# Implementation of mechanisms for concurrent 3D design and visualisation

David Sánchez Crespillo, Antonio Francisco Bennasar Obrador, Ricardo Galli Granada, Yuhua Luo
University of Balearic Islands

***Abstract*-- To perform 3D editing in a cooperative environment presents problems that do not exist in other environments. To maintain the consistency among all the replicated instances of the scene being edited is the key problem.**

**In this paper we present the mechanisms to solve these problems such as collaborative operations on multiple scenes, cooperative copy and paste operations, and the ability of reversing actions (*undo*) in a cooperative editing environment.**

**A brief introduction to a cooperative 3D editing and visualisation tool is presented. The protocol to ensure its memory consistency is discussed. The data structures, as well as the techniques used to manipulate the data to realise the mechanisms are described.**

***Index terms*– CSCW, cooperative 3D editing, cooperative visualisation, VRML.**

## 1. INTRODUCTION

The development of a networked, multi-user cooperative 3D editor is a great challenge due to the need of consistency control in a multi-user environment.

The management of multiple scenes, copy and paste and "undo" are clear examples of operations that need to be redefined and redesigned for cooperative systems.

The management of multiple 3D scenes allows the user to visualise and work with different designs at a time. This is especially useful when using the 3D editing tool to combine designs from different specialities.

Clipboard operations can support rapid prototyping and editing by allowing the user to copy existing entities, paste them to another position of the document and change its properties.

The undo is a natural operation for any editing tool that gives the user the chance to correct errors and test some operations.

These operations are natural in a single user environment in which only one user is able to change the attributes at a given time. However, when coming to an on-line cooperative, multi-user, networked environment, they become a lot more complicated. The reason is that there are more than one user acting in the shared virtual environment, the scope of their operations must be clearly limited to avoid inconsistency in the distributed database.

In this paper, we present the implementation of these three mechanisms in a cooperative concurrent 3D design and visualisation tool. The tool is the central part of a multi-site, multi-user cooperative 3D design system for architecture connected by long distance communication networks. These mechanisms have been implemented using an underlying application protocol, called the Mu3D [Gall00], and techniques of mutual exclusion.

Section 2 of the paper briefly describes the structure of the 3D cooperative design and visualisation tool, the Mu3D protocol, and the mutual exclusion policies to implement the concurrency control.

Sections 3 to 5 introduce the implemented mechanisms: multiple scene management, cooperative clipboards and cooperative undo. Finally, Section 6 gives the conclusions and a description of future works.

The system described in this paper has been developed using the programming language C++, along with the Open Inventor 3D toolkit, on both SGI and Windows NT platforms.

This work has been funded by the ESPRIT 26287 project, M3D and the Spanish national funding CICYT, TIC-98-1530-CE.

## 2. THE CONCURRENT 3D DESIGN AND VISUALISATION TOOL

The concurrent 3D design and visualisation tool is depicted in Figure 1. The users can open different scenes from files or databases and edit them in separate windows. The tool allows multiple users editing the same scene or the same set of scenes simultaneously. The change of the scene will appear on other user's window immediately. The users can "copy" and "paste" using the cooperative clipboards. They can also undo

e-mail: {david,toni,gallir,yuhua}@m3d.uib.es

any operation. As we can see in this situation, a control of concurrency is necessary to ensure the consistency of the global design.

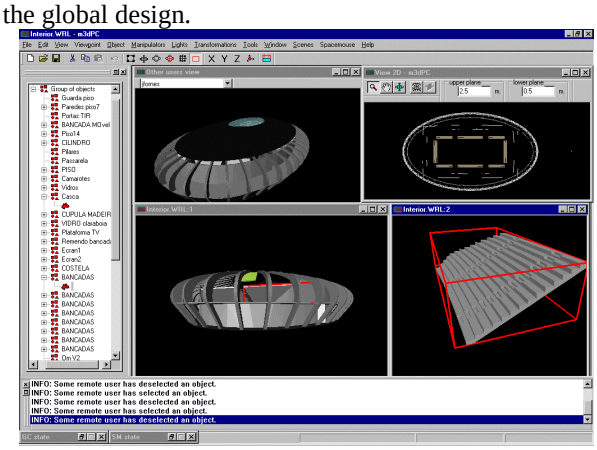

**Figure 1: The concurrent 3D design and visualisation tool**

Since three-dimensional rendering at interactive rates requires fast access to the geometric database, we decided to replicate the scene data on each site. Every replica of the data can be seen as a local copy of a persistent database. The VRML file format is used for data interchange. The three-dimensional design data are stored in a tree-like structure. Every element of the three-dimensional scene is represented by a node of the scene tree.

The cooperative 3D design tool runs on top of a distributed multi-user communications platform, called JESP [Alme95]. An overall scheme of the architecture of the system is shown in Figure 2. More detailed information about the architecture of the system can be found in [Gall97a, Luo99, Benn99, Sanc99].

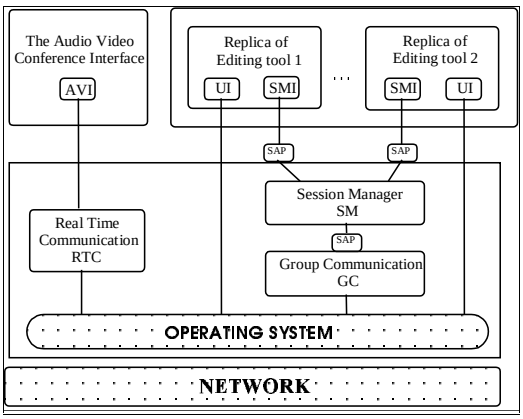


**Figure 2: Overall scheme of the M3D Editor**

### 2.1 The Mu3D Protocol

Maintaining the persistent database consistent is a crucial task in the cooperative working session. If the users are working with different replicas of the same data, they expect the replicas to be consistent. Modifications to the persistent database are sent to all the sites using a*n application protocol*, called *Mu3D* [Galli00]. This protocol is composed of small messages specifying session events and data changes. Every site is responsible of sending local modifications encapsulated in Mu3D messages to the other sites. When a site receives a message, the remote event is recreated locally.

Mu3D messages are classified into *session control messages, selection messages, scene control messages,* and *editing messages.*

### 2.2 Concurrency control

Concurrency control is necessary in cooperative working to resolve conflicts between participants, and to allow them to perform tightly coupled activities. The major concurrency control is realised via the *selection operation.* By this method, most operations that require concurrent control can be performed only on selected objects.

Every time the user selects an object, he becomes the *owner* of this object until it's deselected by him. Users cannot select an object if another user has already selected it. In other words, only one user can modify a given object at a time. This selection policy assures the consistency in the persistent database effectively by the implementation of the Mu3D protocol. Every selection and deselection is notified to the other sites by sending Mu3D messages.

## 3. MANAGEMENT OF MULTIPLE SCENES

Management of multiple scenes is an important mechanism in a multi-user, multi-site cooperative 3D Editor. It increases the productivity by allowing the individual user to work on multiple designs at a time.

The term "scene" is defined as a subset of interrelated geometrical data that can be rendered, visualised and modified by the user. Each scene can be composed of one or more objects, which will be added to it.

The user has to be able to load and unload scenes. The cooperative 3D editing tool will keep them in memory, so that the user can switch from one scene to another. This "scene switching" has to be reproduced in all the other users in the cooperative session, as well as the adding and removal of scenes. The implementation of the Scene Manager is described in the following sections.

### 3.1 Data Structure

The data structures for the scene management are based on the concept of Scene Graph*, used in Open Inventor and VRML*. A scene graph is a Directed Acyclic Graph that organises all the data related to the scene

(geometrical data, properties, illumination...). All these data are modelled as *nodes* of the graph. Editing the scene consists in modifying the parameters of the nodes or the graph topology (see [Wern94] for details).
As depicted in Figure 3, the editor maintains a *tree-like memory structure.* The tree is connected to the application viewer window, so that all the data displayed is stored by the editor will be beneath the tree.
The root of each scene is a Selection node that catches all the selection events. The first level of root descendants is composed of the following branches:

- A Lights Subtree, that handles all the lights in the scene.
- A Viewpoints Subtree, containing points of view attached to the scene.
- A group with common elements, such as the axes, and callback nodes, that are internally used by the application.
- A list of Avatars, this is, geometrical information that represents the position of remote users running the editing tool on their machines.
- A group of Sections, containing information to visualise a "cut" of the scene.
- A list of Worlds, parts in which the active scene is divided. These worlds contain all the geometry in the scene, and can be inserted into and deleted from the scene.

In addition to the scene data, the main tree contains also special internal data: a *Clipboards Subtree* and a *Trash Bins Subtree*.

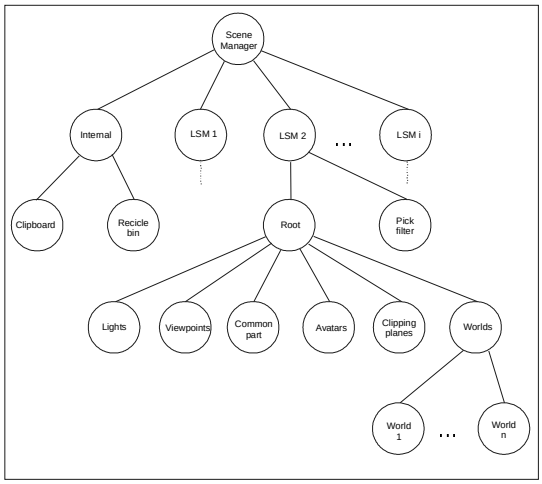


**Figure 3: The physical tree structure of the scenes**

### 3.2 Managing Standard Scenes

As described above, the scene is the basic entity for cooperative working. There is always one active scene, if the user wants to edit a non-active scene, then he must switch it to become the active one.
The following three basic operations are defined on the Scene Manager:

- Insertion of a scene. It is implemented adding an empty subtree under the Scene Manager Tree. This new scene becomes the active one.
- Removal of a scene. It is done removing the subtree from the Scene Manager Tree. If the removed scene is the active one, then another scene must be activated first. If there are no more scenes, then most operations are disabled until there is at least one scene.
- Switching the active scene. It is implemented detaching the root of the old scene from the viewer, and attaching the new one to it.

During a session, users can work in different scenes simultaneously. When the user performs an operation on a scene, the application broadcasts a message to the other participants. They will modify their local copy of the same scene.
Every scene is identified by a unique ID. Every message encompasses the Scene ID, so that changes are applied to the right scene.

## 4. COOPERATIVE CLIPBOARDS

The clipboard paradigm allows users to duplicate, move and interchange data within an application. A memory buffer, called *clipboard*, is used as intermediate storage. Data can be copied or moved ("cut" operation) to this buffer and then pasted into the scene.
Two clipboard operations are presented: c*opy* and *paste*. Note that the *cut* operation is no more than a copy followed by a deletion, and its description and implementation are trivial.

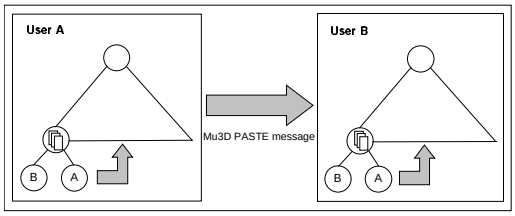


**Figure 4: Performing a cooperative "paste" operation**

As explained in previous section, *clipboards subtree* is stored as a child of the Scene Manager Tree. The scene manager tree contains the clipboards of every session participant. These clipboards are identified by a member ID. Any modification to the clipboards is sent to all members via Mu3D messages.

The messages *do not* contain the actual contents of the clipboard[1]. Instead, they contain the pointer to the original object that has to be copied or pasted. This information, along with the type of the message and the identification of the sender, assures that the clipboards are up-to-date.
When an user requests to *copy* the selected object to the clipboard, the local instance of the editor copies the object to the local clipboard and then a message is broadcast to the session members. When a site receives the message, the editor looks for the clipboard corresponding to the identifier of the sender, and copies the object to the appropriate clipboard.
When the user requests to *paste* the object in the local clipboard to the active scene, the local instance of the editor inserts a copy of the object and a message is broadcast to the other members. When a site receives the message, the editor searches for the clipboard corresponding to the member identifier of the sender, and puts a copy of this clipboard into the active scene. An example can be seen in Figure 4.

## 5. UNDO IN A COOPERATIVE ENVIRONMENT

To reverse (undo) an operation in a cooperative environment is not a trivial issue. Every change (forward or reverse) should be propagated to the other session participants with the extra constraint that changes made by a user could not be superseded or "reversed" by others.
We present in this section the implementation of the undo mechanism in the cooperative 3D editing tool. We also explain how the behaviour of undo operations has been restricted to allow it to work cooperatively. Our goal was to solve a concrete problem, we did not aim to develop a complete framework collaborative undo operations (as shown in [Prak92] and in [Prak94]). To implement the collaborative undo mechanism, we exploited the *selection* policy concept. We also defined several restrictions:

1. Users cannot undo someone else modifications. This is, the editor does not provide global undo. It is not desirable, indeed it can be very annoying, allowing a user to undo someone else operation.
2. Users must not be allowed to undo their modifications on an object that was later modified by someone else.
3. Any user has to be able to undo at least a significant subset of the last changes done to the scene by him.
4. Finally, the consistency has to be maintained in all the replicas of the persistent database, and must not be lost when a change is reversed.

### 5.1 Scopes of Undo

To implement an undo mechanism that fits the above restrictions, a new concept has been developed, called Scopes of Undo. We define the scope as the set of operations that can be reversed, at a given time. This way, in each moment users are able to undo only those actions that belong to the scope. A scheme showing the conceptual model of a scope of undo is depicted in Figure 5.
Actions subjected to undo include modifications in the *linear transformation*, the *rendering parameters*, and *vertices coordinates*, *insertion* and *deletion* of an object.
These actions can be divided in two groups, each one associated with one scope:

- The first scope is inside a selection. Users can undo and redo any action that takes place after the object has been selected, and before it is deselected.
- The second scope is the last action. Actions outside of an object selection policy belong to this scope, such as inserting or deleting objects.

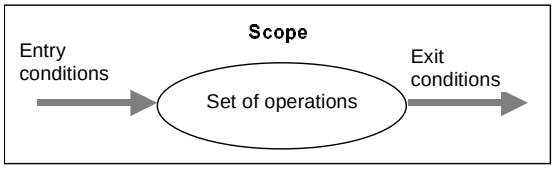


**Figure 5: Conceptual model of the scope of undo**

When an object is selected, several lists are created to store the state of the object before the changes. Each list contains a part of the current state of the selected object, according to the type of the editing operation. This part of the state includes linear *transformation matrices*, the current *rendering parameters*, or the set of *vertices* that compose the object.
These lists are reset every time the user makes a new selection, therefore the lists of states are always local to the current selection.
Before every atomic change is applied, a new element is added to the appropriate list. The following atomic changes are considered:

- When the user starts using an interactive manipulator (when he clicks on the manipulator to start changing the object).
- When the user changes any value of the transformation matrix of the object (via a numerical menu).

[1] The network load is keep minimum because the 3D object data is actually not sent for clipboard operations.

- When the user starts editing the rendering parameters (when he activates the material editor).

Other operations that are "out" of the selection scope (last action scope) are:

- Object deletion. The program will backup the deleted object to be able to restore it, when the user requests it.
- Object insertion. The program will save a pointer to the inserted object, so that it can be deleted if the user asks for it.

The scope for this kind of actions will be restricted to the *last change*. Changes prior to the last insertion or deletion cannot be reversed.

An undo condition has been specified for every recent modification on the scene. This condition will define when this modification can be reversed. It applies to modifications inside the last action scope, and it can be specified as follows:

An insertion can be undone only if the following conditions are true:

- The inserted object has not been *modified* (local or remotely).
- There have been no *remote insertions* between the insertion and its corresponding undo.
- There have been no *remote deletions* between the insertion and its corresponding undo.

In a similar way, a deletion can be undone only if the following conditions are true:

- There have been no *remote insertions* between the deletion and its corresponding undo.
- There have been no *remote deletions* between the deletion and its corresponding undo.

### 5.2 Data Structures for the Selection Scope

Some data structures are maintained to save information about the actions to undo inside a selection. We describe them briefly.

Several State Lists are maintained by every local replica, one for each type of modification to undo. Therefore, there is a list of the previous linear transformations, another list with the previous rendering parameters, and another containing the vertices of the object. These lists are updated every time the user is about to modify the selection.

Since there is more than one single list, the application must know which type was the last modification to be able to access the appropriate list. Another Master List is maintained, containing the types of the last changes performed.

When the user requests to undo the last modification to the selected object, the program queries the master list, checks the last change type, and restores the values from the appropriate list.

### 5.3 Data Structures for the Last Action Scope

A set of data is maintained to check if a reversible change (an insertion or a deletion) was performed. These data will be reset if the user performs some operation that invalidates the undo operation.

The application maintains a pointer to the *last inserted object* to delete it again if the user requests an undo. A backup of the *last deleted object* is saved in an alternate scene tree (named "Trash"). To undo this deletion, the *position* of this object and its previous *location* are maintained.

### 5.4 Updating the Data Structures

Every time the user makes a new selection, the state lists and the master list are emptied.

Every time the user is about to make a change inside the Selection Scope, the data structures described above are updated, adding the previous state. These updates are done when:

- The user starts dragging on an interactive manipulator.
- The user is about to change the geometrical transform values, via a numerical dialog.
- The user starts an interactive material editor.
- The user is about to change a set of vertices of the selected object.

In these cases, the following steps are followed:

- The previous state is added to the appropriate list (the previous rendering parameters to the rendering parameters list, or the previous linear matrix to the transform list).
- The type of change is added to the master list.

Before the user inserts a new object into the scene, the path pointing to the new object is stored in memory.

Before the user deletes an object from the scene, a copy of the object is stored in a special place in memory (named the Trash Bin). All sites have a local copy of the trash bins of the other sites. If the deletion is done locally, the object will be stored in the own trash bin. If the deletion is done remotely, the object will be stored in another trash bin.

### 5.5 Performing the Undo Operation

When the user chooses to undo into the Selection Scope, the following actions take place:

1. Check the type of the last change.
2. Restore the old values from the appropriate list to the appropriate attribute of the selected object.
3. Remove the old state from the list.
4. Send the modification to the remote sites.
5. Remove the type of the last action from the master list.

When the user wants to undo an insertion, the following actions are performed:

1. Select the last inserted object.
2. Delete it (the deletion is already sent to the remote users).
3. Reset the pointer to the last inserted object.

When the user chooses to undo a deletion:

1. Restore the object from the own trash bin in the location previously stored.
2. Send the appropriate message to the remote sites to restore the object from their local copies of our local trash bin.
3. Reset the location values.

### 5.6 Leaving the Undo Scope

When an event prevents further undo of previous operations, the following steps are carried up:

- When an object is deselected, all the state lists and the master list are emptied and reset.
- When the event invalidates the undo of an *insertion*, the pointer to the object that was inserted is reset.
- Similarly, when the event invalidates the undo of a *deletion*, the trash bin is emptied, and the position of the deleted object is reset.

## 6. CONCLUSIONS

We have presented in this paper the implementation of three mechanisms for a 3D cooperative design and visualisation tool: Management of multiple scenes, cooperative clipboards and undo. Our implementation maintains the consistency between all the replicas of the scene, and minimises the network traffic. The Scene Manager also allows us to implement mechanisms such as the "undelete" and the "clipboard operations" in a simple and elegant way.

Further work can be applied to make the undo mechanism more general, and to further exploit the scene management module to allow the editor to deal with other common editing operations.

## 7. REFERENCES


[Alme95] A. Almeida and C. A. Belo. "Support for Multimedia Cooperative Sessions over Distributed Environments." *Proc. Mediacomm'95*, Society for Computer Simulation, Southampton, April, 1995.

[Benn99] Toni Bennasar Obrador, Ricardo Galli, Yuhua Luo, "Edición de objetos 3D con Open Inventor en entornos Windows NT", *Actas del Congreso, IX Congreso Español de Informática Gráfica*, Jaén, 16-18 de junio de 1999, CEIG'99. Pp. 381-382.

[Gall97a] R. Galli, P. Palmer, M. Mascaro, M. Dias, Y. Luo. "A Cooperative 3D Design System." *Proceedings of CEIG97*, Barcelona, Spain, June, 1997.

[Galli97b] R. Galli, P. Palmer, M. Mascaro, M. Dias, Y. Luo, "CODI - A System for Cooperative 3D Design." *Proceedings of 1997 IEEE Conference on Information Visualization*, pp. 286-293. London, August 27-29, 1997.

[Gall99] R. Galli, Y. Luo, D. Sánchez, S. Alves, M. Dias, R. Marques, A. Almeida, J. Silva, J. Manuel , , B. Tummers (EDC), Ed. R.Galli, "M3D technical specifications", ESPRIT Project No. 26287 M3D, Deliverable 1.2, April1999.

[Gall00] R. Galli, Y. Luo. " Mu3D: A Causal Consistency Protocol for a Collaborative VRML Editor", *Proceeding of the VRML'2000 Symposium*, Monterey, CA, Feb 2000, ACM SIGGRAPH.

[Luo98a] Yuhua Luo, Ricardo Galli, Miguel Mascaro, Pere Palmer, "Cooperative Design for 3D Virtual Scenes," *Third IFCIS Conference on Cooperative Information Systems* (CoopIS'98), August 1998, New York, U.S.A.

[Luo98b] Y. Luo, R. Galli, M. Mascaro, P. Palmer, F. J. Riera, C. Ferrer, S. F. Alves, "Real Time Multi-User Interaction with 3D Graphics via Communication Network," *Proceedings of IEEE 1998 Conference on Information Visualization*, July 1998, London.

[Luo99] Yuhua Luo, Ricardo Galli, Antonio Carlos Almeida, Miguel Dias, "A Prototype System for Cooperative Architecture Design." *Proceedings of IEEE 1999 International Conference on Information Visualization*, July 1999, London, pp. 582-588.

[Prak92] Atul Prakash and Michael J. Knister. "Undoing actions in Collaborative Work". In *Proceedings of ACM CSCW'92 Conference on Computer-Suported Cooperative Work*, pages 273-280, October 1992.

[Prak94] Atul Prakash and Michael J. Knister. "A Framework for Undoing Actions in Collaborative Systems". *ACM Transactions on Computer-Human Interaction*, 1(4):295-330, 1994.

[Sanc99] David Sánchez Crespillo, Ricardo Galli, Yuhua Luo, "Un editor 3D interactivo para trabajo cooperativo," *Actas del Congreso, IX Congreso Español de Informática Gráfica*, Jaén, 16-18 de junio de 1999, CEIG'99.

[Wern94] J. Wernecke. *The Inventor Mentor*. Addison-Wesley Publishing Company, 1994.