1993

An Interactive X-Window Based User Interface For The XoX Solid Modeling Library

Poting Wu

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

Report Number:
93-015

http://docs.lib.purdue.edu/cstech/1033

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
AN INTERACTIVE X-WINDOW BASED USER INTERFACE FOR THE XoX SOLID MODELING LIBRARY

Poting Wu
Elias N. Houstis

CSD-TR-93-015
January 1993
An Interactive X-Window Based User Interface for the XoX Solid Modeling Library

Poting Wu and Elias N. Houstis

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907, U.S.A.
e-mail: wu@cs.purdue.edu

CAPO Report CAPO-93-08
January, 1993

Abstract

In this report we describe an interactive user interface XXoX for XoX geometry modeling library [XoXE 92]. XXoX is implemented in the X-window system using Motif widget. The XXoX environment allows the creation of 3-D and 2-D objects using CSG type primitive operations. Furthermore, one can manipulate the specified geometries by orienting, combining, cutting, and deforming the objects. XXoX provides powerful multiple user-interfaces, UNDO/REDO functions, and a command language preprocessor that allows the saving, editing and interpretation of the object's specification file. Examples are presented to demonstrate the functionality and applicability of this interactive system.

This work was supported by NSF grants 9123502-CDA and 9202536-CCR, AFOSR F49620-92-J-0069 and PRF grant 6902003.
CONTENTS

Chapter 1  An Introduction to the XXoX - A Solid Modeling System  1
   1.1 Higher level programming interface to the Motif widget  1
   1.2 Multiple User-Interfaces  2
   1.3 UNDO/REDO functions  2
   1.4 Operations in CSG - Constructive Solid Geometry  3

Chapter 2  Higher Level Programming Interface to the Motif Widget  4
   2.1 Widget Tree Data Structure  5
   2.2 Examples  5
      2.2.1 Example1 - Basic Outline of XXoX  6
      2.2.2 Example2 - Selections of Various Parents  8

Chapter 3  Multiple User - Interfaces  9
   3.1 Interface of Buttons and Dialogs  10
      3.1.1. Tree Structure of Menu Area  10
      3.1.2. Tree Structure of Operation Area  12
      3.1.3. Tree Structure of Viewing Option Area  14
   3.2 Interface of Command Area  15
      3.2.1 Constructive Solid Geometry Commands  15
      3.2.2 Maintaining History Lists  20

Chapter 4  Implementation of UNDO / REDO Functions  21
   4.1 Basic Undo Primitives  22
      4.1.1 Abort Command  22
      4.1.2 Reversible Command  22
   4.2 REDO List  24
   4.3 Function Inversion and Backtracking  26
   4.4 Source - to - Source Preprocessor  27

References  28

APPENDIX - A  XXoX Execution Arguments  29

APPENDIX - B  Specifications of Widget Tree Nodes  30
   B.1 Variables of Creating Menu Widget - in create_menu()  30
   B.2 Variables of Creating Dialog Widget - in create_dialog()  33
APPENDIX - C  Scanner & Parser for Command Area Interface  34
C.1 Keywords  34
C.2 Identifiers, File names, Numbers, Data sets, and Strings  35
C.3 Grammars  36

APPENDIX - D  Examples  42
D.1 Example1 - Engine  42
D.2 Example2 - Vase  43
D.3 Example3 - Chain  44
Chapter 1
An Introduction to the XXoX - A Solid Modeling System

XXoX - A Solid Modeling Software turns your workstation into a powerful 3D geometry specification system. With XXoX you can create 3D primitives and 2D outlines of cross-sections, manipulate the geometry by orienting, combining, cutting, and deforming the objects. XXoX provides extremely powerful multiple user-interfaces, and the UNDO/REDO functions by using the higher level programming interface of the Motif widget in one integrated program.

Developing the user interface for a solid modeling system provides an excellent opportunity to put into practice the interface design techniques. A variety of techniques lend themselves well to graphical interfaces, including the direct application of regularized Boolean set operators, transforming, sweeping, and others. In Constructive Solid Geometry (CSG) systems, the users may be allowed to edit the object by modifying or replacing one of the leaf solids or subtrees. That is, a solid modeling system that can let the users create objects in terms of several representations, while storing them in another.

Roughly speaking, this solid modeling application contains the following important concepts and phases:

1.1 Higher level programming interface to the Motif widget

In this phase, we create the widgets tree to the whole world of programming our applications. We separate the creation into three steps. First, we describe the contents of the application widgets in an array tree. Then we initialize the X intrinsic to hold the various entries in the application widgets. Finally, we define two functions create_menu() and create_dialog() that use the widget description to create the widget entries, assigns the callback functions, and creates their subwidget trees by calling themselves recursively.

According to our design, it is very easy to add a new entry to the widget tree. Adding a new entry requires only defining the relative node properties and inserting the entry to the array that describes the widget tree.
1.2 Multiple User-Interfaces

In this phase, we give the notion of multiple user-interfaces. With these interfaces users have several different ways to communicate with the machines. Indeed, users can input their commands into XXoX by two different methods - either from the Buttons and Dialogs or the Command Area.

XXoX creates the Buttons and Dialogs interface which allows users to input data by pushing some specific buttons and entering data into their relative prompted dialogs. After XXoX received users’ input, it generates the relative commands into the Command Area. These include the implementation of Buttons and Dialogs by calling their specified callback functions, and translation from interactive data to command string.

XXoX also allows users to input their commands into the Command Area directly. This means that there is a parser in XXoX to parse and recognize the input commands. And this parser is responsible to execute the relative actions and update their corresponding Buttons and Dialogs. That is, if the users input some changes such as Viewing Coordinate into the Command Area, then XXoX is responsible to update Buttons and Dialogs of Viewing Options.

There is a scrolled History Area inside the Command Area which records all the input commands. Since XXoX will automatically generate the relative commands after those activities of the Buttons and Dialogs, and/or store those input commands from the Command Area, the History Area will always record all users’ responses in a historical list. In order to be more flexible, XXoX maintains the histories for saving and/or retrieving batch files. That is, if the users finish the drawing and decide to keep this result into a file, then XXoX will create a batch file which record all the users’ previous work. Next time, when the users retrieve the batch file, they can get the same result without reconstructing the drawing.

1.3 UNDO/REDO functions

In this phase, we are including the implementation of the UNDO/REDO functions into XXoX. UNDO/REDO are very useful features in many applications such as in programming languages, programming environments, database systems, user interfaces, and so on. Especially for XXoX - a Solid Modeling Software with boolean, transforming, and sweeping operations.

While in implementing these two features, we will face many serious efficiency problems. How to remove an object when we choose to uncreate it? How to restore an object when we choose to undelete it? In order to encounter these problems, we need the parser and the history list as we implemented in the previous phase. With the parser, XXoX can recognize all the input commands no matter how they are input. Thus, we can efficiently implement both the undo and redo tasks by using this history list. And when saving the batch files, we can store the history into the file that is the net effect of the drawing result.
1.4 Operations in CSG - Constructive Solid Geometry

In CSG - Constructive Solid Geometry, simple primitives are combined by means of regularized Boolean set operators and Transformation operators that are included directly in the representation. An object is stored as a tree with operators at the internal nodes and simple primitives at the leaves. Some nodes represent Boolean operators, whereas others perform translation, rotation, and scaling. Since Boolean operations are not, in general, commutative, the edges of the tree are ordered. With the concept of CSG, the ability to edit models by deleting, adding, replacing, and modifying subtrees, coupled with the relatively compact form in which models are stored, have made CSG one of the dominant solid modeling representations.

For example, there are four objects: cylinder-1, cylinder-2, cylinder-3, and box-1. These objects are the primitives that we have in the system now. Then we apply the Boolean and Transform operators to these primitives and get our result. The CSG can be represented as follows:

CSG Tree:
Chapter 2

Higher Level Programming Interface to the Motif Widget

Most applications have many layered menus, sophisticated popup dialogs, working drawing areas, control scroll bars, and text fields for command entry, and so on. All of them are decorated by three-dimensional frames and line separators, arranged by attachment forms and rowcolumns, and described by labels and pixmaps. Creating each widget as in the previous items can be tedious, although the approach allows maximum flexible in configuring widgets and setting the resources of each individual button and list. However, it is often easier to wrap the steps of the widget creation in some higher level functions.

The basic idea for the construction of our higher level programming interface to the Motif widget comes from Douglas A. Young’s book: "The X Window System Programming and Applications with X1, OSFIMotif Edition". We extend his interface in creating menu widgets (p.96) to the whole world of programming our applications. We can separate the widget creation into three steps. First, the contents of the application widgets must be described in an array tree. Then, the application must initialize the X intrinsic to hold the various entries in the application widgets. Finally, the functions create_menu() and create_dialog() use the widget description to create the widget entries, assign the callback functions, and also create their subwidgets by calling itself recursively.
2.1 Widget Tree Data Structure

First, let's look at a structure that defines the widget information needed for each entry in the widget tree.

```c
typedef struct Node {
    char *title ; /* name of the node */
    int type ; /* node type */
    void (*func)() ; /* callback to be invoked */
    caddr_t data ; /* data for the callback */
    char *sub_title ; /* title of subnode */
    struct Node *sub ; /* data for subnode of this node */
    int n_sub_items ; /* how many items in sub_node */
    Widget *widget ; /* pointer to the node widget */
    char *pattern ; /* name of pixmap pattern */
    char *insens ; /* name of insensitive pixmap pattern */
    char **accel ; /* accelerator & mnemonics */
} NODE ;
```

This structure records the name of a widget entry, type of a widget (detail described in Appendix - B), a callback function to be invoked if that entry is selected, any additional data to be passed to that callback function. In addition, if the entry has a subwidget tree attached to it, the structure contains a pointer to an array of type NODE, the length of the array, and also the name of the subwidget tree. Thus, an array of type NODE defines a tree of widgets with their subwidgets. The structure also contains a pointer to the created widget which is globally accessible, two registered pixmap names which replace the text labels either in normal status or in insensitive status, and a string array which stores the accelerator and mnemonics characters.

Appendix - B will include the detail content specifications of the Widget Tree Nodes.

2.2 Examples

Before we look at the function that creates widget entries from this structure, let's look at how this can be used in an application. The following examples show how clearly and easily the application to initialize the X intrinsic to hold the various entries in the application widgets.
### 2.2.1 Example1 - Basic Outline of XXoX

```c
#include <anything_it_need.h>

main(argc, argv)
int argc;
char *argv[];
{
    Widget toplevel;

    /* Initialize the X Intrinsics. */
    toplevel = XtInitialize(argv[0], "XXox",
                            Options, XtNumber(Options), &argc, argv);

    /* Create the widgets tree. */
    create_menu(NULL, toplevel, WidgetData, WidgetNo);

    /* Realize the widgets. */
    XtRealizeWidget(toplevel);

    /* Enter the event loop. */
    XtMainLoop();
}
```

Outline of Tree:
And the descriptions of the array tree in the example application.

```
NODE MenuObj[] = {
    ... /* subwidget tree for menu entries */
};

NODE MenuData[] = {
    {NULL, muPullDown, NULL, NULL, "menu", MenuObj, XtNumber(MenuObj)},
};

NODE CmdData[] = {
    {"Cmd", muCommand, parse, (caddr_t)4, NULL, NULL, 0, &CmdWidget},
};

NODE OperObj[] = {
    ... /* subwidget tree for buttons operations */
};

NODE OperData[] = {
    {"oper", muFrame, NULL, NULL, NULL, OperObj, XtNumber(OperObj)},
};

NODE ViewObj[] = {
    ... /* subwidget tree for viewing options */
};

NODE ViewData[] = {
    {"view", muFrame, NULL, NULL, NULL, ViewObj, XtNumber(ViewObj)},
};

int DrSize[] = { WIDTH, HEIGHT }; /* size of drawing area */

NODE DrawData[] = {
    {NULL, muDraw, NULL, (caddr_t) DrSize, NULL, NULL, 0, &DrawWidget},
};

int FormAttach[][4] = { {-1, 0, -1, -1 }, { 0, -1, -1, -1 },
    { 1, 2, -1, 0 }, { 1, 2, 0, -1 },
    { 1, 2, 3, 4 } };

NODE WidgetData[] = {
    {NULL, muFormItem, NULL, (caddr_t) FormAttach[0], NULL, MenuData, XtNumber(MenuData)},
    {NULL, muFormItem, NULL, (caddr_t) FormAttach[1], NULL, CmdData, XtNumber(CmdData)},
    {NULL, muFormItem, NULL, (caddr_t) FormAttach[2], NULL, OperData, XtNumber(OperData)},
    {NULL, muFormItem, NULL, (caddr_t) FormAttach[3], NULL, ViewData, XtNumber(ViewData)},
    {NULL, muFormItem, NULL, (caddr_t) FormAttach[4], NULL, DrawData, XtNumber(DrawData)},
};

int WidgetNo = XtNumber(WidgetData); /* root of widget tree */
```
2.2.2 Example 2 - Selections of Various Parents

It is easier to add new entries to a widget tree. Rather than having to read the code and trying to decipher the widget tree structure, the widget tree structure is clearly defined by the NODE arrays. Adding a new entry requires only defining relative node properties and adding an entry to the array that describes the widget tree.

Because of the data structure of the widget tree, we can modify the construction of our application outline very easily. Following examples show how XXoX links those children subtrees with different parents.

Outline of Trees:
Chapter 3
Multiple User - Interfaces

What do multiple user-interfaces mean in XXoX? Multiple user-interfaces mean that the users can communicate with the machines in different ways. In XXoX, we allow users to input their commands either by the Button and Dialogs or by the Command Area.

From Button and Dialogs, XXoX will show some suitable dialogs for users to input data. If users want to input their commands through this way, then all they need to do is to click the corresponding button and give the necessary data for the objects they choose. Other alternative way is to input their commands through the Command Area directly. The latter one is especially useful if the users are familiar with the CSG command language and want to do their work more quickly. In both alternatives, XXoX has the ability and the responsibility to generate the corresponding commands from both the Button and Dialogs and the Command Area, and update the relative buttons and dialogs from current command. In fact, we design a parser to recognize the input commands that the users made in XXoX.

The outline of these two user-interfaces is shown as following picture:
3.1 Interface of Buttons and Dialogs

XXoX creates the *Buttons and Dialogs* interface which allows users to input data by pushing some specific buttons and entering data into their relative prompted dialogs. After XXoX received users' input, it generates the relative commands into the *Command Area*. These include the implementation of *Buttons and Dialogs* by calling their specified callback functions, and translation from interactive data to command string.

This interface includes Menu Area, Operation Area, and Viewing Option Area. And their relative widget trees are as follows:

**3.1.1. Tree Structure of Menu Area**

```
Menu
  File
    Load Shapes Object
    Save Shapes Object
    Load Polygon Object
    Save Polygon Object
    Load Batch File
    Save Batch File
    Save & Run PATRAN File
    Restart PATRAN
    Save Screen to XPM File
    Quit
  Edit
    Undo
    Redo
    Skip
    Copy
    Delete
    Clear all
  View
    Boundary
    Mesh
    Hidden-Mesh
    Boundary-Shade
    Mesh-Shade
    PS-Shade
    Gouraud-Shade
    Phong-Shade
    Flat-Shade
  Paint
  Help
```

**Reference Commands**

- `gload`
- `gsave`
- `pload`
- `psave`
- `bload`
- `bsave`
- `patran,neutral,mesh`
- `xpm`
- `quit`
- `undo`
- `redo`
- `skip`
- `=`  
- `delete`
- `clear`
And their relative popup dialogs are shown as follows:
3.1.2. Tree Structure of Operation Area

Reference Commands

<table>
<thead>
<tr>
<th>Select Color</th>
<th>Reference Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>color(red)</td>
</tr>
<tr>
<td>Green</td>
<td>color(green)</td>
</tr>
<tr>
<td>Blue</td>
<td>color(blue)</td>
</tr>
<tr>
<td>Cyan</td>
<td>color(cyan)</td>
</tr>
<tr>
<td>Magenta</td>
<td>color(magenta)</td>
</tr>
<tr>
<td>Yellow</td>
<td>color(yellow)</td>
</tr>
<tr>
<td>Orange</td>
<td>color(orange)</td>
</tr>
<tr>
<td>Pink</td>
<td>color(pink)</td>
</tr>
<tr>
<td>White</td>
<td>color(white)</td>
</tr>
<tr>
<td>Tan</td>
<td>color(tan)</td>
</tr>
<tr>
<td>Point</td>
<td>point</td>
</tr>
<tr>
<td>Line</td>
<td>line</td>
</tr>
<tr>
<td>Arc</td>
<td>arc</td>
</tr>
<tr>
<td>Circle</td>
<td>circle</td>
</tr>
<tr>
<td>Sheet</td>
<td>sheet</td>
</tr>
<tr>
<td>Disc</td>
<td>disc</td>
</tr>
<tr>
<td>Box</td>
<td>box</td>
</tr>
<tr>
<td>Sphere</td>
<td>sphere</td>
</tr>
<tr>
<td>Cylinder</td>
<td>cylinder</td>
</tr>
<tr>
<td>Cone</td>
<td>cone</td>
</tr>
<tr>
<td>Scale</td>
<td>scale</td>
</tr>
<tr>
<td>Translate</td>
<td>translate</td>
</tr>
<tr>
<td>Rotate</td>
<td>rotate</td>
</tr>
<tr>
<td>Intersection</td>
<td>&amp;</td>
</tr>
<tr>
<td>Union</td>
<td>/</td>
</tr>
<tr>
<td>Difference</td>
<td>-</td>
</tr>
<tr>
<td>Sweep1</td>
<td>sweep1</td>
</tr>
<tr>
<td>Sweep2</td>
<td>sweep2</td>
</tr>
<tr>
<td>Skin</td>
<td>skin</td>
</tr>
</tbody>
</table>
And their relative popup dialogs are shown as follows:

Point, Line, Arc
Circle, Sheet, Disc
Box, Sphere, Cylinder
Cone, Sweep, Skin
3.1.3. Tree Structure of Viewing Option Area

Reference Commands

- select
- eyexyz1, eyedeg1
- eyexyz0, eyedeg0
- lightxyz, lightdeg
- center
- zoom
- unzoom
- zoom
- conform
- conform
- unconform
3.2 Interface of Command Area

To implement the Command Area Interface including scanner and parser, first we need to redirect the content of the Motif Command widget to the scanner which is generated by lex, or flex. Then define the grammar of the XXoX application CSG (Constructive Solid Geometry) commands for the parser which is generated by yacc, or bison.

Appendix - C will include the detail about their lexical rules and grammars.

3.2.1 Constructive Solid Geometry Commands

The available CSG (Constructive Solid Geometry) and viewing control commands in XXoX are listed as follows:

1. Objects Assignment:

   \[ id = object \]
   Assign object to an identifier id.

2. Primitives Creations:

   point (point)
   Create a point object at point.

   line (point1, point2)
   Create a line object from point1 to point2.

   arc (point, num1, num2, num3)
   Create an arc object center at point with radius num1, beginning angle num2, and ending angle num3.

   circle (point, num)
   Create a circle object center at point with radius num.

   sheet (point, num1, num2)
   Create a sheet object corner at point with length num1 and width num2.

   disc (point, num)
   Create a disc object center at point with radius num.
box (point , num1 , num2 , num3 solid)
Create a box object corner at point with length num1, width num2, and height num3 of shell type solid - shell or solid (default is solid).

sphere (point , num solid)
Create a sphere object center at point with radius num of shell type solid - shell or solid (default is solid).
Create a sphere object center at point with radius num of shell type solid - shell or solid (default is solid).

cylinder (point , num1 , num2 solid)
Create a capped cylinder object center at point with radius num1 and height num2 of shell type solid - shell or solid (default is solid).

cone (point , num1 , num2 solid)
Create a capped cone object center at point with radius num1 and height num2 of shell type solid - shell or solid (default is solid).

sweep1 (swept-object , path-object)
Sweep an swept-object along another path-object.

sweep2 (swept-object , path-object , point)
Sweep an swept-object along another path-object while maintaining the relative position point of the swept-object with respect to the tangent space of the path-object.

skin (num1 , num2 , shape1 , shape2 , objects-list)
Create a surface that fits through given 1-dimensional objects-list with degrees num1 & num2 and shape options shape1 & shape2 - XOX_C2 or XOX_Patches.

3. Transformation Operators:

scale (object , point)
Scale object by three scale factors point.

translate (object , point)
Translate object by a specified vector point.

rotate (object , point1 , point2 , num)
Rotate object about a user specified axis with beginning point point1, ending point point2, and rotate angle num in degree.
4. Boolean Operators:

\[ \text{object1} \& \text{object2} \]
Find the intersection of object1 and object2.

\[ \text{object1} \mid \text{object2} \]
Find the union of object1 and object2.

\[ \text{object1} - \text{object2} \]
Find the difference of object1 and object2.

5. File Input/Output Interfaces:

\texttt{gload (file_id)}
Load object from XOX format file file_id.

\texttt{gsave (id , file_id)}
Save object id to XOX format file file_id.

\texttt{pload (file_id)}
Load object from Polygon format file file_id.

\texttt{psave (id , file_id)}
Save object id to Polygon format file file_id.

\texttt{bload (file_id)}
Load batch file file_id to internal command history and execute.

\texttt{bsave (id , file_id)}
Select object id for the active target and save internal command history to batch file file_id.

\texttt{xpm (file_id)}
Save drawing window to XPM format file file_id.

\texttt{quit}
Quit the Solid Modeling program.

6. Editing Commands:

\texttt{select (id)}
Select active object id.
delete (id)
   Delete object id.

clear
   Delete all objects.

undo
   Undo the previous command.

redo
   Redo the previous Undo command which is on the top of redo list.

skip
   Skip the previous Undo command which is on the top of redo list.

7. Viewing Options:

color (colors)
   Set default color associated with objects to colors - red, green, blue, cyan, magenta, yellow, orange, pink, white, or tan.

conform
   To automatically scale the viewbox associated with a graphics window to include all of its currently associated objects entities.

unconform
   Turn off the automatically conformation.

eyexyz0 (point)
   Set a three Cartesian coordinate point that define the end point of the eye vector for a window.

eyedeg0 (point)
   Set a three Spherical coordinate point that define the end point of the eye vector for a window.

eyexyz1 (point)
   Set a three Cartesian coordinate point that define the begin point of the eye vector for a window.

eyedeg1 (point)
   Set a three Spherical coordinate point that define the begin point of the eye vector for a window.
lightxyz (point)
  Set the direction of the three Cartesian coordinate point light source for a window.

lightdeg (point)
  Set the direction of the three Spherical coordinate point light source for a window.

zoom (num1, num2, num3, num4)
  Zoom on an unitless rectangular sub-region (num1, num2) - (num3, num4) of a window.

unzoom
  Reset all panning & zooming which have taken place.

center (num1, num2)
  Pan to an unitless location (num1, num2) in window.

8. Foreign Software Interfaces:

  patran (yes_no1, yes_no2, yes_no3 patran_file)
  Generate the corresponding PATRAN language to session file patran_file (default is "xxox.ses"). Where yes_no1: generate mesh, yes_no2: output neutral file, yes_no3: run PATRAN.

  neutral (string, yes_no1, [yes_no2], yes_no3 neutral_typ)
  Output Neutral file with title string of format neutral_typ - neutral, NASTRAN, ANSYS, ABAQUS, or IGES (default is neutral). Where yes_no1: include Phase-I data, yes_no2: include GFEG/CFEG Table, yes_no3: include Phase-0 data.

  mesh (elmCtyp, mesh_opt, datas obj_nme)
  Generate Finite Element Mesh on object obj_nme or all objects with element type elmCtyp - TRI, QUAD, WEDGE, HEX, or BAR, option mesh_opt - ISOPARAMETRIC, NUMBER, or LENGTH, and its relative datas.
3.2.2 Maintaining History Lists

There is either a scrolled History List inside the Command Area, or a single Motif List Widget if users disable the Command Area, which records all the input commands. Since XXoX will store the input commands from the Command Area, and automatically generate the relative commands even from the Buttons and Dialogs, the list in the History List will contain all users’ responses.

In order to implement this feature, it is possible to let XXoX has the power for storing the command history into a script file and/or restoring the command script from the file into the command history. The batch script files are very useful when the users want to keep the drawings result they made so that they can retrieve the same result for the object without reconstructing the drawing.

In fact, the parser we implement in XXoX can be used in our next subject -- UNDO/REDO functions. We will discuss and compare the design decisions of the UNDO/REDO features in XXoX on next chapter.
Chapter 4
Implementation of UNDO / REDO Functions

Interactive systems, such as editors and program development environments, allow users to construct and modify data objects in real time. Since users may make some mistakes and change their minds, it is very important to support facilities that permit users to reverse the effects of the past actions and to restore an object to the previous state. Such abilities are called UNDO/REDO features.

A crude definition of UNDO is to restore a previous state in a user’s program or application. In fact, the simplest interface to recovery facilities is the UNDO command. UNDO can be used in a lot of applications such as fault-tolerate systems, database systems, programming languages and environments, editors, formatters, and so on.

What is a REDO feature? REDO is a facility that can cause the last command that was undone by the UNDO to be resubmitted. It happens when a user tries to resubmit some command that has been undone incorrectly or to repeat/copy some command that is already present at an earlier point.

If a user interface includes REDO as well as UNDO, then the user interface can provide a complete recovery scheme. UNDO/REDO are the basis of recovery in many systems, but their semantics may be quite different in those systems. For example, in some system, UNDO is a command rather than a meta-command so it is itself appended to the history list. While in other system, UNDO is to undo the execution of the previous command and to allow users to specify any subsequence of the history list to be undone.
4.1 Basic Undo Primitives

In XXoX, we implement the *Undo* theories to the solid modeling applications. There are two kinds of undo actions in XXoX - *Abort command* and *Reversible command*.

### 4.1.1 Abort Command

The abort command is to cancel the effect of current activity, that is to negate obvious mistakes of the previous command. For example, when users active the "clear all" command in XXoX, XXoX will show the below warning prompt dialog to the users. If the users decide to do it, they can just click the *OK* button. Otherwise, click the *CANCEL* button to ignore this action.

![Warning Popup](image)

### 4.1.2 Reversible Command

The reversible command is not only to negate obvious mistakes but also to handle transactions that are perceived as correct but subsequently found to be erroneous. In XXoX, users can apply the reversible command on all functions it supported by *Undo* command. For example,
There are two types of reversible commands for undo: *forward undo* and *backward undo*.

Assume we have an undo list as follows:

\[
< S_1, S_2, \ldots, S_{n-1}, S_n >
\]  

(a). **Forward Undo (edit command)** -- builds a new state from the current one and undo lists. The new undo list is constructed from a sequence of states.

If we perform the *forward undo* to the undo list \([1]\), then it becomes:

\[
\text{Undo(1) [1]} \Rightarrow < S_1, S_2, \ldots, S_{n-1}, S_n, S'_{n-1} >
\]

It will copy state \(S_{n-1}\) to a new state \(S'_{n-1}\). But there are two disadvantages when we use this approach:

I. Expensive for storing information, particularly in Solid Modeling.

II. Ambiguous undo actions -- For example, if we undo the result of \([2]\) again, there were two possible ways to do:

\[
\text{Undo(1) [2]} \Rightarrow < S_1, S_2, \ldots, S_{n-1}, S_n, S'_{n-1}, S'_{n-2} >
\]

or

\[
\text{Undo(1) [2]} \Rightarrow < S_1, S_2, \ldots, S_{n-1}, S_n >
\]

That is, Undo(1) Undo(1) \([1]\) is not equal to Undo(2) \([1]\).

(b). **Backward Undo (meta-command)** -- applies the current undo list to the current state to obtain a new state and reinstates the undo list at that point.

If we perform the *backward undo* to the undo list \([1]\), then it becomes:

\[
\text{Undo(1) [1]} \Rightarrow < S_1, S_2, \ldots, S_{n-1} >
\]

It will truncate the history of the computation. The disadvantage of this approach is that it cannot undo the undo if we truncate the history of computation.

To solve this problem, in XXoX we keep a *Redo list* as well as an *Undo list*. In other words, we keep the truncated history in the *Redo list* when we perform the *Undo* command.
4.2 **REDO List**

In order to undo the previous *Undo* commands -- *Redo* commands, in XXoX we use a *Redo list* as well as an *Undo list*. For example, if we have the undo list the same as [1] and we perform UNDO(3) [1] command, then we can have the following undo/redo lists:

\[
\begin{align*}
\text{Undo List:} & \quad <S_1, S_2, \ldots, S_{n-1}, S_n> \\
\text{Redo List:} & \quad <S_n, S_{n-1}, S_{n-2}>
\end{align*}
\]

The advantage of keeping both the redo and undo lists are:

(a). **Easily REDO the UNDO commands** -- as we mentioned in the previous section.

(b). **The SKIP command is possible** -- this command allows a systematic approach to the ambiguities inherent in the *Redo* command.

\[
\begin{align*}
\text{Undo List:} & \quad <S_1, S_2, \ldots, S_{i-1}, S_i, S_{i+1}, \ldots, S_{n-1}, S_n> \\
\text{Redo List:} & \quad <S_n, S_{n-1}, S_{n-2}>
\end{align*}
\]

(c). **The INSERT command is possible** -- this command is the inverse command of SKIP command for logically consistent alternatives.

\[
\begin{align*}
\text{Undo List:} & \quad <S_1, S_2, \ldots, S_{i-1}, S_i, S_{i+1}, \ldots, S_{n-1}, S_n> \\
\text{Redo List:} & \quad <S_1, S_2, \ldots, S_{i-1}, S_i, S_{i+1}, \ldots, S_{n-1}, S_n>
\end{align*}
\]
In XXoX, users can apply these two types of commands -- *SKIP* and *INSERT* for constructing their geometry work. For example,
4.3 Function Inversion and Backtracking

The reason XXoX used the Function Inversion and Backtracking is that it is very expensive and inefficient to store old objects in the Undo list especially for solid modeling application. That is, if we store all the old objects in the undo list for a solid modeling application, then it is very inefficient since we may need huge memory to perform the task. In such cases, it is much better to store the relative Inverse functions of commands in the undo list instead. For example,

<table>
<thead>
<tr>
<th>Commands</th>
<th>Inverse Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale(sx, sy, sz)</td>
<td>scale(1/sx, 1/sy, 1/sz)</td>
</tr>
<tr>
<td>translate(tx, ty, tz)</td>
<td>translate(-tx, -ty, -tz)</td>
</tr>
<tr>
<td>rotate(deg)</td>
<td>rotate(-deg)</td>
</tr>
<tr>
<td>create(object)</td>
<td>delete(object)</td>
</tr>
<tr>
<td>delete(object)</td>
<td>restore(object)</td>
</tr>
</tbody>
</table>

Table 1: Inverse Functions in XXoX

In XXoX, we store the deleted objects in a bounded array. Therefore, when deleting too many objects, we cannot undelete the objects which deleted early. Furthermore, when creating a new object by using the same name of an existing object — replacing an old object, we also need to store the old object. Thus, we can un-create the new object and restore the old object later.

The internal structure of Undo list and Redo list in XXoX are as follows:

**Undo list** in History Widget:

<table>
<thead>
<tr>
<th>Command #1</th>
<th>Inverse Function #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command #2</td>
<td>Inverse Function #2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Command #n</td>
<td>Inverse Function #n</td>
</tr>
</tbody>
</table>

Checking Point

**Redo list** in a string array:

<table>
<thead>
<tr>
<th>Command #n+k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command #n+k-1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Command #n+1</td>
</tr>
</tbody>
</table>

To apply the Redo command, XXoX needs to reparse the command which is on top of the Redo list.
4.4 Source - to - Source Preprocessor

After users apply the *Undo* commands, XXoX will delete the previous command on the top of *Redo list* which we discussed in the previous sections. Therefore, the History Widget omits all the undoing commands for efficiency, and save the net effect to the batch file for further use. Thus, XXoX is like a Source-to-Source Preprocessor for the CSG - *Constructive Solid Geometry* language. The outline is as follows:

For example,

\[
\begin{align*}
\text{a} &= \text{box}(0,0,0,2,2,2) \\
\text{b} &= \text{sphere}(1,1,2,2) \\
\text{a} &= \text{a} \land \text{b} \\
\text{undo} \\
\text{XXoX} \\
\text{a} &= \text{box}(0,0,0,2,2,2) \\
\text{b} &= \text{sphere}(1,1,2,2)
\end{align*}
\]
References


APPENDIX - A

XXoX Execution Arguments

The XXoX command recognizes several optional execution arguments which may be use to you. The format of the XXoX command is as follows:

```markdown
% xxox [ -noint ] [ -nocomd ] [ -nohead ]
[ -batch file ] [ -poly file ] [ -face file ]
[ -quit ] [ -server ]
[ -width width ] [ -height height ] [ -color color ]
```

where

- **noint** = Disable the interactive Menus, Buttons, Scales, and popup Dialogs.
- **nocomd** = Disable the interactive Command Input Area.
- **nohead** = Disable all the interactive widgets and title information.
- **batch file** = Retrieve and execute the batch file `file` when starting program.
- **poly file** = Generate the polygon file `file` automatically.
- **face file** = Generate the face file `file` automatically.
- **quit** = Quit XXoX automatically.
- **server** = Enable the communication server between XXoX and xellpack system.
- **width width** = Specify width of the drawing window to `width`.
- **height height** = Specify height of the drawing window to `height`.
- **color color** = Specify background color of the drawing window to `color`.
APPENDIX - B

Specifications of Widget Tree Nodes

B.1 Variables of Creating Menu Widget - in create_menu()

Now let's look at the create_menu() function. This function loops through the array of widget tree entries creating an appropriate widget for each entry. To ignore function item in the NODE structure can be done by assigned its value to NULL (or 0). Following we will describe the creating actions in create_menu() according with the types of node. The structure relations in widget tree are:

```plaintext
[all]:
  widget: widget pointer
  pattern: name of pixmap pattern in normal status
  insens: name of pixmap pattern in insensitive status
  accel: [0]-XmNmnemonic, [1]-XmNaccelerator

muButton:
  Class Name: XmPushButton
  title: XmNlabelString
  func & data: XmNactivateCallback

muLArrow:
muRArrow:
  Class Name: XmArrowButton
  title: XmNlabelString
  func & data: XmNactivateCallback

muToggle:
  Class Name: XmToggleButton
  title: XmNlabelString
  func: XmNValueChangedCallback
  data: [0]-XmNset, [1]-callback data, [2]-widget pointer

muLabel:
  Class Name: XmLabel
  title: XmNlabelString
  data: widget pointer
```
\textbf{muText:}  
Class Name: XmText  
title: XmNvalue  
data: widget pointer

\textbf{muSelect:}  
Class Name: XmSelectionBox  
title: XmNlistLabelString  
data: widget pointer

\textbf{muFile:}  
Class Name: XmFileSelectionBox  
title: XmNlistLabelString  
data: [0]-XmNmustMatch, [1]-XmNdirMask, [2]-widget pointer

\textbf{muRowColumn:}  
Class Name: XmRowColumn  
data: XmNnumColumns  
sub_: pointer to its subwidget tree array of type \texttt{NODE}

\textbf{muFrame:}  
Class Name: XmRowColumn - XmFrame  
data: XmNnumColumns  
sub_: pointer to its subwidget tree array of type \texttt{NODE}

\textbf{muMenu:}  
Class Name: XmCascadeButton  
title: XmNlabelString  
data: XmNmenuHelpWidget  
sub_: pointer to its submenu

\textbf{muMenuT:}  
Class Name: XmCascadeButton - XmToggleButton, XmNradioBehavior  
title: XmNlabelString  
sub_: pointer to its submenu

\textbf{muPullDown:}  
Class Name: XmRowColumn - MenuBar  
sub_: pointer to its submenu

\textbf{muRadio:}  
Class Name: XmRowColumn - RadioBox  
data: XmNnumColumns  
sub_: pointer to its subwidget of toggle buttons

\textbf{muOption:}  
Class Name: XmRowColumn - SimpleOptionMenu  
data: [0]-XmNbuttonCount, [1]-XmNbuttons, [2]-widget pointer
muScale:
Class Name: XmScale
title: XmNtitleString
func: XmNvalueChangedCallback
data: [0]-XmNminimum, [1]-XmNmaximum, [2]-XmNvalue,
[3]-callback data, [4]-widget pointer

muMatrix:
Class Name: XbaeMatrix
title: XmNtitleString
func: XmNvalueChangedCallback
data: [0]-matrix rows & visible rows, [1]-matrix columns,
[2]-visible columns, [3]-widget pointer

muSeparate:
Class Name: XmSeparator

muDraw:
Class Name: XmDrawingArea
data: [0]-XtNwidth, [1]-XtNheight

muScroll:
Class Name: XmScrolledWindow
data: [0]-pointer to vertical scrollbar,
[1]-callback function of vert., [2]-callback data,
[3]-pointer to horizontal scrollbar,
[4]-callback function of horiz., [5]-callback data,
sub_: pointer to its subwidget tree array of type NODE

muScrText:
Class Name: XmScrolledText
data: [0]-XmNrows, [1]-XmNcolumns, [2]-widget pointer

muCommand:
Class Name: XmCommand
name: XmNpromptString
func: XmNcommandEnteredCallback
data: XmNhistoryVisibleItemCount

muFormItem:
Class Name: XmForm
sub_: pointer to its subwidget tree array of type NODE

muForm:
Class Name: any class of subwidget
data: [0]-top, [1]-bottom, [2]-left, [3]-right attach.
value = -1: attach form, 0: none, >0: attach widget #

muVCustD:
Class Name: Unmanaged popup dialog
title: XmNlabelString
data: widget pointer
sub_: pointer to its subwidget of popup dialog
B.2 Variables of Creating Dialog Widget - in create_dialog()

In addition, the structure relations for popup dialogs in create_dialog() are:

\[ \text{dgRowColumn:} \]
\[
\begin{array}{ll}
\text{Class Name:} & \text{XmRowColumn} \\
\text{title:} & \text{XmNlabelString} \\
\text{func:} & \text{XmNactivateCallback} \\
\text{data:} & \text{widget pointer} \\
\text{sub_:} & \text{pointer to its subwidget of popup dialog}
\end{array}
\]

\[ \text{dgFrame:} \]
\[
\begin{array}{ll}
\text{Class Name:} & \text{XmRowColumn - XmFrame} \\
\text{data:} & \text{XmNnumColumns} \\
\text{sub_:} & \text{pointer to its subwidget tree array of type NODE}
\end{array}
\]

\[ \text{dgOK:} \]
\[
\begin{array}{l}
\text{func:} \\
\text{XmNokCallback}
\end{array}
\]

\[ \text{dgCANCEL:} \]
\[
\begin{array}{l}
\text{func:} \\
\text{XmNcancelCallback}
\end{array}
\]

\[ \text{dgHELP:} \]
\[
\begin{array}{l}
\text{func:} \\
\text{XmNactivateCallback - XmNhelpCallback}
\end{array}
\]
APPENDIX - C

Scanner & Parser for Command Area Interface

To implement the Command Area Interface including scanner and parser, first we need to redirect the content of the Motif Command widget to the scanner which is generated by lex, or flex. Then define the grammar of the XXoX application CSG (Constructive Solid Geometry) commands for the parser which is generated by yacc, or bison. Following are the definitions of those Keywords, Identifiers, File Names, Numbers, Data Sets, Strings, and Grammars.

C.1 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

<table>
<thead>
<tr>
<th>ABAQUS</th>
<th>ANSYS</th>
<th>arc</th>
<th>BAR</th>
<th>blood</th>
<th>load</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>box</td>
<td>bsave</td>
<td>center</td>
<td>circle</td>
<td></td>
</tr>
<tr>
<td>clear</td>
<td>color</td>
<td>cone</td>
<td>conform</td>
<td>cyan</td>
<td></td>
</tr>
<tr>
<td>cylinder</td>
<td>delete</td>
<td>disc</td>
<td>eyedeg0</td>
<td>eyedeg1</td>
<td></td>
</tr>
<tr>
<td>eyexyz0</td>
<td>eyexyz1</td>
<td>gload</td>
<td>green</td>
<td>gsave</td>
<td></td>
</tr>
<tr>
<td>HEX</td>
<td>IGES</td>
<td>ISOPARAMETRIC</td>
<td>LENGTH</td>
<td>lightdeg</td>
<td></td>
</tr>
<tr>
<td>lightxyz</td>
<td>line</td>
<td>magenta</td>
<td>mesh</td>
<td>NASTRAN</td>
<td></td>
</tr>
<tr>
<td>neutral</td>
<td>no</td>
<td>NUMBER</td>
<td>orange</td>
<td>patran</td>
<td></td>
</tr>
<tr>
<td>pink</td>
<td>pload</td>
<td>point</td>
<td>psave</td>
<td>QUAD</td>
<td></td>
</tr>
<tr>
<td>quit</td>
<td>red</td>
<td>redo</td>
<td>rotate</td>
<td>scale</td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>sheet</td>
<td>shell</td>
<td>skin</td>
<td>skip</td>
<td></td>
</tr>
<tr>
<td>solid</td>
<td>sphere</td>
<td>sweep1</td>
<td>sweep2</td>
<td>tan</td>
<td></td>
</tr>
<tr>
<td>translate</td>
<td>TRI</td>
<td>unconf orm</td>
<td>undo</td>
<td>unzoom</td>
<td></td>
</tr>
<tr>
<td>WEDGE</td>
<td>white</td>
<td>XOX_C2</td>
<td>XOX_Patches</td>
<td>xpm</td>
<td></td>
</tr>
<tr>
<td>yellow</td>
<td>yes</td>
<td>zoom</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 34 -
C.2 Identifiers, File names, Numbers, Data sets, and Strings

Their relative lexical rules are as follows:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>{letter}{letter}{digit}*</td>
</tr>
<tr>
<td>fid</td>
<td>{letter}{letter}{digit}*</td>
</tr>
<tr>
<td>num</td>
<td>{numreal}{decint}{octint}{hexint}</td>
</tr>
<tr>
<td>data</td>
<td>({simnum}</td>
</tr>
<tr>
<td>string</td>
<td>(''</td>
</tr>
<tr>
<td>letter</td>
<td>[A-Za-z_]</td>
</tr>
<tr>
<td>fletter</td>
<td>[A-Za-z_/~]</td>
</tr>
<tr>
<td>digit</td>
<td>[0-9]</td>
</tr>
<tr>
<td>nzdigit</td>
<td>[1-9]</td>
</tr>
<tr>
<td>otdigit</td>
<td>[0-7]</td>
</tr>
<tr>
<td>hxdigit</td>
<td>[0-9a-fA-F]</td>
</tr>
<tr>
<td>digits</td>
<td>{digit}+</td>
</tr>
<tr>
<td>exponent</td>
<td>[Ee][+-]?{digits}</td>
</tr>
<tr>
<td>real1</td>
<td>{digits}(.</td>
</tr>
<tr>
<td>real2</td>
<td>(.</td>
</tr>
<tr>
<td>real3</td>
<td>{digits}({exponent})?</td>
</tr>
<tr>
<td>numreal</td>
<td>(-)?({real1}</td>
</tr>
<tr>
<td>decint</td>
<td>(-)?{nzdigit}{digit}*</td>
</tr>
<tr>
<td>octint</td>
<td>(0){otdigit}*</td>
</tr>
<tr>
<td>hexint</td>
<td>(0)[xX]{hxdigit}</td>
</tr>
<tr>
<td>simnum</td>
<td>({numreal}</td>
</tr>
</tbody>
</table>
C.3 Grammars

Rules of parser which created by yacc or bison are as follows:

- **statements**
  - `uncond_stmt`
  - `cond_stmt`
  - `bload (file_id)`
    - Load batch file `file_id` to internal command history and execute.
  - `bsave (id , file_id)`
    - Select object `id` for the active target and save internal command history to batch file `file_id`.
  - `undo`
    - Undo the previous command.
  - `redo`
    - Redo the previous Undo command which is on the top of redo list.
  - `skip`
    - Skip the previous Undo command which is on the top of redo list.

- **uncond_stmt**
  - `edit_stmt`
  - `control_stmt`
  - `transform_stmt`
  - `patran_stmt`
  - `neutral_stmt`
  - `quit`
    - Quit the Solid Modeling program.
  - `Empty statement.`

- **edit_stmt**
  - `gsave (id , file_id)`
    - Save object `id` to XOX format file `file_id`.
  - `psave (id , file_id)`
    - Save object `id` to Polygon format file `file_id`.
  - `xpm (file_id)`
    - Save drawing window to XPM format file `file_id`.
  - `select (id)`
    - Select active object `id`.
  - `delete (id)`
    - Delete object `id`.
  - `clear`
    - Delete all objects.

- **point**
  - `num1 , num2 , num3`
colors:
- red
- green
- blue
- cyan
- magenta
- yellow
- orange
- pink
- white
- tan

color (colors)
Set default color associated with objects to colors.

class
: view_stmt
| conform
To automatically scale the viewbox associated with a graphics window to include all of its currently associated objects entities.

| unconform
Turn off the automatically conformation.

eyexyz (point)
Set a three Cartesian coordinate point that define the end point of the eye vector for a window.

eyedeg (point)
Set a three Spherical coordinate point that define the end point of the eye vector for a window.

eyexyz1 (point)
Set a three Cartesian coordinate point that define the begin point of the eye vector for a window.

| eyedeg1 (point)
Set a three Spherical coordinate point that define the begin point of the eye vector for a window.

lightxyz (point)
Set the direction of the three Cartesian coordinate point light source for a window.

| lightdeg (point)
Set the direction of the three Spherical coordinate point light source for a window.

| zoom (num1, num2, num3, num4)
Zoom on an unitless rectangular sub-region (num1, num2) - (num3, num4) of a window.

| unzoom
Reset all panning & zooming have taken place.

| center (num1, num2)
Pan to unitless location (num1, num2) in window.
transform_stmt  : scale (id, point)
    Scale object id by three scale factors point.

translate (id, point)
    Translate object id by a specified vector point.

rotate (id, point1, point2, num)
    Rotate object id about a user specified axis with beginning point point1, ending point point2, and rotate angle num in degree.

yes_no  : yes
    | no

patran_file : file_id
    |

patran_stmt  : patran (yes_no1, yes_no2, yes_no3 patran_file)
    Generate the corresponding PATRAN language to session file patran_file (default is xxx.xxx).
    Where yes_no1: generate mesh, yes_no2: output neutral file, yes_no3: run PATRAN.

phaseI  : no
    | yes, no
    | yes, yes

neutral_typ : neutral
    |
    | NASTRAN
    | ANSYS
    | ABAQUS
    | IGES

neutral_stmt  : neutral (string, phaseI, yes_no neutral_typ)
    Output Neutral file with title string of type neutral_typ (default is neutral). Where phaseI: include Phase-I data [and include GFEG/CFEG Table], yes_no: include Phase-0 data.

cond_stmt  : mesh_stmt
    | asgn_stmt

celmnt_typ : TRI
    |
    | QUAD
    | WEDGE
    | HEX
    | BAR
mesh_opt : ISOPARAMETRIC
| NUMBER
| LENGTH

datas : num
| data

obj_nme : , id

mesh_stmt : mesh( elmt_typ , mesh_opt , datas obj_nme )
  Generate Finite Element Mesh on object obj_nme
  or all objects with element type elmt_typ, option
  mesh_opt, and its relative data datas.

asgn_stmt : id = exp
  Assign object exp to an identifier id.

exp : bool_exp
| primary_exp

bool_exp : exp & primary_exp
  Find the intersection of two objects exp and
  primary_exp.
| exp | primary_exp
  Find the union of two objects exp and
  primary_exp.
| exp - primary_exp
  Find the difference of two objects exp and
  primary_exp.

primary_exp : id
| ( exp )
| transform_exp
| create_exp
| sweep_exp
| skin_exp
| file_exp
transform_exp : scale ( exp , point )
                Scale object exp by three scale factors point.

translate ( exp , point )
Translate object exp by a specified vector point.

rotate ( exp , point1 , point2 , num )
Rotate object exp about a user specified axis with
beginning point point1, ending point point2, and
rotate angle num in degree.

solid : , solid
| , shell
|

create_exp : point ( point )
Create a point object at point.

| line ( point1 , point2 )
Create a line object from point1 to point2.

| arc ( point , num1 , num2 , num3 )
Create an arc object center at point with radius
num1, beginning angle num2, and ending angle
num3.

| circle ( point , num )
Create a circle object center at point with radius
num.

| sheet ( point , num1 , num2 )
Create a sheet object corner at point with length
num1 and width num2.

| disc ( point , num )
Create a disc object center at point with radius
num.

| box ( point , num1 , num2 , num3 solid )
Create a box object corner at point with length
num1, width num2, and height num3 of shell type
solid (default is solid).

| sphere ( point , num solid )
Create a sphere object center at point with radius
num of shell type solid (default is solid).

| cylinder ( point , num1 , num2 solid )
Create a capped cylinder object center at point
with radius num1 and height num2 of shell type
solid (default is solid).

| cone ( point , num1 , num2 solid )
Create a capped cone object center at point with
radius num1 and height num2 of shell type solid
(default is solid).
sweep_exp

: sweep1 (exp1, exp2)
  Sweep an swept-object exp1 along another path-object exp2.
  ! sweep2 (exp1, exp2, point)
  Sweep an swept-object exp1 along another path-object exp2 while maintaining the relative position point of the swept-object with respect to the tangent space of the path-object.

shape

: XOX_C2
  ! XOX_Patches

skin_obj

: id, point

skin_lst

: skin_obj
  ! skin_lst, skin_obj

skin_exp

: skin (num1, num2, shape1, shape2, skin_lst)
  Create a surface that fits through given 1-dimensional objects skin_lst with degrees num1 & num2 and shape options shape1 & shape2.

file_exp

: gload (file_id)
  Load object from XOX format file file_id.
  ! pload (file_id)
  Load object from Polygon format file file_id.
APPENDIX - D

Examples

D.1 Example 1 - Engine

Batch File:

```plaintext
unconform
eyexyzl(40, 80, 10)
color(tan)
shaft = translate(rotate(cylinder(0, 0, 0, 0, 0, 5, 5.4), 0, 0, 0, 1, 0, -90), 2.6, 0, 13.36)
rod = cylinder(0, 0, -0.52, 1.56, 1.04) - cylinder(0, 0, -0.52, 1.04, 1.04)
rotate(rod, 0, 0, 0, 1, 0, -90)
rod = translate(rod, 0, 0, 2.6) | box(-0.26, -0.52, 3.64, 0.52, 1.04, 9.2)
ring = cylinder(0, 0, -0.52, 0.91, 1.04) - cylinder(0, 0, -0.52, 0.52, 1.04)
rod = shaft | rod | translate(rotate(ring, 0, 0, 0, 1, 0, -90), 0, 0, 13.36)
delete(shaft)
delete(ring)
rod = rod & cylinder(0, 0, -2.6, 18)
conform

unconform
eyexyzl(50, 10, -80)
color(green)
pis = cylinder(0, 0, 10.36, 2.6, 4.68) - box(-1.04, -3, 10.36, 2.08, 6.156)
pis = pis - cylinder(0, 0, 9.36, 2.4.92)
pis = pis - translate(rotate(cylinder(0, 0, -4, 0.52, 8), 0, 0, 0, 0, 1, 0, -90), 0, 0, 13.36)
zoom(0.3, 0.6, 0.7, 0.1)
conform
```

- 42 -
D.2  Example2 - Vase

Batch File:

```plaintext
color(red)
a1 = circle(0,0,0,3,4)
a2 = circle(0,0,0,5)
a3 = circle(0,0,0,3,3)
a4 = circle(0,0,0,6,3,5)
color(green)
a5 = skin(3,3,XOX_C2,XOX_Patches,a1,4,0,-3,a2,5,0,0,a3,3,0,3,a4,3,5,0,6)
delete(a1)
delete(a2)
delete(a3)
delete(a4)
```

```plaintext
unconforn
eyexyzl(40,80,40)
color(red)
crank = cone(0,0,0,6.16,6.16) & cylinder(0,0,0,6.16,2)
shaft = cone(0,0,2.2,5.4,6) & cylinder(0,0,2.2,5.4,3)
crank = crank | shaft | cylinder(0,0,5.6,0,6,3)
delete(shaft)
crank = crank | rotate(crank,0,0,0,0,0,1,0,0,0,180) | cylinder(2.6,0,-1,1.04,2)
rotate(crank,0,0,0,0,0,0,90)
translate(crank,-0.046,0,0)
zoom(0,0,1,1.5)
conform

unconforn
color(orange)
cap = cylinder(0,0,0,5,5,12)
cap = cap - cylinder(0,0,0,4,2.6,11) - cylinder(0,0,0,1.4,1.4) - box(-5,0,4,10,5,14)
ly = box(-5.06,-8.06,12.12,8.06,0.2) - box(-5.06,-8.06,10,16.12,6.06,0.2)
cy = cylinder(6.06,-5.06,10,2,0.2)
ly = ly | cy | rotate(cy,0,0,0,0,0,0,1,0,-90)
delete(cy)
cap2 = ly | translate(ly,0,0,2) | translate(ly,0,0,4) | translate(ly,0,0,6)
delete(ly)
cap2 = cap2 - cylinder(0,0,0,5,5,12)
conform
```
D.3 Example 3 - Chain

Batch File:

eyexyz1(50, 100, 100)
color(red)
path = circle(0, 0, 0.5)
color(green)
single = rotate(disc(5, 0, 0, 1), 0, 0, 0, 1, 0, -90)
color(cyan)
torus = sweep2(single, path, 5, 0, 0)
delete(path)
delete(single)
t2 = translate(rotate(torus, 0, 0, 1, 0, 0, -90), 8, 0, 0)
t3 = translate(rotate(torus, 0, 0, 0, 1, 0, -90), 0, 8, 0)
t4 = translate(torus, 16, 0, 0)
t5 = translate(t2, 16, 0, 0)
t6 = translate(t4, 16, 0, 0)
t7 = translate(t5, 16, 0, 0)
t8 = translate(t6, 16, 0, 0)