor becomes active, it gets all the SnapSensors that it could possibly snap to via the *snapTypes* field, hence referred to as the SnapSensors-of-interest list. While the TransformSensor is active the SnapSensors-of-interest list is used to make range checks against it. If a SnapSensor comes within range of the TransfomSensor it invokes the SnapIn behavior. The SnapIn behavior sets the position and orientation of the TransformSensor to be that of the SnapSensor and disables the TransformSensor. If there is more than one SnapSensor within the range of a TransformSensor then the behavior is not defined. When a snapped TransformSensor is moved out of range of a SnapSensor, the SnapOut behavior is invoked (see Figure 3). SnapOut behavior enables the TransformSensor so that its position and orientation values start taking effect.

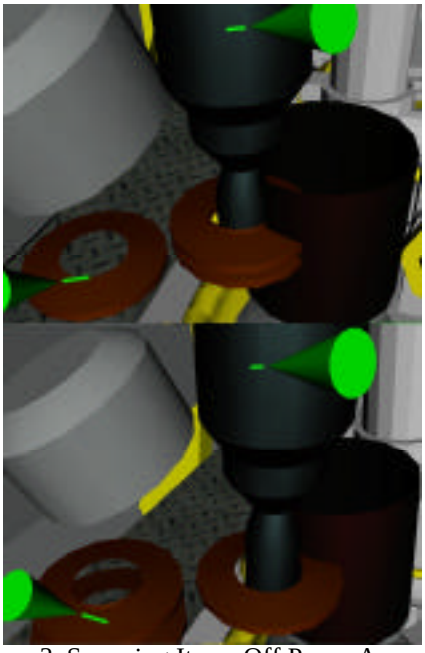

Figure 3. Snapping Items Off Pump Assembly

Snap In behavior:

- SnapSensor sends out *isActive* TRUE eventOut.
- SnapSensor sends out its position and orientation along type-matched, "dynamic routes" to the TransformSensor.
- SnapSensor disables the TransformSensor.
- TransformSensor receives the new position and orientation and sends it along routes possibly to a Transform node, thereby changing the position and orientation of the object(s) under this transform.
- TransformSensor is disabled. It keeps track of its translation and rotation but does not send these along routes.

Snap Out behavior:

- SnapSensor enables the TransformSensor that has snapped to it.
- TransformSensor starts sending out its translation and rotation along routes.
- SnapSensor sends out an *isActive* FALSE eventOut.

The use of type-matched, "dynamic routes" (or instantaneous routes) allows flexible re-use of objects and saves the overhead of maintaining specific routes between a given SnapSensor and TransformSensor. Such routes are transparent to the user and the content author does not have to be concerned about creating these routes, only the snapTypes need to be correctly specified.

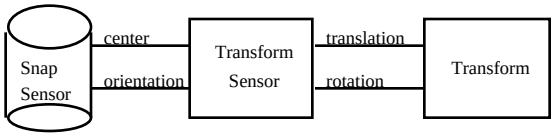

Figure 4. Interaction between SnapSensor and TransformSensor

4.3 Anomalies

A TransformSensor cannot snap to a SnapSensor that is a child of the Transform that the TransformSensor is modifying. This is because, depending on the center specified for the SnapSensor, the TransformSensor will either be always snapped or will never snap to the SnapSensor. Figure 5 illustrates this anomaly. TransformSensor TS1 cannot be snapped to SnapSensor SS1. Such anomalies can be avoided by adding a check which eliminates such anomalies when the TransformSensor builds the list of SnapSensor(s) it can snap to.

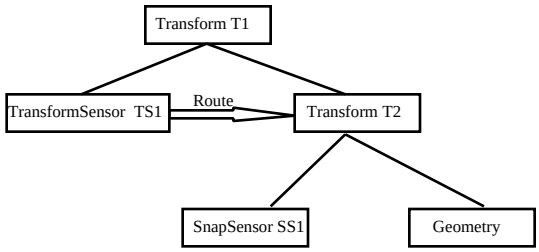

Figure 5. Anomaly Associated With SnapSensors

5. TWO-HANDED MANIPULATION

Two-handed manipulation of VRML sensors is useful in the case where a person must manipulate two objects at the same time, such as two throttles shown below in Figure 6 (CylinderSensors) or in the case where one Sensor must be manipulated with fine control.

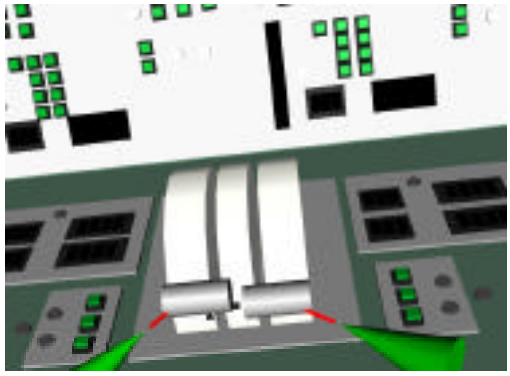

Figure 6. Simultaneous Two-handed Use of Separate CylinderSensors

Our two-handed implementation of VRML sensors uses FakeSpace Pinch Gloves. The pinch gloves and the sensors attached to it are represented in the virtual environment by a cone and segment aligned to the palm of the hand. We have assigned simple "pinches" (finger touches) to represent the most commonly used interaction modalities. Table 3 describes some of these operations.

Finger Combination	Resulting Action
Index and Thumb Pinch	Picking, Object Motion
Middle and Thumb Pinch	Forward View Motion
Ring and Thumb Pinch	Backward View Motion

Table 3. Pinch Mappings

The graphical representation makes it easy to determine intersections with the objects. A number of one-handed and two-handed interactions are possible in the virtual environment as described in [2, 4]. Some of the interaction functionality are application-specific and some can be directly authored into a scene in VRML. In [11] we described the one-handed interaction with standard VRML sensors in an immersed virtual environment. Now, we describe free-form two-handed manipulation of the TransformSensor (see Figure 7). The first hand to touch an object (H_1) is used to position the object, and the second hand is used to orient the object (H_{2a}, H_{2b}). This accommodates both left-handed and right-handed people.

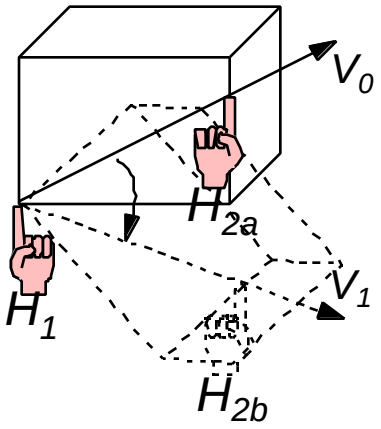

Figure 7. Two-handed Manipulation

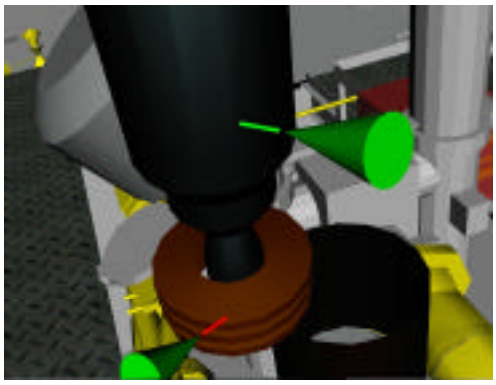

Figure 8. Two-hand Positioning of Single Sensor

1. The first hand that does the pick operation determines the object to be manipulated (Roll from the first hand is ignored).
2. Translation is determined by the first hand to select the object.
3. The second hand intersects the same object and the segment between first and second intersection points on the object defines a vector, V_0. Its direction is from the first intersection point to the second.
4. Either one or both hands are moved and the segment between the two endpoints of the two hands determines Vector V_1. No further intersections with the object are required.
5. The object is rotated so that initial vector is aligned to the new vector as the two hands change locations.

6. RELATED AND FUTURE WORK

To date we have not come across any other work towards adapting VRML 2.0 for free-form immersed manipulation and interactive assembly. We are aware of virtual environment systems, such as those by MultiGen, Deneb Robotics, and Division Inc., that support snapping objects into place while immersed. Snapping is a common and effective way to deal with imprecision in movement and lack of physical feedback in a virtual environment.

The work by Cutler [2] does provide insight into issues related to two-handed direct manipulation. They point out that most interesting tasks are coordinated and asymmetric: both hands perform different sub-tasks in a synergistic way to get a complex task done. We plan to create such tasks via the extensions to VRML 2.0 (TransformSensor and SnapSensor) that we have described in this paper.

The see-through interface proposed by Bier [1] allows for the use of both hands to select objects and perform operations on them simultaneously. The non-dominant hand is used for trivial activities like positioning and the dominant hand is used for triggering operations. This can save time and reduce cognitive load, because the user can combine more steps into a single mental chunk. Our technique for two-handed manipulation accommodates both left-handed and right-handed people.

In our implementation it is possible to pick on two different pieces of geometry (which could be parts of a hierarchy) and use both hands independently to manipulate these objects. So it is possible to perform unimanual, bimanual symmetric and bimanual asymmetric tasks [4].

In this paper we have outlined our approach to manipulate a single sensor (object) with both hands. Munlin [6] points out that it is difficult to accurately position objects in a 3D environment using 3D input devices. Positioning of objects in a 3D environment using 3D input devices is greatly simplified by providing feedback (via SnapSensors, visual cues etc.). Munlin [6] also mentions a rich set of constraints such as screwfit, gear contact, and rack & pinion contact that need to be implemented in order to be able to perform constraint-based assembly modeling. We plan to extend VRML 2.0 to incorporate complex constraints which will prove useful for performing interactive assembly while immersed.

7. SUMMARY

This paper outlines our approach for free-form manipulation of VRML objects while immersed. Extending VRML, it contributes to a unifying framework for specifying immersed interaction in a virtual environment. Along with the existing sensors in VRML, our added TransformSensor and SnapSensor help to provide knowledge on how to manipulate objects or controls that are stored in the world using the VRML format. We have worked with the goal of equipment operations and maintenance training in mind, but the issues addressed to accomplish this goal are likely common to most immersed uses of VRML.

ACKNOWLEDGEMENTS

The work described in this paper was funded by the Office of Naval Research, as part of the Virtual Environments for Training program, contract no. N00014-95-C-0179 and is derived from previous work funded by USAF Armstrong Labs contract no. F41624-93-C-5000. Engine Room models of the ship MER2 area shown in figures are derived from polygonal models provided by Ingalls Shipbuilding.

References

[1] Bier, E., Stone, M., Fishkin, K., Buxton, W., Baudel, T., A Taxonomy Of See-through Tools, *Proceedings of CHI'94*, 358-364.

[2] Cutler, L. D., Frohlich, B., Hanrahan, P., Two-Handed Direct Manipulation On The Responsive Workbench, *Symposium on Interactive 3D Graphics*, 1997.

[3] Durlach, N. I., A. S. Mavor, ed., *Virtual Reality: Scientific and Technological Challenges*, pp. 261-267, 292-296. Copyright 1995, National Academy Press.

[4] Guiard, Yves. Symmetric Division Of Labor In Human Skilled bimanual action: the kinematic chain as a model, *The Journal of Motor Behaviour*, 19(4):486-517, 1987.

[5] Mine, Mark R., Brooks, Frederick P., Sequin, Carlo H. Moving Objects In Space: Exploiting Proprioception In Virtual Environment Interaction, *Proceedings of SIGGRAPH '97,* Los Angeles, CA, Aug. 1997.

[6] Munlin Mud-Armeen. Interactive Assembly Modeling within a Virtual Environment, In *International Conference on Robotics Vision and Parallel Processing for Industrial Automation (ROVPIA '96)*, Ipoh, Perak, Malaysia, Nov. 28-30, 1996.

[7] Munro, A., M.C. Johnson, Q.A. Pizzini, D.S. Surmon, and J.L. Wogulis. A Tool for Building Simulation-Based Learning Environments, In *Simulation-Based Learning Technology Workshop Proceedings, ITS'96*, Montreal, Québec, Canada, June 1996.
http://btl.usc.edu/rides/shortPapers/bldsim.html

[8] Rickel, J., W. L. Johnson. Integrating Pedagogical Capabilities in a Virtual Environment Agent, *Proceedings of First International Conference on Autonomous Agents*, Feb. 1997.
http://www.isi.edu/isd/VET/agents97.ps

[9] Rohlf, J. & J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-time 3D Graphics, *Proceedings of SIGGRAPH '94*, Orlando FL, Aug.1994.

[10] Stiles, R., McCarthy, L. Johnson., W. L., Munro, A., Pizzini, Q. Virtual Environments for Shipboard Training, In *Proceedings of Intelligent Ships Symposium II,* The American Society of Naval Engineers, Philadelphia, PA, Nov. 1996.
http://vet.parl.com/~vet/iships/iships_ToC.html

[11] Stiles, R., Tewari, S., and Mehta, M.. Adapting VRML 2.0 for Immersive Use, In *Proceedings of VRML '97 Second Symposium on the Virtual Reality Modeling language*, Monterey, CA, Feb. 1997.