UAH RESEARCH REPORT NO. 803

APPLICATIONS OF ARTIFICIAL INTELLIGENCE TO SPACE STATION AND AUTOMATED SOFTWARE TECHNIQUES

High Level Robot Command Language

Final Report for the Period
June 1, 1988 - May 31, 1989

NAS8-37813

Prepared by:

James W. McKee
Johnson Research Center
The University of Alabama in Huntsville
Huntsville, Alabama 35899

Prepared for:

Walt Mitchell
System Software Branch
Information and Electronic Systems Lab

Marshall Space Flight Center
National Aeronautics and
Space Administration
Marshall Space Flight Center, AL 35812

July 1989

TABLE OF CONTENTS

High Level Robot Command Language

1 Report Overview

This report is the final report for the High Level Robot Command Language project. This report reviews the progress made on the project since the first bi-annual report. As such, the next two sections are similar to the corresponding sections in the first report. The section on the research approach contains an update of the status of the various tasks in the project. The appendices contain the listing of the new software developed during this period. This report and the previous report should be used together to gain an insight into the software being developed in this project.

2 Project Abstract

The objective of this project is to develop a "system" that will allow a person not necessarily skilled in the art of programming robots to quickly and naturally create the necessary data and commands to enable a robot to perform a desired task.

The system will use a menu driven graphical user interface. This interface will allow the user to input data and to select objects to be moved. There will be an imbedded expert system to process the knowledge about objects and the robot to determine how they are to be moved. There will be automatic path planning to avoid obstacles in the work space and to create a near optimum path. The system will contain the software to generate the required robot instructions.

3 Research goals

The ability of a human to take control of a robotic system in order to handle unforeseen changes in the robot's work environment or scheduled tasks is essential in any use of robots in space. But in cases in which the work environment is known, a human controlling a robot's every move by tele-robotics is both time consuming and frustrating to the human (especially if there is a time delay in the loop).

A system is needed in which the user can give the robotic system commands to perform tasks but need not tell the system how to perform the tasks. To be useful, this system should to be able to plan and perform the tasks faster than the task could be performed by a telerobotic system. The interface between the user and the robot system must be natural and meaningful to the user.

In this project, a set of programs that will allow an unskilled user to program a robot by way of a natural graphical computer interface will be developed. The user will selects objects to be manipulated by selecting representations of the objects on a 2-D projection of a 3-D model of the robot's work environment. The user may move in the work environment by changing both the viewpoint and magnification of the 2-D projection.

The system will use an expert system and path planning programs to transform user selection of items to be moved into commands for the robot. The system will first determines if the desired task is possible given the abilities of the robot and any constraints on the movement of the object. If the task is possible, the system will determines what movements the robot needs to make to perform the task. The movements will then be transformed into commands for the robot. The information defining the robot, the work environment, and how objects may be moved is stored in a data base accessible to the system and displayable to the user.

4 Research approach

The project has been divided in to eleven major tasks that will require at least four years to complete. These tasks can be grouped into four logical groups: user interface, path planning, sensor input, and robot interface and control. The user interface group of tasks consist of four tasks: object definition interface task, object selection interface task, data base, and expert system. The path planning group of tasks consists of two major tasks: geometric path planning and dynamic path planning. The sensor input task consist of one task: robot calibration to work space. The robot interface and control group of tasks consists of four tasks: protocol task, robot motion simulation task, system-robot control task, and system-telerobotic control task.

The following is a more detailed description of each task and the task's present status. Figure 1 is a data flow block diagram for the project. The figure shows the inter relation between the various software tasks.

User interface group:

1. Object definition interface task -- This will be a menu driven program that will allow a user to create graphical descriptions of the robot's work space and the robot. The program will also question the user about the physical attributes of objects and the restrictions on how objects can be moved.

Status: Code has been written to allow the user to select from a menu of primitive objects and place the selected 3-D objects in 3-D space. Work needs to be done on moving the objects and joining two or more objects to form a new object. The code developed for this portion is included in an appendix.

2. Object selection interface task -- This will be a menu driven program that will allow the user to select objects to be moved and to select where the objects are to be placed.

Status: This task will much of the code being developed in the object definition interface.

3. Data base task -- The data base program will store and retrieve the graphical representations of objects. It will also store the knowledge about objects. There will be set of interface functions that will allow other programs to store and retrieve needed data without the need to know the structure of the data base.

Status: The data base has been coded and tested. What we have works, but it will probably need to be expanded to store the expert system rules and facts about objects and the kinematics of the robot. The code for the data base is included in the Appendix.

4. Expert system task -- After the user selects an object and where the object is to be placed, the expert system will examine the facts about the object and its movement to determine if it is possible to move the object, and if so, how to move the object. The expert system will create a list of constraints on the motion and a list of data points. The path planning programs will use this data to plan a good path for the robot.

Status: CLIPS has been selected as expert system shell. CLIPS has been ported to Silicon Graphics computer. The listing of the required facts and rules has been started.

Path planning group:

5. Geometric path planning task -- The geometric path planning program will determine where in the robot's work space the robot may move with the object. It will use the description of the robot's environment to calculate the "free space" for the robot.

Status: Literature search started.

6. Dynamic path planning task -- The dynamic path planning program will use the constraints on the motion of the object and constraints on the motion of the robots joints to plan a "good" path in the robot's free space .

Status: The dynamic path planner will be developed in conjunction with the geometric path planner. Therefore, both rely on the same literature search.

Sensor input group:

7. Robot calibration task -- The system must be able to calibrate the robot to the work environment. The calibration task will use a television camera attached to the end-effector of the robot to acquire images of fiducial points on the work environment. The program will then calculate the relation between the robot and the work environment.

Status: Literature search started.

Robot interface and control group:

8. Protocol task -- At the present time there is no standard robot interface protocol. This task is to develop an efficient interface between a multi-tasking operating system and the PUMA 562 robot. The PUMA uses the DDCMP protocol.

Status: Complete, documentation written. Source code listing of protocol, make files, and utility programs included in previous report.

9. Robot motion simulation task -- Users may wish to "see" what the robot is going to do before actually having the robot move. This program will allow the user to view a simulation of the robots motion in moving the selected object. The user may change his point of view and zoom in or out.

Status: Wire frame robot simulation working.

10. System-robot control task -- This program converts the system position points into actual robot commands. Since there is no universal robot command language, this program will be unique to each type of robot. The PUMA 562 uses VAL II.

Status: Code was written to interface the tele-robotic interface to the robot.

11. System-telerobotic control task -- The user may wish to take direct control of the robot. This program will create an interface on the computer that will allow the user to set the speed and other limits on the movement of the robot. Then the user could move the robot by using a mouse. The user would use video feedback to determine the relation between the robot and the work space.

Status: Two programs are being written: first moves robot in robot joint space, second moves robot in world co-ordinate space.

5 Anticipated results

This project will create a complete high level robotic programming system in which the user will "program" the robot by simple selections on a graphical display. This project has been divided into tasks with well defined interfaces. In each task, the programs may be modified, changed, or replace without effecting the operation of the other tasks. This will create an environment in which research may be performed in specific areas and the results evaluated in a total system.

6 Publications and presentations resulting from this work

Publications

"A Graphical, Rule Based Robotic Interface System," To Appear In: Fourth Conference on Artificial Intelligence for Space Applications (AISA), James W. McKee and John Wolfsberger, November 15-16, 1988, Huntsville, Alabama.

"High Level Intelligent Control of Telerobotic Systems," To Appear In: Conference on Automation and Robotics for Space and Military Applications, James W. McKee and John Wolfsberger, June 21-23, 1988, Huntsville.

Presentations

"A Graphical, Rule Based Robotic Interface System," presented at the Fourth Conference on Artificial Intelligence for Space Applications (AISA), November 1988, Huntsville, Alabama.

"High Level Intelligent Control of Telerobotic Systems," Conference on Automation and Robotics for Space and Military Applications, June 1988, Huntsville Alabama.

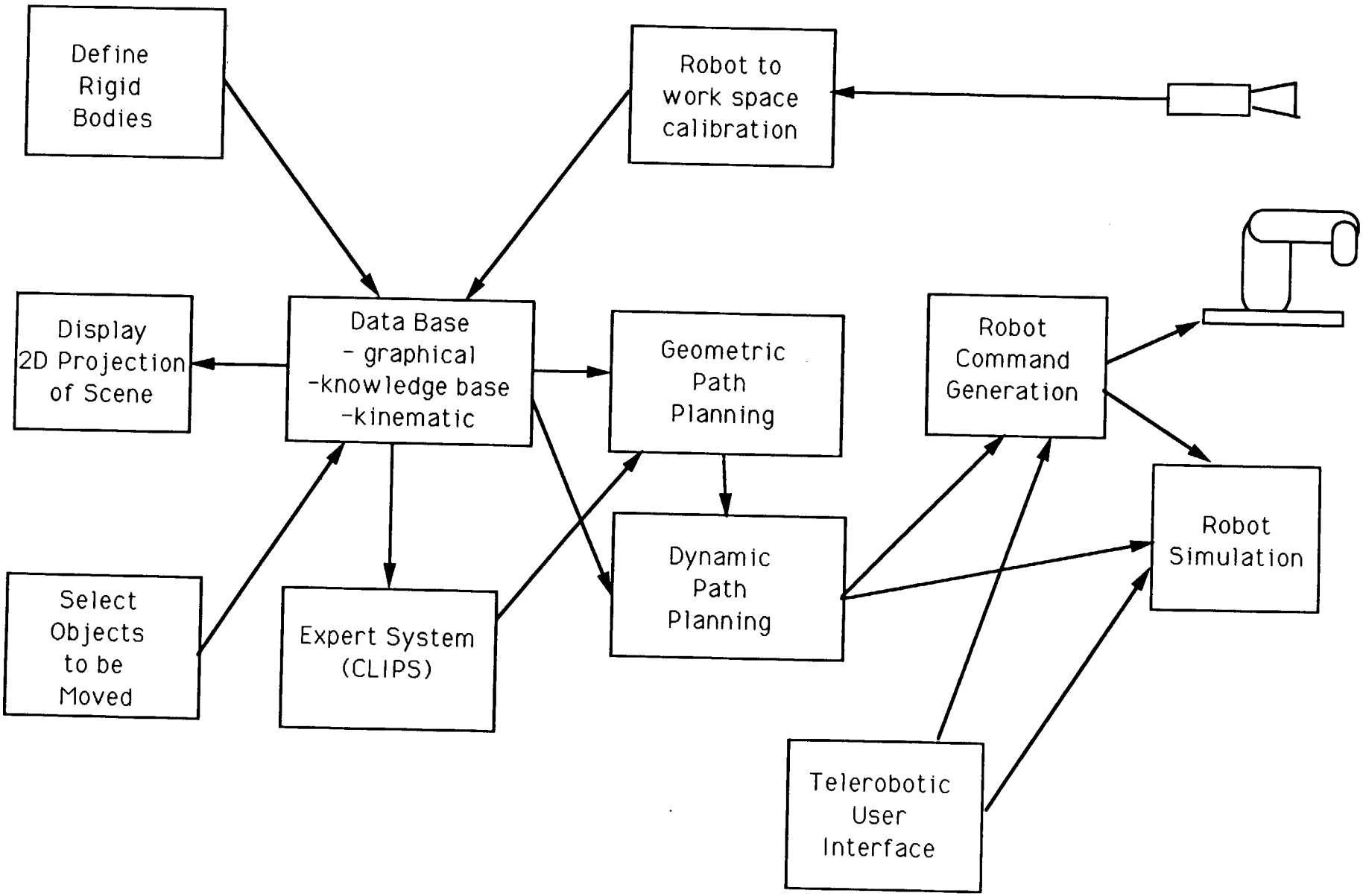

Figure 1 Data Flow Block Diagram

Appendix A User Interface Software

Appendix A.1 Include files

```
/*************************************************************************
  Filename:  defs.h

  By:  Tim Thompson

  Purpose:  This file contains many of the definitions and declarations
            needed for other modules of the program.
*************************************************************************/

#include "gl.h"
#include "stdio.h"
#include "device.h"
#include "math.h"

/*  used by generator.c     */
#define span 64
#define MaxPolys 128

#define red 8
#define green 72
#define yellow 136
#define blue 200
#define magenta 264
#define cyan 328
#define white 392

/*  used in windows.c       */
#define WholeScreen 0
#define FrontView   1
#define SideView    2
#define TopView     3
#define TextWindow  4
#define Windows     5

/*  used in
popmenus.c      */
#define POPCOLOR1 512
#define POPCOLOR2 1024
#define POPCOLOR3 1536
#define POPCOLOR4 2048
#define POPCOLOR5 2560
#define ENDCOLOR 3072
#define MASKVALUE POPCOLOR1 | POPCOLOR2 | POPCOLOR3 | POPCOLOR4 | POPCOLOR5


#define POPUPBACKGROUND POPCOLOR1
#define POPUPTEXT       POPCOLOR2
```

```
#define POPUPHIGHLIGHT  POPCOLOR3
#define POPUPACTIVE     POPCOLOR4
#define POPUPSHADOW     POPCOLOR5

#define CROSSCOLOR POPCOLOR2

/*  used by gprimitives.c and vectors.c   */
#define PI M_PI

/*  used by vectors.c     */
#define VectStackSize MATRIXSTACKDEPTH
#define convert 1.7453293e-3

/*  used by vectors.c     */
typedef struct {
    float i;
    float j;
    float k;
} vector;

/*  used by popmenus.c    */
struct popupentry {
    short type;
    char *text;
    Boolean flag;
};

struct menutype {
    int x;
    int y;
    char *title;
    struct popupentry *list;
};

struct menulist {
    struct menutype *menu;
    struct menulist *next;
    struct menulist *last;
};
```

```
/*************************************************************************
  Filename:  kindefs.h

  By:  Tim Thompson

  Purpose:  This file defines the structures used to represent kinematic
            objects and transformations.

            An object is (currently) made up of the following parts:

              1.  A name.
              2.  A flag indicating if the object has been modified or not.

              3.  A type. ('u'=undefined, 'o'=object, 'r'=rigid body)
              4.  A sub-component:
                    a.  object
                  OR
                    b.  rigid body.
              5.  A scaling factor for all sub-components.
              6.  A link to the next object.
              7.  A link to parent object.


            A transformation is made up of the following parts:

              1.  A type. ('t'=translation, 'r'=rotation, 'u'=undefined)
              2.  An axis. ('x', 'y', 'z', or 'a' for all axes)
              3.  An amount:
                    a.  rotation angle
                  OR
                    b.  translation distance
                  OR
                    c.  rotation angles (around all three axes)
                  OR
                    d.  translation distances (along all three axes)
              4.  A link to the next transform for the object.

*************************************************************************/

struct xform {
    char type, axis;
    union {
      short angle;
      float dist;
      struct {
        short x;
        short y;
        short z;
      } rot;
      struct {
        float x;
        float y;
```

```
            float z;
          } trans;
        } amt;
        struct xform *next;
};

struct kinobject {
        char name[20];
/*      int modified; */    /* Treated as Boolean */
        char type;
        union {
          OBJECT *rbody;
          struct kinobject *subobj;
        } obtype;
        float scale;
        struct xform *xform;
        struct kinobject *nextkobj;
/*      struct kinobject *parent; */
};

typedef struct xform XFORM;
typedef struct kinobject KOBJ;

KOBJ *NewKObj ();
XFORM *NewXform ();
```

```
/*************************************************************************
  Filename:  obj.h

  Written by:   Allan Rideout
  Modified by:  Timothy A. Thompson

  Purpose:  This file contains all the structures used to implement the
            winged edge database.

            An object is defined in terms of a list of faces.
            A face is defined in terms of a list of its bounding edges.
               A face also has a pointer to an attribute structure.  This
               attribute structure contains thej color of the face, a
               vector normal to the face, and an integer flag which
               is reserved for future use.
            A bounding edge is defined simply as an edge.
            An edge is defined as two vertices.  An edge also contains
               pointers back to the two faces of which is is the
               intersection.
            A vertex is defined by its x, y, and z coordinates (Local
               coordinate system).  A vertex also contains pointer to
               a list of all its incident edges.

*************************************************************************/
/* obj.h 01.11.89*/

#include "string.h"

struct  face{ struct face  *nextfce;
              struct bedge *bedg;
              struct attribute *attr;
            };
struct bedge{ struct bedge *nextbedg;
              struct edge  *edg;
            };
struct edge { struct face  *fce1,*fce2;
              struct vertex *vtx1,*vtx2;
            };
struct vertex{ float x,y,z;
               struct iedge *iedg;
             };
struct iedge{ struct iedge *nextiedg;
              struct edge  *edg;
            };
struct corner{ struct corner *nextcorn;
               struct vertex *vtx;
             };
struct object{ int name;
               struct object *nextobj;
               struct face   *fce;
               struct corner *corn,*rcorn;
             };
```

```
struct attribute { int colr;
                   vector norm;
                   int flags;
                 };

typedef struct face FACE;
typedef struct bedge BEDGE;
typedef struct edge EDGE;
typedef struct vertex VERTEX;
typedef struct iedge IEDGE;
typedef struct corner CORNER;
typedef struct object OBJECT;
typedef struct attribute ATTRIBUTE;
```

```
/*************************************************************************
  Filename:  dbdefs.h

  By:  Tim Thompson

  Purpose:  This file contains definitions which are needed so that a
            module can use the routines in "interface.c" which are used
            to interact with the rigid body (winged edge) database.
            It also contains certain other definitions needed by modules
            using the database.

*************************************************************************/

#include "obj.h"

OBJECT *NewRb ();
FACE *FirstFace (), *NextFace (), *SameFace ();
CORNER *NewCorn ();
VERTEX *NewVertex (), *GetVert ();
ATTRIBUTE *NewAttribute ();

extern FACE *fce;
extern BEDGE *bedg;
extern EDGE *edg;
extern VERTEX *vtx;
extern IEDGE *iedg;
extern CORNER *corn;
extern OBJECT *obj;
extern ATTRIBUTE *attr;

#define sfce  sizeof(FACE)
#define sbedg sizeof(BEDGE)
#define sedg  sizeof(EDGE)
#define svtx  sizeof(VERTEX)
#define siedg sizeof(IEDGE)
#define scorn sizeof(CORNER)
#define sobj  sizeof(OBJECT)
#define sattr sizeof(ATTRIBUTE)
```

```
/************************************/
/*  menudefs.h                      */
/************************************/

#define BLANK -1
#define DONE 100

#define XAXIS 1
#define YAXIS 2
#define ZAXIS 3
#define ALLTHREE 4
#define SAME 5
#define SELECT 6
#define CSHELL 7
#define DUMBCHOICE 8

#define SIM 1
#define ODUI 2
#define EXITPROGRAM 3

#define WORLD 1
#define ADD 2
#define DELETE 3
#define SAVESCENE 4
#define LOADSCENE 5
#define HACK 6

#define CYLINDER 1
#define PIPE 2
#define SPHERE 3
#define CONE 4
#define PARALLELEPIPED 5
#define BOX 6
#define CUBE 7

#define ROTATE 1
#define TRANSLATE 2
#define ZOOM 3
#define CLIP 4

#define DEFAULTCOLOR 1
#define SETDEFCOLOR 2
#define REDCOLOR red
#define GREENCOLOR green
#define YELLOWCOLOR yellow
#define BLUECOLOR blue
#define MAGENTACOLOR magenta
#define CYANCOLOR cyan
#define WHITECOLOR white

struct popupentry superselection[] = {
  {SIM, "Simulate", TRUE},
```

```
    {ODUI, "RunODUI", TRUE},
    {EXITPROGRAM, "Exit Program", TRUE},
    {0, 0, TRUE}
  };

  struct popupentry menu1select[] = {
    {WORLD, "Change World View", TRUE},
    {ADD, "Add an Object", TRUE},
    {DELETE, "Delete an Object", FALSE},
    {SAVESCENE, "Save Scene", TRUE},
    {LOADSCENE, "Load Scene", TRUE},
    {HACK, "Hack it", TRUE},
    {BLANK, " ", TRUE},
    {CSHELL, "System (C-Shell)", TRUE},
    {BLANK, " ", TRUE},
    {DONE, "Previous Menu", TRUE},
    {0, 0, TRUE}
  };

  struct popupentry chgworldselect[] = {
    {ROTATE, "Rotate View", FALSE},
    {TRANSLATE, "Translate", FALSE},
    {ZOOM, "Zoom / Unzoom", FALSE},
    {CLIP, "Set Clipping Planes", TRUE},
    {BLANK, " ", TRUE},
    {DONE, "Previous Menu", TRUE},
    {0, 0, TRUE}
  };

  struct popupentry addobjselect[] = {
    {CYLINDER, "Cylinder", TRUE},
    {PIPE, "Pipe", TRUE},
    {SPHERE, "Sphere", TRUE},
    {CONE, "Cone", TRUE},
    {CUBE, "Cube", TRUE},
    {BOX, "Box", TRUE},
    {PARALLELEPIPED, "Parallelepiped", TRUE},
    {BLANK, " ", TRUE},
    {DONE, "Previous Menu", TRUE},
    {0, 0, TRUE}
  };

  struct popupentry colorselect[] = {
    {DEFAULTCOLOR, "Default Color", TRUE},
    {BLANK, " ", TRUE},
    {REDCOLOR, "Red", TRUE},
    {GREENCOLOR, "Green", TRUE},
    {YELLOWCOLOR, "Yellow", TRUE},
    {BLUECOLOR, "Blue", TRUE},
    {MAGENTACOLOR, "Magenta", TRUE},
    {CYANCOLOR, "Cyan", TRUE},
    {WHITECOLOR, "White", TRUE},
```

```
    {BLANK, " ", TRUE},
    {SETDEFCOLOR, "Set Default Color", TRUE},
    {BLANK, " ", TRUE},
    {DONE, "Previous Menu", TRUE},
    {0, 0, TRUE}
  };

  struct popupentry axisselection[] = {
    {XAXIS, "About X Axis", TRUE},
    {YAXIS, "About Y Axis", TRUE},
    {ZAXIS, "About Z Axis", TRUE},
    {ALLTHREE, "About all Axes", TRUE},
    {SAME, "No change", TRUE},
    {BLANK, " ", TRUE},
    {SELECT, "Select Point", TRUE},
    {CSHELL, "System (C-Shell)", TRUE},
    {BLANK, " ", TRUE},
    {DUMBCHOICE, "Next Menu", TRUE},
    {DONE, "Previous Menu", TRUE},
    {0, 0, TRUE}
  };

  struct popupentry dummychoices[] = {
    {1, "Go To Next Menu", TRUE},
    {BLANK, " ", TRUE},
    {2, "Choice 2", TRUE},
    {3, "Choice 3", TRUE},
    {4, "Choice 4", TRUE},
    {5, "Choice 5", TRUE},
    {6, "Choice 6", TRUE},
    {7, "Choice 7", TRUE},
    {8, "Choice 8", TRUE},
    {9, "Choice 9", TRUE},
    {10, "Choice 10", TRUE},
    {0, 0, TRUE}
  };

  struct popupentry dummychoices10[] = {
    {1, "Go To Next Menu", FALSE},
    {BLANK, " ", TRUE},
    {2, "Choice 2", TRUE},
    {3, "Choice 3", TRUE},
    {4, "Choice 4", TRUE},
    {5, "Choice 5", TRUE},
    {6, "Choice 6", TRUE},
    {7, "Choice 7", TRUE},
    {8, "Choice 8", TRUE},
    {9, "Choice 9", TRUE},
    {10, "Choice 10", TRUE},
    {0, 0, TRUE}
  };
```

Appendix A.2 Object Definition User Interface

```
/*************************************************************************
  Filename:  odui.c

  by Timothy A. Thompson

  OBJECT DEFINITION USER INTERFACE  (ODUI)

*************************************************************************/

#include "defs.h"
#include "dbdefs.h"
#include "menudefs.h"
#include "generator.h"
#include "kindefs.h"

#define POLYS 16

struct menutype supermenu, menu1, chgworldmenu, addobjmenu, colormenu;
struct menutype mainmenu, dumbmenu1, dumbmenu2, dumbmenu3, dumbmenu4,
                          dumbmenu5, dumbmenu6, dumbmenu7, dumbmenu8,
                          dumbmenu9, dumbmenu10;

vector view;
KOBJ *Scene;

/*************************************************************************
  main -  main program routine

*************************************************************************/
main ()
{
  int i, option;

  InitializeMenus ();

  InitializeWindowLocs ();
  InitOrtho (-500.0, 500.0, -500.0, 500.0, 500.0, -500.0);

  InitDataBase ();

  ginit ();
  gconfig ();

  cursoff ();

  qdevice (LEFTMOUSE);
  qdevice (MIDDLEMOUSE);
  qdevice (RIGHTMOUSE);

  BuildColorMap ();
```

```
    PopupColorInit ();

    setdepth (0x000, 0xFFF);
    zbuffer (TRUE);
    zclear ();

    SetWindow (WholeScreen);
    color (BLUE);
    clear ();
    InitTextWindow ();

    for (i=FrontView; i<TextWindow; i++)
      BorderWindow (i);

    view.i = 0.0;
    view.j = 0.0;
    view.k = 1.0;

    Scene = NewKObj ("scene");

    pushmenu (&supermenu);
    openmenus ();
    option = SIM;
    while (option != EXITPROGRAM) {
      option = checkmenu();
      closemenus ();
      switch (option) {
        case SIM: Tumble (); break;
        case ODUI: OduiRoutine(); break;
        case EXITPROGRAM: break;
        default: break;
      }
      openmenus();
    }
    closemenus ();
    popmenu ();

    unqdevice (RIGHTMOUSE);
    unqdevice (MIDDLEMOUSE);
    unqdevice (LEFTMOUSE);

    gexit ();

    system ("gclear");
    printf ("ODUI terminated.\n");
}

/*****************************************************************************
  InitializeMenus -  Initializes the locations and titles of all menus used

                     by ODUI.
```

```
*************************************************************************/
InitializeMenus ()
{
  supermenu.x = 0;
  supermenu.y = 757;
  supermenu.title = "SUPER MENU";
  supermenu.list = superselection;

  menu1.x = 200;
  menu1.y = 757;
  menu1.title = "MAIN MENU";
  menu1.list = menu1select;

  addobjmenu.x = 400;
  addobjmenu.y = 757;
  addobjmenu.title = "ADD AN OBJECT";
  addobjmenu.list = addobjselect;

  chgworldmenu.x = 400;
  chgworldmenu.y = 757;
  chgworldmenu.title = "CHANGE WORLD MENU";
  chgworldmenu.list = chgworldselect;

  colormenu.x = 0;
  colormenu.y = 500;
  colormenu.title = "COLOR SELECTION MENU";
  colormenu.list = colorselect;

  mainmenu.x = 0;
  mainmenu.y = 757;
  mainmenu.title = "SIMULATION MENU";
  mainmenu.list = axisselection;

  dumbmenu1.x = 200;
  dumbmenu1.y = 757;
  dumbmenu1.title = "SAMPLE MENU 1";
  dumbmenu1.list = dummychoices;

  dumbmenu2.x = 400;
  dumbmenu2.y = 757;
  dumbmenu2.title = "SAMPLE MENU 2";
  dumbmenu2.list = dummychoices;

  dumbmenu3.x = 600;
  dumbmenu3.y = 757;
  dumbmenu3.title = "SAMPLE MENU 3";
  dumbmenu3.list = dummychoices;

  dumbmenu4.x = 800;
  dumbmenu4.y = 757;
  dumbmenu4.title = "SAMPLE MENU 4";
  dumbmenu4.list = dummychoices;
```

```
dumbmenu5.x = 0;
dumbmenu5.y = 500;
dumbmenu5.title = "SAMPLE MENU 5";
dumbmenu5.list = dummychoices;

dumbmenu6.x = 200;
dumbmenu6.y = 500;
dumbmenu6.title = "SAMPLE MENU 6";
dumbmenu6.list = dummychoices;

dumbmenu7.x = 400;
dumbmenu7.y = 500;
dumbmenu7.title = "SAMPLE MENU 7";
dumbmenu7.list = dummychoices;

dumbmenu8.x = 600;
dumbmenu8.y = 500;
dumbmenu8.title = "SAMPLE MENU 8";
dumbmenu8.list = dummychoices;

dumbmenu9.x = 800;
dumbmenu9.y = 500;
dumbmenu9.title = "SAMPLE MENU 9";
dumbmenu9.list = dummychoices;

dumbmenu10.x = 0;
dumbmenu10.y = 300;
dumbmenu10.title = "SAMPLE MENU 10";
dumbmenu10.list = dummychoices10;
}

/*************************************************************************
  Tumble -  This is a test routine activated by the "Simulation" selection
            in the Super Menu.  It was written for testing and
demonstration
            purposes.  This routine will be deleted before this program is
            completed.
*************************************************************************/
Tumble ()
{
  int i, command, oldcommand, tmpcom, xx, yy, zz, dummy;
  int mx, my, mx2, my2, wind, wind2;
  float x1, y1, z1, x2, y2, z2, xx1, yy1, zz1, xx2, yy2, zz2;
  Device button, button2;
  Boolean Done;
  KOBJ *Object1, *Object2, *Object3, *Object4, *Object5, *obj;
  XFORM *xfm, *xfm1, *xfm2, *xfm3, *xfm4, *xfm5;
  OBJECT *rbody;
```

```
xx = yy = zz = 0;
command = XAXIS;
qreset ();

Done = FALSE;

Object1 = NewKObj ("object1");
xfm1 = NewXform ();
SetXformRotMulti (xfm1, 0, 0, 0);
AddXform (Object1, xfm1);

Object2 = NewKObj ("object2");
xfm = NewXform ();
SetXformTrans (xfm, 'x', -250.0);
AddXform (Object2, xfm);
xfm2 = NewXform ();
SetXformRotMulti (xfm2, 0, 0, 0);
AddXform (Object2, xfm2);

Object3 = NewKObj ("object3");
xfm = NewXform ();
SetXformTrans (xfm, 'z', -250.0);
AddXform (Object3, xfm);
xfm3 = NewXform ();
SetXformRotMulti (xfm3, 0, 0, 0);
AddXform (Object3, xfm3);

Object4 = NewKObj ("object4");
xfm = NewXform ();
SetXformTrans (xfm, 'x', 250.0);
AddXform (Object4, xfm);
xfm4 = NewXform ();
SetXformRotMulti (xfm4, 0, 0, 0);
AddXform (Object4, xfm4);

Object5 = NewKObj ("object5");
xfm = NewXform ();
SetXformTransMulti (xfm, -100.0, 0.0, 300.0);
AddXform (Object5, xfm);
xfm5 = NewXform ();
SetXformRotMulti (xfm5, 0, 0, 0);
AddXform (Object5, xfm5);

AddKObj (Scene, Object1);
AddKObj (Scene, Object2);
AddKObj (Scene, Object3);
AddKObj (Scene, Object4);
AddKObj (Scene, Object5);

obj = NewKObj ("pipe1");
rbody = GenPipe (magenta, POLYS, 250.0, 200.0, 200.0, 0);
```

```
SetKObj_Rbody (obj, rbody);
AddKObj (Object1, obj);

obj = NewKObj ("cylinder1");
rbody = GenCylinder (cyan, POLYS, 100.0, 300.0, 0);
SetKObj_Rbody (obj, rbody);
AddKObj (Object1, obj);
xfm = NewXform ();
SetXformTrans (xfm, 'y', -50.0);
AddXform (obj, xfm);

obj = NewKObj ("sphere1");
rbody = GenSphere (cyan, POLYS, 100.0, 0);
SetKObj_Rbody (obj, rbody);
AddKObj (Object1, obj);
xfm = NewXform ();
SetXformTrans (xfm, 'y', -50.0);
AddXform (obj, xfm);

obj = NewKObj ("sphere2");
rbody = GenSphere (cyan, POLYS, 100.0, 0);
SetKObj_Rbody (obj, rbody);
AddKObj (Object1, obj);
xfm = NewXform ();
SetXformTrans (xfm, 'y', 250.0);
AddXform (obj, xfm);

obj = NewKObj ("sphere3");
rbody = GenSphere (yellow, POLYS, 75.0, 0);
SetKObj_Rbody (obj, rbody);
AddKObj (Object2, obj);

obj = NewKObj ("cone1");
rbody = GenCone (green, POLYS, 100.0, 300.0, 0);
SetKObj_Rbody (obj, rbody);
AddKObj (Object3, obj);

obj = NewKObj ("parallelepiped1");
rbody = GenParallelepiped (white, 150.0, 75.0, 50.0, 45.0, 30.0, 0);
SetKObj_Rbody (obj, rbody);
AddKObj (Object4, obj);

obj = NewKObj ("box1");
rbody = GenBox (blue, 200.0, 100.0, 50.0, 0);
SetKObj_Rbody (obj, rbody);
AddKObj (Object5, obj);

obj = NewKObj ("cube1");
rbody = GenCube (red, 100.0, 0);
SetKObj_Rbody (obj, rbody);
AddKObj (Object5, obj);
xfm = NewXform ();
```

```
SetXformTransMulti (xfm, 50.0, 50.0, 0.0);
AddXform (obj, xfm);

obj = NewKObj ("box2");
rbody = GenBox (blue, 200.0, 100.0, 50.0, 0);
SetKObj_Rbody (obj, rbody);
AddKObj (Object5, obj);
xfm = NewXform ();
SetXformTrans (xfm, 'y', 150.0);
AddXform (obj, xfm);

while (!Done) {
  Draw3WinScene (Scene, view);
  printf("  %d %d %d  \n",xx, yy, zz);
if (qtest())
  if (qread(&dummy) == RIGHTMOUSE) {
    oldcommand = command;
    pushmenu (&mainmenu);
    openmenus ();
    command = DUMBCHOICE;
    while ((command == DUMBCHOICE) || (command == SELECT)) {
      command = checkmenu ();
      if (command == DUMBCHOICE) {
        pushmenu (&dumbmenu1);
        tmpcom = checkmenu ();
        while (tmpcom == 1) {
          pushmenu (&dumbmenu2);
          tmpcom = checkmenu ();
          while (tmpcom == 1) {
            pushmenu (&dumbmenu3);
            tmpcom = checkmenu ();
            while (tmpcom == 1) {
              pushmenu (&dumbmenu4);
              tmpcom = checkmenu ();
              while (tmpcom == 1) {
                pushmenu (&dumbmenu5);
                tmpcom = checkmenu ();
                while (tmpcom == 1) {
                  pushmenu (&dumbmenu6);
                  tmpcom = checkmenu ();
                  while (tmpcom == 1) {
                    pushmenu (&dumbmenu7);
                    tmpcom = checkmenu ();
                    while (tmpcom == 1) {
                      pushmenu (&dumbmenu8);
                      tmpcom = checkmenu ();
                      while (tmpcom == 1) {
                        pushmenu (&dumbmenu9);
                        tmpcom = checkmenu ();
                        while (tmpcom == 1) {
                          pushmenu (&dumbmenu10);
                          tmpcom = checkmenu ();
```

```
                        popmenu ();
                        tmpcom = checkmenu ();
                      }
                      popmenu ();
                      tmpcom = checkmenu ();
                    }
                    popmenu ();
                    tmpcom = checkmenu ();
                  }
                  popmenu ();
                  tmpcom = checkmenu ();
                }
                popmenu ();
                tmpcom = checkmenu ();
              }
              popmenu ();
              tmpcom = checkmenu ();
            }
            popmenu ();
            tmpcom = checkmenu ();
          }
          popmenu ();  /* 1 */
        }
        if (command == SELECT) {
          closemenus ();
          curson ();
          button = MIDDLEMOUSE;
          button2 = LEFTMOUSE;
          TurnOnCross ();
          while (GetLineCross (button, &mx, &my)) {
            printf ("window = %d\n",wind = WhichWindow(mx, my));
            FindLine (mx, my, wind, &x1, &y1, &z1, &x2, &y2, &z2);
            printf ("p1 - %f %f %f\n",x1,y1,z1);
            printf ("p2 - %f %f %f\n\n",x2,y2,z2);
            if (GetPointCross (button2, mx, my, &mx2, &my2)) {
              printf ("window = %d\n", wind2 = WhichWindow(mx2, my2));
              FindLine (mx2, my2, wind2, &xx1, &yy1, &zz1, &xx2, &yy2,
&zz2);
              printf ("p1 - %f %f %f\n",xx1, yy1, zz1);
              printf ("p2 - %f %f %f\n\n",xx2, yy2, zz2);
              printf ("Common Point - ");
              if (x1 == x2 && y1 == y2) {
                printf ("%f %f %f\n", x1, y1, zz1);
              }
              if (x1 == x2 && z1 == z2) {
                printf ("%f %f %f\n", x1, yy1, z1);
```

```
                }
                if (y1 == y2 && z1 == z2) {
                  printf ("%f %f %f\n", xx1, y1, z1);
                }
              }
            }
            cursoff ();
            TurnOffCross ();
            openmenus ();
            command = SAME;
            while (!qtest());
            qreset ();
          }
          if (command == CSHELL) {
            closemenus ();
            CShell ();
            openmenus ();
            command = SAME;
          }
        }
        closemenus ();
        popmenu ();
        if (command == SAME) command = oldcommand;
        if (command == DONE) Done = TRUE;
      }

      switch (command) {
        case XAXIS:
          xx += 10;
          xfm1->amt.rot.x = xx;
          xfm2->amt.rot.x = xx;
          xfm3->amt.rot.x = xx;
          xfm4->amt.rot.x = xx;
          xfm5->amt.rot.x = xx;
          break;
       case YAXIS:
          yy += 10;
          xfm1->amt.rot.y = yy;
          xfm2->amt.rot.y = yy;
          xfm3->amt.rot.y = yy;
          xfm4->amt.rot.y = yy;
          xfm5->amt.rot.y = yy;
          break;
       case ZAXIS:
          zz += 10;
          xfm1->amt.rot.z = zz;
          xfm2->amt.rot.z = zz;
          xfm3->amt.rot.z = zz;
          xfm4->amt.rot.z = zz;
          xfm5->amt.rot.z = zz;
          break;
       case ALLTHREE:
```

```
            xx += 10;
            yy += 10;
            zz += 10;
            xfm1->amt.rot.x = xx;
            xfm2->amt.rot.x = xx;
            xfm3->amt.rot.x = xx;
            xfm4->amt.rot.x = xx;
            xfm5->amt.rot.x = xx;
            xfm1->amt.rot.y = yy;
            xfm2->amt.rot.y = yy;
            xfm3->amt.rot.y = yy;
            xfm4->amt.rot.y = yy;
            xfm5->amt.rot.y = yy;
            xfm1->amt.rot.z = zz;
            xfm2->amt.rot.z = zz;
            xfm3->amt.rot.z = zz;
            xfm4->amt.rot.z = zz;
            xfm5->amt.rot.z = zz;
            break;
          default:
            break;
        }
      }
    }

    /*************************************************************************
      Draw3WinScene -  Given a scene or object, this routine draws all the
                       objects in the scene or object in each of the three
                       projection windows.

      Arguments:
          scene --  (KOBJ *) pointer to the scene to draw.
          v --   (vector) viewing vector.  (Normalized vector pointing from the
                 origin of the world coordinate system in a direction "out of
                 the screen".   This vector is used to determine the light
    source.
    *************************************************************************/
    Draw3WinScene (scene, v)
    KOBJ *scene;
    vector v;
    {
      SetWindow (FrontView);
      DrawObj (scene, v);

      SetWindow (SideView);
      PushAll (v);
      RotateAll (&v, -900, 'y');
      DrawObj (scene, v);
      PopAll (&v);
```

```
  SetWindow (TopView);
  PushAll (v);
  RotateAll (&v, 900, 'x');
  DrawObj (scene, v);
  PopAll (&v);
}

/*************************************************************************
  DrawObj -  Given a scene, this routine draw it in the current window.

  Arguments:
      scene --  (KOBJ *) pointer to the scene or object to be displayed.
      v --  (vector)  normalized viewing vector.  (See descripion of "v"
            in "Draw3WinScene".

*************************************************************************/
DrawObj (scene, v)
KOBJ *scene;
vector v;
{
  XFORM *xfrm;

  if (scene) {
    PushAll (v);
    scale (scene->scale, scene->scale, scene->scale);
    xfrm = scene->xform;
    if (xfrm) {
      while (xfrm) {
        switch (xfrm->type) {
          case 't':
            switch (xfrm->axis) {
              case 'a':
                translate(xfrm->amt.trans.x, xfrm->amt.trans.y,
                                                    xfrm->amt.trans.z);

                break;
              case 'x':
                translate (xfrm->amt.dist, 0.0, 0.0);
                break;
              case 'y':
                translate (0.0, xfrm->amt.dist, 0.0);
                break;
              case 'z':
                translate (0.0, 0.0, xfrm->amt.dist);
                break;
              default:
                break;
            }
            break;
          case 'r':
            if (xfrm->axis == 'a')
              RotateMultiEnv(&v,xfrm->amt.rot.x, xfrm->amt.rot.y,
```

```
xfrm->amt.rot.z);
              else
                RotateAll (&v, xfrm->amt.angle, xfrm->axis);
              break;
            default:
              break;
          }
          xfrm = xfrm->next;
        }
      }
      if (scene->type == 'r')
        DrawRbody (scene->obtype.rbody, v);
      if (scene->type == 'o')
        DrawObj (scene->obtype.subobj, v);
      PopAll (&v);
      DrawObj (scene->nextkobj, v);
    }
  }

  /*****************************************************************************
    DrawRbody -  Draws a rigid body in the current window.

    Arguments:
        rbody --  (OBJECT *) pointer to the rigid body to be drawn.
        v -- (vector) normalized viewing vector.

  *****************************************************************************/
  DrawRbody (rbody, v)
  OBJECT *rbody;
  vector v;
  {
    FACE *ply;
    float i, j, k, x, y, z;
    int colr, flags;
    float dp;
    VERTEX *valid;

    if (rbody) {
      ply = FirstFace(rbody);
      while (ply) {
        GetAttribute (ply, &i, &j, &k, &colr, &flags);
        dp = (i*v.i + j*v.j + k*v.k) * (float)(span-1);
        if (dp > 0.0001) {
          valid = GetVert (&x, &y, &z);
          color (colr + (int)dp);
          if (valid) {
            pmv (x, y, z);
            valid = GetVert (&x, &y, &z);
            while (valid) {
              pdr (x, y, z);
              valid = GetVert (&x, &y, &z);
```

```
            }
            pclos ();
          }
        }
        ply = NextFace (ply);
      }
    }
  }

/*****************************************************************************
  OduiRoutine -  This is going to be the main routine of the program
                 when finished.  All main functions of ODUI will be invoked

                 from here.  Some of the routines which are called directly

                 from here now may be called from some other place in the
                 future.

*****************************************************************************/
OduiRoutine ()
{
  int option;
  char scenename[20];

  pushmenu (&menu1);
  openmenus ();
  option = WORLD;
  while (option != DONE) {
    option = checkmenu();
    closemenus ();
    switch (option) {
      case WORLD: ChgWorldView (); break;
      case ADD: AddAnObject (); break;
      case DELETE: DeleteAnObject (); break;
      case SAVESCENE: SaveObj (Scene); break;
      case LOADSCENE: printf ("Enter name of scene to load.\n");
                      scanf ("%s", scenename);
                      LoadObj (Scene, scenename);
                      Draw3WinScene (Scene, view);
                      break;
      case HACK: InitOrtho(-500.,500.,-900.,100.,500.,-500.); break;
      case CSHELL: CShell (); break;
      case DONE: break;
      default: break;
    }
    openmenus();
  }
  closemenus ();
  popmenu ();
}

/*****************************************************************************
```

```
  ChgWorldView -  This function handles rotations, translations, zooming,
                  and clipping plane changes for the global scene.

*****************************************************************************/
ChgWorldView ()
{
  int option;

  pushmenu (&chgworldmenu);
  openmenus ();
  option = WORLD;
  while (option != DONE) {
    option = checkmenu();
    closemenus ();
    switch (option) {
      case ROTATE: RotateWorld (); break;
      case TRANSLATE: TranslateWorld (); break;
      case ZOOM: ZoomWorld (); break;
      case CLIP: Clip (); break;
      case DONE: break;
      default: break;
    }
    openmenus();
  }
  closemenus ();
  popmenu ();
}

/*****************************************************************************
  AddAnObject -  This routine will be the general routine called when
                 an object is to be added to the scene.  This routine
                 is NOT finished and will probably recieve a good deal
                 of revision.

*****************************************************************************/
AddAnObject ()
{
  int option, col;
  int mx, my, mx2, my2, wind, wind2;
  Boolean pointvalid;
  float x1, y1, z1, x2, y2, z2, xx1, yy1, zz1, xx2, yy2, zz2;
  Device button;
  OBJECT *rbody;
  KOBJ *obj;
  XFORM *xfrm;

  pushmenu (&addobjmenu);
  openmenus ();
  option = CYLINDER;
  while (option != DONE) {
    option = checkmenu();
    closemenus ();
```

```
pointvalid = FALSE;
if (option != DONE) {
  button = RIGHTMOUSE;
  printf ("Select Origin with RIGHT button.\n\n");
  TurnOnCross ();
  if (GetLineCross (button, &mx, &my)) {
    wind = WhichWindow(mx, my);
    FindLine (mx, my, wind, &x1, &y1, &z1, &x2, &y2, &z2);
    if (GetPointCross (button, mx, my, &mx2, &my2)) {
      pointvalid = TRUE;
      wind2 = WhichWindow(mx2, my2);
      FindLine (mx2, my2, wind2, &xx1, &yy1, &zz1, &xx2, &yy2, &zz2);
      xfrm = NewXform ();
      if (x1 == x2 && y1 == y2) {
        SetXformTransMulti (xfrm, x1, y1, zz1);
      }
      if (x1 == x2 && z1 == z2) {
        SetXformTransMulti (xfrm, x1, yy1, z1);
      }
      if (y1 == y2 && z1 == z2) {
        SetXformTransMulti (xfrm, xx1, y1, z1);
      }
    }
  }
  TurnOffCross ();
}
switch (option) {
  case CYLINDER:
    if (pointvalid) {
      col = ChooseColor ();
      obj = NewKObj ("cylinder");
      rbody = GenCylinder (col, POLYS, 100.0, 300.0, 0);
      SetKObj_Rbody (obj, rbody);
      AddKObj (Scene, obj);
      AddXform (obj, xfrm);
    }
    break;
  case PIPE:
    if (pointvalid) {
      col = ChooseColor (col);
      obj = NewKObj ("pipe");
      rbody = GenPipe (col, POLYS, 250.0, 200.0, 200.0, 0);
      SetKObj_Rbody (obj, rbody);
      AddKObj (Scene, obj);
      AddXform (obj, xfrm);
    }
    break;
  case SPHERE:
    if (pointvalid) {
      col = ChooseColor ();
      obj = NewKObj ("sphere");
      rbody = GenSphere (col, POLYS, 75.0, 0);
```

```
            SetKObj_Rbody (obj, rbody);
            AddKObj (Scene, obj);
            AddXform (obj, xfrm);
          }
          break;
        case CONE:
          if (pointvalid) {
            col = ChooseColor ();
            obj = NewKObj ("cone");
            rbody = GenCone (col, POLYS, 100.0, 300.0, 0);
            SetKObj_Rbody (obj, rbody);
            AddKObj (Scene, obj);
            AddXform (obj, xfrm);
          }
          break;
        case CUBE:
          if (pointvalid) {
            col = ChooseColor ();
            obj = NewKObj ("cube");
            rbody = GenCube (col, 100.0, 0);
            SetKObj_Rbody (obj, rbody);
            AddKObj (Scene, obj);
            AddXform (obj, xfrm);
          }
          break;
        case BOX:
          if (pointvalid) {
            col = ChooseColor ();
            obj = NewKObj ("box");
            rbody = GenBox (col, 200.0, 100.0, 50.0, 0);
            SetKObj_Rbody (obj, rbody);
            AddKObj (Scene, obj);
            AddXform (obj, xfrm);
          }
          break;
        case PARALLELEPIPED:
          if (pointvalid) {
            col = ChooseColor ();
            obj = NewKObj ("parallelepiped");
            rbody = GenParallelepiped (col, 150.0, 75.0, 50.0, 45.0, 30.0,
0);
            SetKObj_Rbody (obj, rbody);
            AddKObj (Scene, obj);
            AddXform (obj, xfrm);
          }
          break;
        case DONE: break;
        default: break;
      }
      if (option != DONE)
        Draw3WinScene (Scene, view);
      openmenus();
```

```
    }
    closemenus ();
    popmenu ();
}

/*****************************************************************************
  DeleteAnObject -  Deletes an object from the scene.

*****************************************************************************/
DeleteAnObject ()
{
}

/*****************************************************************************
  RotateWorld -  Rotates the entire scene.

*****************************************************************************/
RotateWorld ()
{
}

/*****************************************************************************
  TranslateWorld -  Translates the entire scene.

*****************************************************************************/
TranslateWorld ()
{
}

/*****************************************************************************

  ZoomWorld -  Changes the clipping planes so as to zoom/unzoom the scene.


*****************************************************************************/
ZoomWorld ()
{
}
 ****************************************************************************
  Clip -  Currently used to manually set the clipping planes.  Should be
          replaced in the future by a routine which is easier to use and
          which will keep the user from being able to generate distorted
          displays.

*****************************************************************************/

Clip ()
{
  float left, right, bottom, top, front, back;

  system ("clear");
  printf ("Current View Bounds:\n\n");
```

```
  DisplayOrtho ();
  if (YNresponse ("Change")) {
    printf ("Enter NEW clipping planes:\n");
    printf ("Left   : "); scanf ("%f", &left); printf("\n");
    printf ("Right  : "); scanf ("%f", &right); printf("\n");
    printf ("Top    : "); scanf ("%f", &top); printf("\n");
    printf ("Bottom : "); scanf ("%f", &bottom); printf("\n");
    printf ("Front  : "); scanf ("%f", &front); printf("\n");
    printf ("Back   : "); scanf ("%f", &back); printf("\n");
    InitOrtho (left, right, bottom, top, front, back);
    system ("clear");
    printf ("New View Bounds:\n\n");
    DisplayOrtho ();
  }
}

/*****************************************************************************
  ChooseColor -  General purpose color selection routine.  The "Set
                 Default Color" option is currently unimplemented.

*****************************************************************************/
ChooseColor ()
{
  int option, col;
  static int deflt = RED;

  colormenu.list[12].flag = FALSE;
  pushmenu (&colormenu);
  openmenus ();
  option = SETDEFCOLOR;
  while (option == SETDEFCOLOR) {
    option = checkmenu ();
    closemenus();
    if (option == DEFAULTCOLOR)
      col = deflt;
    else
    {
      if (option == SETDEFCOLOR) {
      }
      else
        col = option;
    }
    openmenus();
  }
  closemenus ();
  popmenu ();
  colormenu.list[12].flag = TRUE;
  return col;
}

/*****************************************************************************
  CShell -  This spawns a UNIX C-Shell in the text window.  The shell can
```

```
            be exited by typing "exit" or CONTROL-D at a prompt.
*****************************************************************/

CShell ()
{
  int dummy;

  system ("/bin/csh");
  while (qtest()) {
    qread (&dummy);
  }
  qreset ();
}

/*****************************************************************
  YNresponse -  This function was intened to be used to be used to ask
                a question and wait for the user to respond by typing
                'y' or 'n'.   There seems to be some problem with it,
though.

  Arguments:
      ch -  (char *) string containing the question to ask.

  Value Returned:  (Boolean) TRUE if the user typed 'y' or 'Y'.
                             FALSE if the user typed 'n' or 'N'.

*****************************************************************/
Boolean YNresponse (ch)
char *ch;
{
  char res;

  printf ("%s? ",ch);
  while (res = getchar(), res!='y' && res!='Y' && res!='n' && res!='N');
  printf ("\n");
  return res == 'y' || res == 'Y';
}
/*****************************************************************
  Unimplemented -  This routine is used to indicate that the user tried
                   to do something that is currently not implemented.

*****************************************************************/
Unimplemented ()
{
  fprintf (stderr, "UNIMPLEMENTED\n\n");
}
```

Appendix A.3 Data Base

```
/*************************************************************************
  Filename:  interface.c

  by Timothy A. Thompson

  Purpose:  The purpose of this package is to allow easy use of the
            rigid body database by hiding the actual structure of the
            underlying database from the application program using it.

  Functions Provided:  AddCorner ()
                       AddPolygon ()
                  <*>  DumpCore ()
                  <*>  DumpVert ()
                       FirstFace ()
                       GetAttribute ()
                       GetVert ()
                       InitDataBase ()
                       NewAttribute ()
                       NewCorn ()
                       NewRb ()
                       NewVertex ()
                       NextFace ()
                       SetAttribute ()
                       SetCorner ()
                       SetVertex ()
                       SameFace ()
                       UniqueRbNum ()

  <*> Diagnostic functions provided for database debugging purposes.

*************************************************************************/

#include "defs.h"
#include "dbdefs.h"

Boolean FaceValid = FALSE;
int CurrentRbNumber = 0;

/*************************************************************************

  UniqueRbNum -  Returns a unique integer number each time is is called for

                 use as a rigid body number.

  Value Returned:  (int) unique integer number.

*************************************************************************/
UniqueRbNum ()
{
  return CurrentRbNumber++;
```

```
}

/*************************************************************************
  InitDataBase  -  Any  chores  necessary  for  the  initialization  of  the
database
                   should be placed in this routine.
*************************************************************************/
InitDataBase ()
{

}

/*************************************************************************
  AddPolygon - Makes a polygon out of the current list of corners
               (built with AddCorner) and adds it to the list of polygons
               which  make  up  the  Rigid  Body  pointed  to  by  rb.    The
attribute
               record pointed to by atr is associated with this polygon.

  Arguments:
      rb --  (OBJECT *) points to the rigid body to which the polygon is to
be
             added.
      atr --   (ATTRIBUTE *)  points  to  the  attribute  structure  for  the
polygon
              being added.
*************************************************************************/
AddPolygon (rb, atr)
OBJECT *rb;
ATTRIBUTE *atr;
{
  FaceValid = FALSE;
  obj = rb;
  attr = atr;
  add_polygon ();
}

/*************************************************************************
  AddCorner - Adds a corner pointed to by cn to the list of corners
              "owned" by the rigid body pointed to by rb.  These are free
              corners which have not yet been made into a polygon by
              AddPolygon.

  Arguments:
      rb --  (OBJECT *) points to the rigid body to which the corner is to
be
             added.
      cn --  (CORNER *) points to the corner to be added.
*************************************************************************/
AddCorner (rb, cn)
OBJECT *rb;
CORNER *cn;
```

```
{
  FaceValid = FALSE;
  obj = rb;
  corn = cn;
  add_corner ();
}

/*************************************************************************
  NewRb - Creates a new rigid body identified by the integer rbnum which
          should be unique.

  Arguments:
      rbnum --  (int) unique integer "name" for this rigid body.

  Value Returned:  (OBJECT *) pointer to the new rigid body or NULL if
                   space for the rigid body could not be allocated.
*************************************************************************/
OBJECT *NewRb (rbnum)
int rbnum;
{
  OBJECT *tmp;

  if ((tmp = (OBJECT *) malloc (sobj)) == NULL) {
    fprintf (stderr, "interface: malloc failed in NewRb\n");
  }
  else
  {
    FaceValid = FALSE;
    tmp->name = rbnum;
    tmp->nextobj = NULL;
    tmp->fce = NULL;
    tmp->corn = NULL;
    tmp->rcorn = NULL;
    obj = tmp;
  }
  return tmp;
}

/*************************************************************************
  NewCorn - Creates a new corner (which will contain a vertex).

  Value Returned:  (CORNER *)  pointer to the new corner or NULL if space
                   for the new corner could not be allocated.
*************************************************************************/
CORNER *NewCorn ()
{
  CORNER *tmp;

  if ((tmp = (CORNER *) malloc (scorn)) == NULL) {
    fprintf (stderr, "interface: malloc failed in NewCorn\n");
  }
  else
```

```
    {
      tmp->nextcorn = NULL;
      tmp->vtx = NULL;
    }
    return tmp;
}

/*************************************************************************
  SetCorner - Sets the corner pointed to by cn with the vertex pointed to
              by vert.

  Arguments:
      cn --  (CORNER *) points to the corner to set.
      vert --  (VERTEX *) points to the vertex which the corner is to be
               defined as.
*************************************************************************/
SetCorner (cn, vert)
CORNER *cn;
VERTEX *vert;
{
  cn->vtx = vert;
}

/*************************************************************************
  NewVertex - Creates a new vertex structure.

  Value Returned:  (VERTEX *) pointer to the new vertex or NULL if space
                   could not be allocated for the new vertex.
*************************************************************************/
VERTEX *NewVertex ()
{
  VERTEX *tmp;

  if ((tmp = (VERTEX *) malloc (svtx)) == NULL) {
    fprintf (stderr, "interface: malloc failed in NewVertex\n");
  }
  else
  {
    tmp->iedg = NULL;
  }
  return tmp;
}

/*************************************************************************
  SetVertex - Sets the values of the coordinates of a newly created vertex


  Arguments:
      vtx --  (VERTEX *) pointer to the vertex to set.
      x, y, z --  (float) x, y, and z coordinates to which the vertex is t

                  be set.
```

```
**********************************************************************/

SetVertex (vtx, x, y, z)
VERTEX *vtx;
float x, y, z;
{
  vtx->x = x;
  vtx->y = y;
  vtx->z = z;
}

/**********************************************************************
  GetVert - Returns the coordinates of the next vertex of a polygon (face)
            or returns FALSE to indicate that there are no more vertices in
            the polygon.  The face from which the vertices are to be taken
            should be selected by first calling "FirstFace", "NextFace",
            or "SameFace".

   Arguments:
       x, y, z --  (float *) x, y, and z coordinates returned.

   Value Returned:  (Boolean) Returns TRUE (non-zero value which happens to
                    be a pointer to the VERTEX record of the vertex)
                    if x, y, and z contain valid coordinates.  Returns FALSE
                    (zero) if the last vertex in the polygon has already
                    been returned.
**********************************************************************/
VERTEX *GetVert (x, y, z)
float *x, *y, *z;
{
  if (!FaceValid) {
    abandon_polygon ();
    loop_poly (1);
    corn = obj->rcorn;
    FaceValid = TRUE;
  }
  if (corn) {
    vtx = corn->vtx;
    *x = vtx->x;
    *y = vtx->y;
    *z = vtx->z;
    corn = corn->nextcorn;
  }
  else
  {
    FaceValid = FALSE;
  }
```

```
  return (FaceValid ? vtx : NULL);
}

/*****************************************************************
  NewAttribute - Creates a new attribute structure.

  Value Returned:  (ATTRIBUTE *) pointer to the new attribute structure or
                   NULL if space could not be allocated for the new
attribute.
*****************************************************************/
ATTRIBUTE *NewAttribute ()
{
  ATTRIBUTE *tmp;

  if ((tmp = (ATTRIBUTE *) malloc (sattr)) == NULL) {
    fprintf (stderr, "interface: malloc failed in NewAttribute\n");
  }
  return tmp;
}

/*****************************************************************

  SetAttribute - Sets the attributes in an attribute structure.

  Arguments:
      attr --  (ATTRIBUTE *) pointer to the attribute structure to set.
      i, j, k --  (float) i, j, and k components of the normal vector of
the
                  polygon.  The vector should be oriented so it points out
                  from the side of the polygon which should be visible.
      colr --  (int) base color of the polygon.  (See the "defs.h" file for
                base color values.)
      flag --  (int) Reserved for future use.
*****************************************************************/
SetAttribute (attr, i, j, k, colr, flag)
ATTRIBUTE *attr;
float i, j, k;
int colr, flag;
{
  attr->norm.i = i;
  attr->norm.j = j;
  attr->norm.k = k;
  attr->colr = colr;
  attr->flags = flag;
}

/*****************************************************************

  GetAttribute - Gets the attributes from an attribute structure.
```

```
    Arguments:
        face --  (FACE *) pointer to the face from which to attributes.
        i, j, k --  (float *) i, j, and k components of the normal vector of

                    the polygon.
        colr --  (int *) base color of the polygon.
        flag --  (int *) Reserved for future use.

      (Note:  See SetAttribute for more information on these values.)
    ************************************************************************/

    GetAttribute (face, i, j, k, colr, flag)
    FACE *face;
    float *i, *j, *k;
    int *colr, *flag;
    {
      ATTRIBUTE *attr;

      attr = face->attr;
      *i = attr->norm.i;
      *j = attr->norm.j;
      *k = attr->norm.k;
      *colr = attr->colr;
      *flag = attr->flags;
    }

    /************************************************************************

      FirstFace - Get the pointer to the first polygon of a rigid body.

      Arguments:
          rb --  (OBJECT *) pointer to the rigid body from which to return
                 the first face.

      Value Returned:  (FACE *) pointer to the first face of the list of faces

                       which make up the rigid body.
    ************************************************************************/

    FACE *FirstFace (rb)
    OBJECT *rb;
    {
      FaceValid = FALSE;
      obj = rb;
      fce = obj->fce;
      return fce;
    }

    /************************************************************************
      NextFace - Get the pointer to the next polygon of a rigid body.
```

```
    Arguments:
        lastfce --  (FACE *) pointer to the current face of the rigid body.

    Value Returned:  (FACE *) pointer to the next face of the rigid body or
                     NULL if "lastfce" was the last face in the list.
*************************************************************************/
FACE *NextFace (lastfce)
FACE *lastfce;
{
  if (lastfce) {
    FaceValid = FALSE;
    fce = lastfce->nextfce;
    return fce;
  }
  else
    return NULL;
}

/*************************************************************************
  SameFace - Returns a pointer to the current face.  This routine must be
             called if one wants to read the vertices of a particular face

             twice in a row, since this routine resets the list of
vertices.

  Arguments:
      lastfce --  (FACE *) pointer to the current face.

  Value Returned:  (FACE *) pointer to the current face.
*************************************************************************/

FACE *SameFace (lastfce)
FACE *lastfce;
{
  FaceValid = FALSE;
  return lastfce;
}

/*************************************************************************

  DumpCore - Traverses the database and prints out the values of all the
             pointers involved in the representation of one rigid body.
             This routine is intended for purposes of debugging.

  Arguments:
      rb --  (OBJECT *) pointer to the rigid body to traverse.
*************************************************************************/
DumpCore (rb)
OBJECT *rb;
{
  printf ("interface: About to dump database\n");
```

```
    obj = rb;
    for (fce=obj->fce; fce; fce=fce->nextfce) {
      printf ("Face:          %ld\n",fce);
      printf ("  bedge:       %ld\n",fce->bedg);
      printf ("  nextface:    %ld\n\n",fce->nextfce);
      for (bedg = fce->bedg; bedg; bedg=bedg->nextbedg) {
        printf ("Bedge:         %ld\n",bedg);
        printf ("  edge:        %ld\n",bedg->edg);
        printf ("  nextbedge:   %ld\n\n",bedg->nextbedg);
        edg = bedg->edg;
        printf ("Edge:          %ld\n",edg);
        printf ("  face1:       %ld\n",edg->fce1);
        printf ("  face2:       %ld\n",edg->fce2);
        printf ("  vert1:       %ld\n",edg->vtx1);
        printf ("  vert2:       %ld\n\n",edg->vtx2);
        DumpVert (edg->vtx1);
        DumpVert (edg->vtx2);
      }
    }
  }

  /*************************************************************************

    DumpVert - Prints information about a vertex.  This routine is for
               database debugging purposes and is called by "DumpCore".

    Arguments:
        vt --   (VERTEX *) pointer to vertex from which to print information.

  *************************************************************************/
  DumpVert (vt)
  VERTEX *vt;
  {
    printf ("Vertex:        %ld\n",vt);
    printf ("  X, Y, Z:     %f, %f, %f\n",vt->x,vt->y,vt->z);
    printf ("  iedge:       %ld\n\n",vt->iedg);
    for (iedg=vt->iedg;iedg;iedg=iedg->nextiedg) {
      printf ("Iedge:         %ld\n",iedg);
      printf ("  edge:        %ld\n",iedg->edg);
      printf ("  nextiedge:   %ld\n\n",iedg->nextiedg);
    }

  }
```

Appendix A.4 3-D windows

```
/*****************************************************************
  Filename:  window.c

  by Timothy A. Thompson

  Purpose:  This package implements and controls the three orthogonal
            projection windows and the text window.  Services include:
            setting up the color map for shading purposes, initializing and

            sizing windows, selecting current window, setting and reading
            current clipping planes, and mapping between screen coordinates

            and 3d world coordinates.

  Functions Provided:  BorderWindow ()
                       BuildColorMap ()
                       DisplayOrtho ()
                       FindLine ()
                       GetOrtho ()
                       GetWindowSides ()
                       InitializeWindowLocs ()
                       InitOrtho ()
                       InitTextWindow ()
                       SetWindow ()
                       WhichWindow ()

*****************************************************************/

#include "defs.h"

int Left[Windows], Right[Windows], Bottom[Windows], Top[Windows];
float OrthoLeft, OrthoRight, OrthoBottom, OrthoTop, OrthoNear, OrthoFar;

/*****************************************************************
  BuildColorMap -  Initializes the color map so objects can be shaded
                   properly on the screen.  Currently, seven colors are
                   supported:  red, green, yellow, blue, magenta, cyan, and

                   white.  The map is generated such that there are "span"

                   number of intensities of each color.  "span" is defined

                   in file "defs.h".

*****************************************************************/
BuildColorMap ()
{
  int i, intensity;

  /*  Color Ramp colors  */
```

```
    for (i=0; i<span; i++) {
      intensity = 256/span+125+i*128/span;
      mapcolor (red+i    , intensity,         0,         0);
      mapcolor (green+i  ,         0, intensity,         0);
      mapcolor (yellow+i , intensity, intensity,         0);
      mapcolor (blue+i   ,         0,         0, intensity);
      mapcolor (magenta+i, intensity,         0, intensity);
      mapcolor (cyan+i   ,         0, intensity, intensity);
      mapcolor (white+i  , intensity, intensity, intensity);
    }
  }

  /*************************************************************************
    InitializeWindowLocs -  Sets the default positions and sizes of the
                            three projection windows and the text window.

  *************************************************************************/
  InitializeWindowLocs ()
  {
    Left   [WholeScreen] = 0;
    Right  [WholeScreen] = 1023;
    Bottom [WholeScreen] = 0;
    Top    [WholeScreen] = 767;

    Left   [FrontView] = 0;
    Right  [FrontView] = 511;
    Bottom [FrontView] = 0;
    Top    [FrontView] = 383;

    Left   [SideView] = 512;
    Right  [SideView] = 1023;
    Bottom [SideView] = Bottom [FrontView];
    Top    [SideView] = Top    [FrontView];

    Left   [TopView] = Left  [FrontView];
    Right  [TopView] = Right [FrontView];
    Bottom [TopView] = 384;
    Top    [TopView] = 767;

    Left   [TextWindow] = Left   [SideView];
    Right  [TextWindow] = Right  [SideView];
    Bottom [TextWindow] = Bottom [TopView];
    Top    [TextWindow] = Top    [TopView];
  }

  /*************************************************************************
    InitOrtho -  Initializes the clipping planes to the values supplied.

    Arguments:
        left --  (float) left clipping plane.  (X minimum)
        right --  (float) right clipping plane.  (X maximum)
```

```
        bottom --  (float) bottom clipping plane.  (Y minimum)
        top --  (float) top clipping plane.  (Y maximum)
        near --  (float) near clipping plane.  (Z maximum)
        far --  (float) far clipping plane.  (Z minimum)

***************************************************************************/
InitOrtho (left, right, bottom, top, near, far)
float left, right, bottom, top, near, far;
{
  OrthoLeft = left;
  OrthoRight = right;
  OrthoBottom = bottom;
  OrthoTop = top;
  OrthoNear = near;
  OrthoFar = far;
}

/***************************************************************************
  GetOrtho -  Returns the current clipping plane values.

  Arguments:
      left --  (float *) returns left clipping plane.  (X minimum)
      right --  (float *) returns right clipping plane.  (X maximum)
      bottom --  (float *) returns bottom clipping plane.  (Y minumum)
      top --  (float *) returns top clipping plane.  (Y maximum)
      near --  (float *) returns near clipping plane.  (Z maximum)
      far --  (float *) returns far clipping plane.  (Z minimum)

***************************************************************************/
GetOrtho (left, right, bottom, top, near, far)
float *left, *right, *bottom, *top, *near, *far;
{
  *left = OrthoLeft;
  *right = OrthoRight;
  *bottom = OrthoBottom;
  *top = OrthoTop;
  *near = OrthoNear;
  *far = OrthoFar;
}

/***************************************************************************
  DisplayOrtho -  Prints the current clipping plane values to standard
                  output.

***************************************************************************/
DisplayOrtho ()
{
  printf ("Left  %7.2f   Right  %7.2f\n", OrthoLeft, OrthoRight);
  printf ("Top   %7.2f   Bottom %7.2f\n", OrthoTop, OrthoBottom);
  printf ("Front %7.2f   Back   %7.2f\n", OrthoNear, OrthoFar);
}
```

```
/*************************************************************************
  InitTextWindow -  Sets the size and location of the textport and clears
                    the text window.

*************************************************************************/
InitTextWindow ()
{
  textport (Left[TextWindow], Right[TextWindow], Bottom[TextWindow],
            Top [TextWindow]);
  system ("clear");
}

/*************************************************************************
  SetWindow -  Makes the indicated window active for drawing.

  Arguments:
      WindowNum --  (int) window to make active.  (See file "defs.h" for
                    window names and their corresponding number.)

*************************************************************************/
SetWindow (WindowNum)
int WindowNum;
{
  if (WindowNum != TextWindow) {
    viewport(Left[WindowNum]+1, Right[WindowNum]-1, Bottom[WindowNum]+1,
             Top[WindowNum]-1);
    color (BLACK);
    clear ();
    zclear ();
    switch (WindowNum) {
      case FrontView:

ortho(OrthoLeft,OrthoRight,OrthoBottom,OrthoTop,-OrthoNear,-OrthoFar);
        break;
      case SideView:

ortho(OrthoFar,OrthoNear,OrthoBottom,OrthoTop,-OrthoRight,-OrthoLeft);
        break;
      case TopView:

ortho(OrthoLeft,OrthoRight,OrthoFar,OrthoNear,-OrthoTop,-OrthoBottom);
        break;
      default:
        break;
    }
  }
}

/*************************************************************************
  BorderWindow -  Draws a white border around a window and makes that
window
                  active.
```

```
  Arguments:
      WindowNum --  (int) window around which to draw border.  (See file
                    "defs.h"  for  window  names  and  their  corresponding
number.)
*****************************************************************************/
BorderWindow (WindowNum)
int WindowNum;
{
  if (WindowNum != TextWindow) {
    viewport(Left[WindowNum], Right[WindowNum], Bottom[WindowNum],
             Top[WindowNum]);
    color (WHITE);
    clear ();
    SetWindow (WindowNum);
  }
}

/*****************************************************************************
  WhichWindow -  Given the x and y screen coordinates of a point, returns
an
                 integer number indicating which window the point is in.

  Arguments:
      x, y --  (int) x and y screen coordinates of point.

  Value Returned:   (int)  window  which  contains  the  point.   (See  file
"defs.h"
                   for window names and their corresponding number.)
*****************************************************************************/
WhichWindow (x, y)
int x, y;
{
  int i;

  for (i=1; i<TextWindow; i++)
    if (x > Left[i] && x < Right[i] && y > Bottom[i] && y < Top[i])
      return (i);
  return (-1);
}

/*****************************************************************************
  FindLine -  Given the x and y coordinates of a point and the window which

              contains the point, returns the x, y, and z coordinates in
              world  space  of  two  points  which  define  a  line  whose
projection
              on the screen appears as the point given.

              Note:  If the point (mx, my) is not in "window", the result
```

```
                is a line which is outside of the clipping planes.  Be sure
                the window is correct by calling "WhichWindow" first.

    Arguments:
        mx, my --   (int) screen coordinates of the projection point of line
to
                    determine.
        window --  (int) window which contains the point (mx, my)
        wx1, wy1, wz1 --  (float *)  returns x, y, and z world coordinates of
                    one endpoint of the line.
        wx2, wy2, wz2 --  (float *)  returns x, y, and z world coordinates of
                    the other endpoint of the line.
*****************************************************************************/
FindLine (mx, my, window, wx1, wy1, wz1, wx2, wy2, wz2)
int mx, my, window;
float *wx1, *wy1, *wz1, *wx2, *wy2, *wz2;
{
  switch (window) {
    case FrontView:
      *wx1 = (float)(mx-Left[window])/(float)(Right[window] - Left[window])
              * (OrthoRight - OrthoLeft) + OrthoLeft;
      *wx2 = *wx1;
      *wy1 = (float)(my-Bottom[window])/(float)(Top[window] -
Bottom[window]) * OrthoTop - OrthoBottom) + OrthoBottom;
      *wy2 = *wy1;
      *wz1 = OrthoNear;
      *wz2 = OrthoFar;
      break;
    case SideView:
      *wx1 = OrthoRight;
      *wx2 = OrthoLeft;
      *wy1  =  (float)(my-Bottom[window])/(float)(Top[window]  -
Bottom[window]) *
                                        (OrthoTop - OrthoBottom) + OrthoBottom;
      *wy2 = *wy1;
      *wz1 = (float)(mx-Left[window])/(float)(Right[window] - Left[window])
              * (OrthoFar - OrthoNear) - OrthoFar;
      *wz2 = *wz1;
      break;
    case TopView:
      *wx1 = (float)(mx-Left[window])/(float)(Right[window] - Left[window])
              * (OrthoRight - OrthoLeft) + OrthoLeft;
      *wx2 = *wx1;
      *wy1 = OrthoTop;
      *wy2 = OrthoBottom;
      *wz1 = (float)(my-Bottom[window])/(float)(Top[window] -
Bottom[window]) * (OrthoFar - OrthoNear) - OrthoFar;
      *wz2 = *wz1;
```

```
        break;
      default:
        break;
    }
}

/*****************************************************************
  GetWindowSides -  Returns (in screen coordinates) the locations of the
                    sides of the given window.

  Arguments:
      window --  (int) window whose sides are needed.
      left --  (int *) returns left side (x coordinate) of window.
      right --  (int *) returns right side (x coordinate) of window.
      bottom --  (int *) returns bottom side (y coordinate) of window.
      top --  (int *) returns top side (y coordinate) of window.

*****************************************************************/
GetWindowSides (window, left, right, bottom, top)
int window, *left, *right, *bottom, *top;
{
  if (window >= 0 && window < Windows) {
    *left = Left [window];
    *right = Right [window];
    *bottom = Bottom [window];
    *top = Top [window];
    return TRUE;
  }
  else
    return FALSE;
}
```

Appendix A.5 Pop-up Menus

```
/*************************************************************************
  Filename:  popmenu.c

  by Tim Thompson

  Purpose:      This package implements (among other things) a pop-up menu
                system for user input.  When the input routine is called,
                all active menus are displayed.  The active menu has a red
                title bar as opposed to a white title bar for the others.
                When the user moves the pointer to one of the valid (not
                dimmed) selections in the active menu and presses the right
                mouse button, a value corresponding to the selected item is
                returned.  Menus can be moved around on the screen by
                selecting the title bar of the menu to move with the right
                mouse button, moving the menu with the mouse, and then
                releasing the button.  That menu will then appear in that
                location every time it is opened until it is moved again.
                The package also supports menus with blank line separators
                and "dimmed" items which cannot be selected. The menu system
                is actually implemented as a stack where the top menu on the
                stack is the active menu.  Menus are added to the stack by
                "pushmenu" and removed by "popmenu".  A menu choice is
                returned from the active menu when "checkmenu" is called.
                Open menus (menus currently on the stack) can be turned on
                and off by calling "openmenus" and "closemenus",
                respectively.

                This package also implements a cross-hair selection system.
                The user may select points in three-dimensional space by moving
                cross-hairs in the three projection windows.  The user may
                may select either a point or a line in three-dimensional space
                by calling one of two routines.

Functions Provided:

                        calcbounds ()
                        checkmenu ()
                        closemenus ()
                        DrawPrimaryCross ()
                        DrawSecondaryCross ()
                        GetLineCross ()
                        GetPointCross ()
                        openmenus ()
                        popmenu ()
                        popup ()
                        PopupColorInit ()
                        pushmenu ()
                        RetrieveScreenEnv ()
                        shadowpopup ()
                        StoreScreenEnv ()
```

```
                        TurnOffCross ()
                        TurnOnCross ()

    Implementation Notes:
        The menus use the following structures defined in file "defs.h":
            struct popupentry {
                short type;
                char *text;
                Boolean flag;
            };

            struct menutype {
                int x;
                int y;
                char *title;
                struct popupentry *list;
            };

        The "type" field in popupentry should contain a positive number
        to be returned when the item is selected.  A popupentry with
        "type" equal to zero indicates the end of the popupentry array.
        A popupentry with "type" less than zero is assumed to be a blank
        line separator in the menu and will be ignored. (These cannot be
        selected by the user.)

        The "text" field in popupentry is the text displayed for that
        particular menu selection.

        The "flag" field should contain "FALSE" if the item is to be
        "dimmed" such that it cannot be selected.  If "flag" is "TRUE",
        the item can be selected.

        The "x" and "y" fields in menutype should contain the screen
        coordinates at which the top left corner of the menu will appear.

        The "title" field in menutype is the text that will be printed in
        the title bar of the menu.

        The "list" field is a pointer to the popupenty array for that menu.
*************************************************************************/

#include "defs.h"

struct menulist *openlist, *lastmenu;
Boolean MenusOpen = FALSE, MenusWereOpen = FALSE; CrossOn = FALSE;
short savecolor, savemask;
short llx, lly, urx, ury;

/*************************************************************************
  PopupColorInit -  Initializes the portion of the color map which is used
```

```
                    to draw the menus (with a write-mask).

*************************************************************************/
PopupColorInit ()
{
  int i;

  /*  Popup Menus -- Background color  */
  for (i=POPCOLOR1; i<POPCOLOR2; i++)
    mapcolor (i, 0, 0, 0);

  /*  Popup Menus -- Text color  */
  for (i=POPCOLOR2; i<POPCOLOR3; i++)
    mapcolor (i, 255, 255, 255);

  /*  Popup Menus -- Highlight color  */
  for (i=POPCOLOR3; i<POPCOLOR4; i++)
    mapcolor (i, 125, 125, 125);

  /*  Popup Menus -- Active Title bar color  */
  for (i=POPCOLOR4; i<POPCOLOR5; i++)
    mapcolor (i, 255, 0, 0);

  /*  Popup Menus -- Shadowed Text color  */
  for (i=POPCOLOR5; i<ENDCOLOR; i++)
    mapcolor (i, 130, 160, 200);
}

/*************************************************************************
  openmenus -  Sets the graphics system for menu display and opens the
menus.
               Any menus on the menu-stack will become visible.

*************************************************************************/
openmenus ()
{
  struct menulist *menupntr;

  if (!MenusOpen) {
    StoreScreenEnv ();
    MenusOpen = TRUE;
    menupntr = openlist;
    while (menupntr != NULL) {
      popup (menupntr->menu, menupntr->next == NULL);
      menupntr = menupntr->next;
    }
    curson ();
  }
}

/*************************************************************************
  calcbounds -  Calculates the locations of the edges of a menu.  This
```

```
                routine is used internally by the menu system and should
                not be needed by a user of the menu package.

    Arguments:
        menu --    (struct menutype *) pointer to the menu of which to
  calculate
                edge locations.
        menuleft --  (short int *) screen location of left edge of menu.
        menuright --  (short int *) screen location of right edge of menu.
        menutop --  (short int *) screen location of top edge of menu.
        menubottom --   (short int *) screen location of bottom edge of menu.

        menucount --  (short int *) number of choices available in menu.
  ************************************************************************/
  calcbounds (menu, menuleft, menuright, menutop, menubottom, menucount)
  struct menutype *menu;
  short *menuleft, *menuright, *menutop, *menubottom, *menucount;
  {
    *menucount = 0;
    while ((*(((*menu).list)+(*menucount))).type)
      (*menucount)++;
    *menutop = menu->y;
    *menubottom = menu->y - *menucount*16;
    if (*menutop > 751) {
      *menutop = 751;
      *menubottom = *menutop - *menucount*16;
      menu->y = *menutop;
    }
    if (*menubottom < 0) {
      *menubottom = 0;
      *menutop = *menubottom + *menucount*16;
      menu->y = *menutop;
    }
    *menuleft = menu->x;
    *menuright = menu->x + 200;
    if (*menuleft < 0) {
      *menuleft = 0;
      *menuright = 200;
      menu->x = *menuleft;
    }
    if (*menuright > 1023) {
      *menuright = 1023;
      *menuleft = 823;
      menu->x = *menuleft;
    }
  }

  /************************************************************************
    popup -  Displays a menu.  This routine is used internally by the menu
             system and should not be called by a user of the menu package.
```

```
  Arguments:
      menu --  (struct menutype *) pointer to menu to display.
      active --  (Boolean) flag indicating whether this menu is to be
                 displayed as the active menu (i.e. red title bar).

*****************************************************************/
popup(menu, active)
struct menutype *menu;
Boolean active;
{
  register short i;
  short menutop, menubottom, menuleft, menuright, menucount;

  if (MenusOpen) {
    calcbounds (menu, &menuleft, &menuright, &menutop, &menubottom,
&menucount);
    color(POPUPBACKGROUND);          /* menu background */
    cursoff();
    rectfi(menuleft, menubottom, menuright, menutop);
    color(active ? POPUPACTIVE : POPUPTEXT);
    rectfi(menuleft, menutop+1, menuright, menutop+16);
    if (active) {
      color (POPUPTEXT);
      recti (menuleft, menutop+1, menuright, menutop+16);
    }
    color(POPUPBACKGROUND);
    cmov2i(menuleft + 10, menutop + 2);
    charstr (menu->title);
    color(POPUPTEXT);                /* menu text */
    move2i(menuleft, menubottom);
    draw2i(menuleft, menutop);
    draw2i(menuright, menutop);
    draw2i(menuright, menubottom);
    for (i = 0; i < menucount; i++) {
      color(POPUPTEXT);
      move2i(menuleft, menutop - (i+1)*16);
      draw2i(menuright, menutop - (i+1)*16);
      color(menu->list[i].flag ? POPUPTEXT : POPUPSHADOW);
      cmov2i(menuleft + 10, menutop - 14 - i*16);
      charstr((menu->list[i]).text);
    }
  }
}

/*****************************************************************
  shadowpopup -  Displays the outline of a menu.  This is done when a menu

                 is being moved.  This routine is used internally by the
                 menu system and should not be called by a user of the menu

                 package.
```

```
 Arguments:
     menu --  (struct menutype *) pointer to menu whose outline is to be
              displayed.
     active --  (Boolean) flag indicating whether this menu is to be
                displayed as the active menu (i.e. red outline).

*****************************************************************/
shadowpopup(menu, active)
struct menutype *menu;
Boolean active;
{
  register short i;
  short menutop, menubottom, menuleft, menuright, menucount;

  if (MenusOpen) {
    calcbounds  (menu,  &menuleft,  &menuright,  &menutop,  &menubottom,
&menucount);
    color(active ? POPUPACTIVE : POPUPTEXT);
    recti(menuleft, menubottom, menuright, menutop+16);
  }
}
/*****************************************************************
  checkmenu -  This is the workhorse routine of the menu system.  The user
               calls this routine when he wishes to read in one of the menu
               selections.  This routine causes the current selections to
be
               highlighted and returns the value associated with the item
               selected with the press of the right mouse button.  This
               routine also moves a menu when its title bar is selected.
               If this routine is called with the menu-stack empty or if
               the menus are not turned on, checkmenu returns a value of
               zero (0).  (Zero is not a valid "type" entry as it signals
               the end of the popupentry array.)

    Value Returned:  (short int) the "type" entry from the popupenty array
                     associated with the active menu which corresponds to the
                     selection the user makes with the mouse.

*****************************************************************/
checkmenu ()
{
  struct menulist *temp, *menupntr;
  struct menutype *menu, *validmenu;
  short menutop, menubottom, menuleft, menuright, menucount;
  short tmpleft, tmpbottom, tmpright, tmptop, tmpcount;
  short lasthighlight = -1, highlight;
  int dx, dy;
  Device val, x, y;
```

```
  if (MenusOpen && lastmenu != NULL) {
    menu = lastmenu -> menu;
    qreset ();
    tie (RIGHTMOUSE, MOUSEX, MOUSEY);
    setvaluator (MOUSEX, (menu->x)+100, 0, XMAXSCREEN);
    setvaluator (MOUSEY, (menu->y)-8, 0, YMAXSCREEN);
    curson();
    while (1) {
      calcbounds (menu, &menuleft, &menuright, &menutop, &menubottom,
                                                          &menucount);

      x = getvaluator(MOUSEX);
      y = getvaluator(MOUSEY);
      if (menuleft < x && x < menuright && menubottom < y && y < menutop) {

        highlight = (menutop - y)/16;

        if (lasthighlight != -1 && lasthighlight != highlight) {
                    /* not last selection -- turn off old box */
          color(POPUPBACKGROUND);
          cursoff();
          rectfi(menuleft+1, menutop - lasthighlight*16 - 15,
                             menuright-1, menutop - lasthighlight*16 - 1);
          color(menu->list[lasthighlight].flag ? POPUPTEXT : POPUPSHADOW);

          cmov2i(menuleft + 10, menutop - 14 - lasthighlight*16);
          charstr(menu->list[lasthighlight].text);
          curson();
        }
        if (lasthighlight != highlight) {
                  /*  turn on new box   */
          cursoff();
          color(menu->list[highlight].flag && menu->list[highlight].type >
0
               ? POPUPHIGHLIGHT : POPUPBACKGROUND);
          rectfi(menuleft+1, menutop - highlight*16 - 15,
                             menuright-1, menutop - highlight*16 - 1);
          color(menu->list[highlight].flag ? POPUPTEXT : POPUPSHADOW);
          cmov2i(menuleft + 10, menutop - 14 - highlight*16);
          charstr(menu->list[highlight].text);
          curson();
        }
        lasthighlight = highlight;
/*        swapbuffers ();  */
      }
      else /* the cursor is outside the menu */
      {
        if (lasthighlight != -1) {
          cursoff();
          color(POPUPBACKGROUND);
          rectfi(menuleft+1, menutop - lasthighlight*16 - 15,
```

```
                        menuright-1, menutop - lasthighlight*16 - 1);
            color(menu->list[highlight].flag ? POPUPTEXT : POPUPSHADOW);
            cmov2i(menuleft + 10, menutop - 14 - lasthighlight*16);
            charstr(menu->list[highlight].text);
            curson();
            lasthighlight = -1;
/*            swapbuffers ();  */
          }
        }
        if (qtest()) {
          if (qread(&val) == RIGHTMOUSE) {
            qread (&x);
            qread (&y);
            if (val == 1) {
              if (menuleft<x && x<menuright && menubottom<y && y<menutop) {
                color(0);
                cursoff();
                rectfi(menuleft, menubottom, menuright, menutop+16);
                curson();
                x = (menutop - y)/16;
                break;
              }
              else
              {
                /*  menu mover  */
                temp = openlist;
                validmenu = NULL;
                while (temp != NULL) {
                  if (x > temp->menu->x && x < temp->menu->x+200 &&
                            y > temp->menu->y && y < temp->menu->y+16)
                    validmenu = temp->menu;
                  temp = temp->next;
                }
                if (validmenu != NULL) {
                  dx = (int)x - (*validmenu).x;
                  dy = (int)y - (*validmenu).y;
                  color(0);
                  cursoff ();
                  clear ();
                  linewidth (2);
                  while (qtest() ? ((qread(&val)!=RIGHTMOUSE) || (val!=0)) :
1) {
                    color(0);
                    calcbounds (validmenu, &tmpleft, &tmpright, &tmptop,
                                                    &tmpbottom, &tmpcount);

                    gsync ();
                    recti(tmpleft, tmpbottom, tmpright, tmptop+16);
                    validmenu->x = getvaluator (MOUSEX) - dx;
                    validmenu->y = getvaluator (MOUSEY) - dy;
                    menupntr = openlist;
                    while (menupntr != NULL) {
```

```
                        if (menupntr->menu == validmenu)
                          gsync ();
                        shadowpopup (menupntr->menu, menupntr->next == NULL);
                        menupntr = menupntr->next;
/*                      swapbuffers();  */
                      }
                    }
                    linewidth (1);
                    closemenus ();
                    openmenus ();
                  }
                }
              }
            }
          }
        tie (RIGHTMOUSE, 0, 0);
        while (!qtest());
        while (qtest ()) qread(&val);
        return ((menu->list[x].type > 0) && (menu->list[x].flag)) ?
                menu->list[x].type : (popup(menu, TRUE), checkmenu ());
      }
      else
      {
        return 0;
      }
}

/*****************************************************************************
  closemenus -  Turns off the display of menus and returns the graphics
                environment back the way it was before the call to
                "openmenus".

*****************************************************************************/
closemenus ()
{
  if (MenusOpen) {
    MenusOpen = FALSE;
    RetrieveScreenEnv ();
  }
}

/*****************************************************************************
  pushmenu -  Pushes a new menu on the menu-stack.  The new menu becomes
              the active menu.

  Arguments:
      menu --  (struct menutype *) pointer to the menu to place on stack.

*****************************************************************************/
pushmenu (menu)
struct menutype *menu;
```

```
{
  struct menulist *newmenu, *temp;

  if ((newmenu = (struct menulist *) malloc (sizeof(struct
menulist)))==NULL)
    printf ("popmenu:  malloc failed in pushmenu\n");
  newmenu->menu = menu;
  newmenu->next = NULL;
  newmenu->last = lastmenu;
  if (openlist != NULL)
    lastmenu->next = newmenu;
  else
    openlist = newmenu;
  lastmenu = newmenu;
  if (MenusOpen) {
    closemenus ();
    openmenus ();
  }
}

/*************************************************************************
  popmenu -  Removes a menu off the top of the stack.  The new top menu
             becomes the active menu.  A call to this routine when the
menu-
             stack is empty has no effect.

*************************************************************************/
popmenu ()
{
  struct menulist *temp;

  if (openlist) {
    if (lastmenu == openlist) {
      temp = openlist;
      openlist = NULL;
      lastmenu = NULL;
    }
    else
    {
      temp = lastmenu;
      lastmenu = temp->last;
      lastmenu->next = NULL;
      if (MenusOpen) {
        closemenus ();
        openmenus ();
      }
    }
    free (temp);
  }
}

/*************************************************************************
```

```
  TurnOnCross -  Sets up the graphics system for the display of of the
point
                 selection cross-hairs.  If the popup menus were turned on,
                 that fact is noted and they are turned off.

*************************************************************************/
TurnOnCross ()
{
  if (MenusOpen) {
    MenusWereOpen = TRUE;
    closemenus ();
  }
  CrossOn = TRUE;
  StoreScreenEnv ();
}

/*************************************************************************
  TurnOffCross -  Restores the graphics system back the way it was before
                  the cross-hairs were turned on and turns the cross-hairs
                  off.  If menus were turned on before TurnOnCross was
                  called, they are turned on again.

*************************************************************************/
TurnOffCross ()
{
  CrossOn = FALSE;
  RetrieveScreenEnv ();
  if (MenusWereOpen) {
    MenusWereOpen = FALSE;
    openmenus ();
  }
}

/*************************************************************************
  GetLineCross -  Causes the cross-hair to be displayed for the selection
                  of a LINE in three-dimensional space.  The cross-hair
                  follows the mouse pointer until the user clicks one of
                  the buttons.  If the button selected is the one expected,
                  then the current position (screen coordinates) of the
                  cross-hair is returned as parameters and the function
                  takes on the value "TRUE".  If the button selected is not
                  the one expected, the function takes on the value
"FALSE".

  Arguments:
      button --  (Device) the button which the user is expected to press to
                 select the line with.  This parameter should take on one
```

```
                    of the following values:  LEFTMOUSE, MIDDLEMOUSE, or
                    RIGHTMOUSE (from file "/usr/include/device.h".
        xpos, ypos --  (int *) x and y screen coordinates of the cross-hair
                       when the user presses the button specified in
"button".
                       The values returned in these parameters only have
                       meaning if the function returns "TRUE".

  Value Returned:  (int) Actually, this is treated as a Boolean function.
                   "TRUE" is returned if the user selected a line with the

                   button specified in "button".  "FALSE" is returned if
                   either of the other two buttons was pressed.

**************************************************************************/
GetLineCross (button, xpos, ypos)
Device button;
int *xpos, *ypos;
{
  Device mx, my, oldmx, oldmy, val;
  int wind;
  Boolean changed;

  if (CrossOn) {
    changed = TRUE;
    mx = (int) getvaluator (MOUSEX);
    my = (int) getvaluator (MOUSEY);
    do {
      if ((wind = WhichWindow (mx, my)) != -1 && changed) {
        cursoff ();
        color (0);
        clear ();
        DrawPrimaryCross (wind, mx, my);
        curson ();
      }
      oldmx = mx;
      oldmy = my;
      mx = (int) getvaluator (MOUSEX);
      my = (int) getvaluator (MOUSEY);
      changed = (mx != oldmx) || (my != oldmy);
    } while (!qtest());
    if (qread(&val) == button) {
      *xpos = mx;
      *ypos = my;
      while (!qtest());
      qreset();
      return TRUE;
    }
    else
    {
      while (!qtest());
      qreset();
```

```
        return FALSE;
      }
    }
  }

/*****************************************************************
  GetPointCross -  Causes the cross-hair to be displayed for the selection

                   of a POINT in three-dimensional space.  The cross-hair
                   follows the mouse pointer until the user clicks one of
                   the buttons.    If the button selected is the one
expected,
                   then the current position (screen coordinates) of the
                   cross-hair is returned as paramenters and the function
                   takes on the value "TRUE".   If the button selected is
not
                   the one expected, the function takes on the value
"FALSE".

                   This routine assumes that the point to be selected has
                   already been narrowed-down to a line by the use of
                   "GetLineCross".  The coordinates of the line returned
                   from "GetLineCross" may be passed to "GetPointCross"
                   so that a single point on the line in space may be
                   selected.

    Arguments:
        button --  (Device) the button which the user is expected to press to

                   select the line with.  This parameter should take on one
                   of the following values:  LEFTMOUSE, MIDDLEMOUSE, or
                   RIGTHMOUSE (from file "/usr/include/device.h".
        linex, liney --  (int)  x and y screen coordinates of the point which

                         corresponds to a line in 3d space.  This is the line

                         from which the final point returned is expected.
        xpos, ypos --  (int *) x and y screen coordinates of the point
                       selected on the line defined by "linex" and "liney".
                       When these two parameters are taken with "linex" and
                       "liney", a single point in 3d space may be computed.

    Value Returned:  (int) Actually, this is treated as a Boolean function.
                     "TRUE" is returned if the user selected a line with the

                     buton specified AND the mouse pointer was in a window
                     in which the line from which the point was to be
selected
                     does not appear to be a point.  If either of these
                     criteria are not met, "FALES" is returned.

*****************************************************************/
```

```
GetPointCross (button, linex, liney, xpos, ypos)
Device button;
int linex, liney, *xpos, *ypos;
{
  Device mx, my, oldmx, oldmy, val;
  int wind, primewind;
  Boolean changed;

  if (CrossOn) {
    primewind = WhichWindow (linex, liney);
    changed = TRUE;
    mx = (int) getvaluator (MOUSEX);
    my = (int) getvaluator (MOUSEY);
    do {
      wind = WhichWindow (mx, my);
      if (wind != -1 && wind != primewind && changed) {
        cursoff ();
        color (0);
        clear ();
        DrawPrimaryCross (primewind, linex, liney);
        DrawSecondaryCross (primewind, wind, mx, my);
        curson ();
      }
      oldmx = mx;
      oldmy = my;
      mx = (int) getvaluator (MOUSEX);
      my = (int) getvaluator (MOUSEY);
      changed = (mx != oldmx) || (my != oldmy);
    } while (!qtest());
    if (qread(&val) == button) {
      *xpos = mx;
      *ypos = my;
      while (!qtest());
      qreset ();
      return TRUE;
    }
    else
    {
      while (!qtest());
      qreset ();
      return FALSE;
    }
  }
}

/*****************************************************************************
  DrawPrimaryCross -  Draws the primary cross for use in selecting a line.

  Arguments:
      wind --  (int) window in which the primary cross is to appear.
      mx, my --  (int) the x and y screen coordinates of the cross-hair.
```

```
*****************************************************************/
DrawPrimaryCross (wind, mx, my)
int wind, mx, my;
{
  int left, right, bottom, top, other;
  float ratio;

  if (wind != -1) {
    color (CROSSCOLOR);
    switch (wind) {
      case (FrontView) :
        GetWindowSides (FrontView, &left, &right, &bottom, &top);
        move2i (mx, top);
        gsync ();
        draw2i (mx, bottom);
        move2i (left, my);
        draw2i (right, my);
        GetWindowSides (TopView, &left, &right, &bottom, &top);
        move2i (mx, top);
        gsync ();
        draw2i (mx, bottom);
        GetWindowSides (SideView, &left, &right, &bottom, &top);
        move2i (left, my);
        gsync ();
        draw2i (right, my);
        break;
      case (SideView) :
        GetWindowSides (SideView, &left, &right, &bottom, &top);
        move2i (mx, top);
        gsync ();
        draw2i (mx, bottom);
        move2i (left, my);
        draw2i (right, my);
        ratio = (float)(mx-left)/(float)(right-left);
        GetWindowSides (FrontView, &left, &right, &bottom, &top);
        move2i (left, my);
        gsync ();
        draw2i (right, my);
        GetWindowSides (TopView, &left, &right, &bottom, &top);
        other = bottom + (int)(ratio * (float)(top-bottom));
        move2i (left, other);
        gsync ();
        draw2i (right, other);
        break;
      case (TopView) :
        GetWindowSides (TopView, &left, &right, &bottom, &top);
        move2i (mx, top);
        gsync ();
        draw2i (mx, bottom);
        move2i (left, my);
        draw2i (right, my);
```

```
            ratio = (float)(my-bottom)/(float)(top-bottom);
            GetWindowSides (FrontView, &left, &right, &bottom, &top);
            move2i (mx, top);
            gsync ();
            draw2i (mx, bottom);
            GetWindowSides (SideView, &left, &right, &bottom, &top);
            other = left + (int)(ratio * (float)(right-left));
            move2i (other, top);
            gsync ();
            draw2i (other, bottom);
            break;
          default :
            break;
        }
      }
    }

    /*************************************************************************
      DrawSecondaryCross -  Draws the secondary cross for use in selecting a
                            point.

      Arguments:
          primewind --  (int) window in which the primary cross is located.
          mx, my --  (int) the x and y screen coordinates of the secondary
                     cross-hair.

    *************************************************************************/
    DrawSecondaryCross (primewind, wind, mx, my)
    int primewind, wind, mx, my;
    {
      int left, right, bottom, top, other;
      float ratio;

      if (wind != -1 && primewind != -1) {
        color (CROSSCOLOR);
        switch (primewind) {
          case (FrontView) :
            switch (wind) {
              case (SideView) :
                GetWindowSides (SideView, &left, &right, &bottom, &top);
                move2i (mx, top);
                gsync ();
                draw2i (mx, bottom);
                ratio = (float)(mx-left)/(float)(right-left);
                GetWindowSides (TopView, &left, &right, &bottom, &top);
                other = bottom + (int)(ratio * (float)(top-bottom));
                move2i (left, other);
                gsync ();
                draw2i (right, other);
                break;
              case (TopView) :
                GetWindowSides (TopView, &left, &right, &bottom, &top);
```

```
                move2i (left, my);
                gsync ();
                draw2i (right, my);
                ratio = (float)(my-bottom)/(float)(top-bottom);
                GetWindowSides (SideView, &left, &right, &bottom, &top);
                other = left + (int)(ratio * (float)(right-left));
                move2i (other, top);
                gsync ();
                draw2i (other, bottom);
                break;
              default :
                break;
            }
            break;
          case (SideView) :
            GetWindowSides (FrontView, &left, &right, &bottom, &top);
            move2i (mx, top);
            gsync ();
            draw2i (mx, bottom);
            GetWindowSides (TopView, &left, &right, &bottom, &top);
            move2i (mx, top);
            gsync ();
            draw2i (mx, bottom);
            break;
          case (TopView) :
            GetWindowSides (FrontView, &left, &right, &bottom, &top);
            move2i (left, my);
            gsync ();
            draw2i (right, my);
            GetWindowSides (SideView, &left, &right, &bottom, &top);
            move2i (left, my);
            gsync ();
            draw2i (right, my);
            break;
          default :
            break;
        }
      }
    }

    /*****************************************************************
      StoreScreenEnv -  Stores the current graphic environment so popup menus
                        or cross-hairs can be drawn.

    *****************************************************************/
    StoreScreenEnv ()
    {
      zbuffer (FALSE);
      zclear ();
      savecolor = getcolor ();
      savemask = getwritemask();
      getviewport (&llx, &urx, &lly, &ury);
```

```
    pushmatrix ();
    viewport (0, 1023, 0, 767);
    ortho2 (-0.5, 1023.5, -0.5, 767.5);
    writemask (MASKVALUE);
}

/*****************************************************************************
    RetrieveScreenEnv -  Retrieves a stored graphic environment after popup
                         menus or cross-hairs have been displayed.
*****************************************************************************/
RetrieveScreenEnv ()
{
    cursoff ();
    color (0);
    clear ();
    popmatrix ();
    color (savecolor);
    writemask (savemask);
    viewport (llx, urx, lly, ury);
    zbuffer (TRUE);
}
```

Appendix A.6 Primitive Rigid Body Generator

```
/*****************************************************************************
  Filename:  generator.c

  by Tim Thompson

  Purpose:  This package is used to generate several different types of
            rigid bodies for storage in the winged edge database.  These
            primitive objects include:  cylinders, pipes, spheres,
            parallelepipeds, boxes, cubes, and cones.

            While the rigid bodies generated my have any color and may be
            any size, their orientations are always the same.  To change
            the orientations, a transform should be added to the object
        which contains the rigid body.

  Functions Provided:  BuildColorMap ()
                       GenBox ()
                       GenCone ()
                       GenCube ()
                       GenCylinder ()
                       GenParallelepiped ()
                       GenPipe ()
                       GenSphere ()

*****************************************************************************/

#include "defs.h"
#include "dbdefs.h"
/*****************************************************************************
  GenCylinder -  Generates a winged edge database representation of a
                 cylinder.  The cylinder is oriented with its axis along
                 the y-axis.  The base of the cylinder is in the plane
                 y=0.  The top of the cylinder is in the plane y=height.
                 The cylinder will be represented with "polys" number of
                 polygons around the outer edge.

  Arguments:
      colr --   (int) base color value (should be one of the base values
                defined in file "defs.h".
      polys --  (int) number of polygons to use to represent the outside
                (curved part) of the cylinder.
      radius --  (float) radius of the cylinder. (The cylinder will be
                 inscribed within this radius.)
      height --  (float) height of the cylinder or the distance it will
                 extend along the y-axis.
      flags --  (int) reserved for future use.

  Value Returned:  (OBJECT *) pointer to the new cylinder.

*****************************************************************************/
```

```
OBJECT *GenCylinder (colr, polys, radius, height, flags)
float radius, height;
int colr, polys, flags;
{
  float inc, x1, x2, z1, z2, cpi, cpj, cpk, length, ni, nk;
  register cnt, i;
  VERTEX *vtxlow[MaxPolys], *vtxhigh[MaxPolys];
  CORNER *corn;
  ATTRIBUTE *attr;
  OBJECT *RigidBody;

  RigidBody = NewRb(UniqueRbNum ());

  inc = (2.0*PI/(float)polys);
  x1 = radius;
  z1 = 0.0;
  for (i=0; i<polys; i++) {
    vtxlow[i] = NewVertex ();
    vtxhigh[i] = NewVertex ();
  }
  SetVertex (vtxlow[0], x1, 0.0, z1);
  SetVertex (vtxhigh[0], x1, height, z1);

  cpi = sin(inc)*radius;
  cpk = radius-cos(inc)*radius;
  length = sqrt(cpi*cpi + cpk*cpk);
  for (cnt=1; cnt<=polys; cnt++) {
    x2 = cos((float)cnt*inc)*radius;
    z2 = sin((float)cnt*inc)*radius;
    if (cnt < polys) {
      SetVertex (vtxlow[cnt], x2, 0.0, z2);
      SetVertex (vtxhigh[cnt], x2, height, z2);
    }
    cpi = z2 - z1;
    cpk = x1 - x2;

    ni = cpi/length;
    nk = cpk/length;

    attr = NewAttribute ();
    SetAttribute (attr, ni, 0.0, nk, colr, flags);

    corn = NewCorn();
    SetCorner (corn, vtxhigh[cnt-1]);
    AddCorner (RigidBody, corn);

    corn = NewCorn();
    SetCorner (corn, vtxhigh[cnt<polys ? cnt:0]);
    AddCorner (RigidBody, corn);

    corn = NewCorn();
    SetCorner (corn, vtxlow[cnt<polys ? cnt:0]);
```

```
        AddCorner (RigidBody, corn);

        corn = NewCorn();
        SetCorner (corn, vtxlow[cnt-1]);
        AddCorner (RigidBody, corn);

        AddPolygon (RigidBody, attr);

        x1 = x2;
        z1 = z2;
      }
      attr = NewAttribute ();
      SetAttribute (attr, 0.0, 1.0, 0.0, colr, flags);
      for (cnt = polys - 1; cnt >= 0; cnt--) {
        corn = NewCorn ();
        SetCorner (corn, vtxhigh[cnt]);
        AddCorner (RigidBody, corn);
      }
      AddPolygon (RigidBody, attr);
      attr = NewAttribute ();
      SetAttribute (attr, 0.0, -1.0, 0.0, colr, flags);

      for (cnt = 0; cnt < polys; cnt++) {
        corn = NewCorn ();
        SetCorner (corn, vtxlow[cnt]);
        AddCorner (RigidBody, corn);
      }
      AddPolygon (RigidBody, attr);

      return RigidBody;
    }

    /*************************************************************************
      GenPipe -  Generates a winged edge database representation of a pipe.
                 (A pipe is a hollow cylinder.)  The pipe is oriented with its

                 axis along the y-axis.  The base of the pipe is in the
                 plane y=0.  The top of the pipe is in the plane y=height.
                 The pipe will be represented with "polys" number of polygons
                 around the outer and inner edges.

      Arguments:
          colr --  (int) base color value (should be one of the base values
                   defined in file "defs.h".
          polys --  (ind) number of polygons to use to represent the curved
                    parts of the pipe.
          outradius --  (float) outer radius of the pipe.
          inradius -- (float) inner radius of the pipe.
          height -- (float) height of the pipe or the distance it will extend
                    along the y-axis.
          flags --  (int) reserved for future use.
```

```
  Value Returned:  (OBJECT *) pointer to the new pipe.

*************************************************************************/
OBJECT *GenPipe (colr, polys, outradius, inradius, height, flags)
int colr, polys, flags;
float outradius, inradius, height;
{
  float cosval1, cosval2, sinval1, sinval2,
        xout1, xout2, zout1, zout2, xin1, xin2, zin1, zin2,
        cpi, cpk, length, ni, nk, inc;
  register cnt, i;

  VERTEX *vhighin[MaxPolys], *vhighout[MaxPolys], *vlowin[MaxPolys],
         *vlowout[MaxPolys];
  CORNER *corn;
  ATTRIBUTE *attr;
  OBJECT *RigidBody;

  inc = 2.0 * PI/(float)polys;

  RigidBody = NewRb(UniqueRbNum ());

  cosval1 = 1.0;
  sinval1 = 0.0;
  xout1 = outradius;
  zout1 = 0.0;
  xin1 = inradius;
  zin1 = 0.0;

  for (i=0; i<polys; i++) {
    vhighin[i] = NewVertex();
    vhighout[i] = NewVertex();
    vlowin[i] = NewVertex();
    vlowout[i] = NewVertex();
  }

  SetVertex (vhighin[0] , xin1 , height, zin1 );
  SetVertex (vhighout[0], xout1, height, zout1);
  SetVertex (vlowin[0]  , xin1 , 0.0   , zin1 );
  SetVertex (vlowout[0] , xout1, 0.0   , zout1);

  cpi = sin(inc)*outradius;
  cpk = outradius - cos(inc)*outradius;

  length = sqrt (cpi*cpi + cpk*cpk);

  for (cnt=1; cnt<=polys; cnt++) {
    cosval2 = cos ((float)cnt * inc);
    sinval2 = sin ((float)cnt * inc);

    xout2 = cosval2 * outradius;
    zout2 = sinval2 * outradius;
```

```
xin2  = cosval2 * inradius;
zin2  = sinval2 * inradius;

if (cnt < polys) {
  SetVertex (vhighin[cnt] , xin2 , height, zin2 );
  SetVertex (vhighout[cnt], xout2, height, zout2);
  SetVertex (vlowin[cnt]  , xin2 , 0.0   , zin2 );
  SetVertex (vlowout[cnt] , xout2, 0.0   , zout2);
}

cpi = zout2 - zout1;
cpk = xout1 - xout2;

ni = cpi/length;
nk = cpk/length;

attr = NewAttribute ();
SetAttribute (attr, ni, 0.0, nk, colr, flags);

corn = NewCorn ();
SetCorner (corn, vlowout[cnt-1]);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vhighout[cnt-1]);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vhighout[cnt<polys ? cnt : 0]);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vlowout[cnt<polys ? cnt : 0]);
AddCorner (RigidBody, corn);

AddPolygon (RigidBody, attr);

attr = NewAttribute ();
SetAttribute (attr, -ni, 0.0, -nk, colr, flags);

corn = NewCorn ();
SetCorner (corn, vlowin[cnt-1]);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vlowin[cnt<polys ? cnt : 0]);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vhighin[cnt<polys ? cnt : 0]);
AddCorner (RigidBody, corn);
```

```
        corn = NewCorn ();
        SetCorner (corn, vhighin[cnt-1]);
        AddCorner (RigidBody, corn);

        AddPolygon (RigidBody, attr);

        attr = NewAttribute ();
        SetAttribute (attr, 0.0, 1.0, 0.0, colr, flags);

        corn = NewCorn ();
        SetCorner (corn, vhighout[cnt-1]);
        AddCorner (RigidBody, corn);

        corn = NewCorn ();
        SetCorner (corn, vhighin[cnt-1]);
        AddCorner (RigidBody, corn);

        corn = NewCorn ();
        SetCorner (corn, vhighin[cnt<polys ? cnt : 0]);
        AddCorner (RigidBody, corn);

        corn = NewCorn ();
        SetCorner (corn, vhighout[cnt<polys ? cnt : 0]);
        AddCorner (RigidBody, corn);

        AddPolygon (RigidBody, attr);

        attr = NewAttribute ();
        SetAttribute (attr, 0.0, -1.0, 0.0, colr, flags);

        corn = NewCorn ();
        SetCorner (corn, vlowout[cnt-1]);
        AddCorner (RigidBody, corn);

        corn = NewCorn ();
        SetCorner (corn, vlowout[cnt<polys ? cnt : 0]);
        AddCorner (RigidBody, corn);

        corn = NewCorn ();
        SetCorner (corn, vlowin[cnt<polys ? cnt : 0]);
        AddCorner (RigidBody, corn);

        corn = NewCorn ();
        SetCorner (corn, vlowin[cnt-1]);
        AddCorner (RigidBody, corn);

        AddPolygon (RigidBody, attr);

        xout1 = xout2;
        zout1 = zout2;
    }
```

```
    return RigidBody;
}

/*************************************************************************
  GenSphere -  Generates a winged edge database representation of a sphere.
               The sphere is oriented with its center located at the
origin.
               The sphere will be represented with "polys" number of
polygons
               around any "latitude" line as well as around any "longitude"
               line.

  Arguments:
      colr --  (int) base color value (should be one of the base values
               defined in file "defs.h".
      polys --  (int) number of polygons to use to represent any "slice" of
                the sphere between two lines of "latitude".
      radius -- (float) the radius of the sphere.  (The sphere will be
                inscribed within this radius.)
      flags --  (int) reserved for future use.

  Value Returned:  (OBJECT *) pointer to the new sphere.

*************************************************************************/
OBJECT *GenSphere (colr, polys, radius, flags)
int colr, polys, flags;
float radius;
{
  register lat, longitude;
  float xlow1,xlow2,xhigh1,xhigh2,ylow,yhigh,zlow1,zlow2,zhigh1,zhigh2;
  float cpi,cpj,cpk,length,ni,nj,nk;
  float inc;radlow,radhigh;
  register FirstFlag, LastFlag;

  VERTEX *vtx[MaxPolys], *vtxa, *vtxb, *vtx0;
  CORNER *corn;
  ATTRIBUTE *attr;
  OBJECT *RigidBody;

  RigidBody = NewRb(UniqueRbNum ());

  FirstFlag = TRUE;
  LastFlag = FALSE;
  inc=(2.0*PI/(float)polys);
  for (lat=(int)(-(polys/4)); lat<(int)(polys/4); lat++) {
    ylow=sin((float)(lat)*inc)*radius;
    yhigh = sin((float)(lat+1)*inc)*radius;
    radlow = cos((float)(lat)*inc)*radius;
    radhigh = cos((float)(lat+1)*inc)*radius;
```

```
        if (lat==(int)(polys/4)-1) {
          LastFlag = TRUE;
          vtxa = NewVertex();
        }
        if (FirstFlag) {
          vtxa = NewVertex();
          SetVertex (vtxa, 0.0, ylow, 0.0);
          for (longitude=0; longitude < polys; longitude++)
            vtx[longitude] = vtxa;
        }
        for (longitude=0; longitude < polys; longitude++) {
          xlow1 = cos((float)longitude*inc)*radlow;
          xlow2 = cos((float)(longitude+1)*inc)*radlow;
          zlow1 = sin((float)longitude*inc)*radlow;
          zlow2 = sin((float)(longitude+1)*inc)*radlow;
          xhigh1 = cos((float)longitude*inc)*radhigh;
          xhigh2 = cos((float)(longitude+1)*inc)*radhigh;
          zhigh1 = sin((float)longitude*inc)*radhigh;
          zhigh2 = sin((float)(longitude+1)*inc)*radhigh;

          if (longitude == 0){
            vtx0 = vtx[0];
          }

          if (!LastFlag) {
            if (longitude == 0) {
              vtxa = NewVertex ();
              SetVertex (vtxa, xhigh1, yhigh, zhigh1);
            }
            vtxb = vtxa;
          }
          if (longitude < polys-1) {
            if (!LastFlag) {
              vtxa = NewVertex ();
            }
            SetVertex (vtxa, xhigh2, yhigh, zhigh2);
          }
          else
          {
            if (!LastFlag) {
              vtxa = vtx[0];
            }
          }

          if (FirstFlag) {
            cpi = (yhigh-ylow)*((zhigh2-zlow1)-(zhigh1-zlow1));
            cpj =  (zhigh1-zlow1)*(xhigh2-xlow1)-(xhigh1-xlow1)*(zhigh2-zlow1);

            cpk = (yhigh-ylow)*((xhigh1-xlow1)-(xhigh2-xlow1));
          }
          else
          {
```

```
        cpi = (yhigh-ylow)*(zlow2-zlow1);
        cpj = (zhigh1-zlow1)*(xlow2-xlow1)-(xhigh1-xlow1)*(zlow2-zlow1);
        cpk = (ylow-yhigh)*(xlow2-xlow1);
      }

      length = sqrt(cpi*cpi+cpj*cpj+cpk*cpk);

      ni = cpi/length;
      nj = cpj/length;
      nk = cpk/length;

      attr = NewAttribute ();
      SetAttribute (attr, ni, nj, nk, colr, flags);

      corn = NewCorn();
      SetCorner (corn, vtxa);
      AddCorner (RigidBody, corn);

      if (!LastFlag) {
        corn = NewCorn();
        SetCorner (corn, vtxb);
        AddCorner (RigidBody, corn);
      }

      corn = NewCorn();
      SetCorner (corn, vtx[longitude]);
      AddCorner (RigidBody, corn);

      if (!FirstFlag) {
        corn = NewCorn();
        SetCorner (corn, longitude<(polys-1) ? vtx[longitude+1] : vtx0);
        AddCorner (RigidBody, corn);
      }

      AddPolygon (RigidBody, attr);
      if (!LastFlag) {
        vtx[longitude] = vtxb;
      }
    }
    FirstFlag = FALSE;
  }
  return RigidBody;
}

/*****************************************************************
  GenParallelepiped -  Generates a winged edge database representation of a

                       parallelepiped.  The parallelepiped is oriented such

                       that one corner is always at the origin with the
                       sides of the parallelepiped extending down the
                       positive x, y, and z axes.  The "top" and "bottom"
```

```
                        of the parallelepiped are always parallel with the
                        x-z plane.  The angle that the other sides make with

                        respect to the x-y plane and the y-z plane are
                        accepted as arguments.

    Arguments:
        colr --  (int) base color value (should be one of the base values
                 defined in file "defs.h".
        length --  (float) distance the parallelepiped will extend along the

                   positive x axis.
        width -- (float) distance the parallelepiped will extend along the
                 positive z axis.
        height --  (float) distance the parallelepiped will extend along the

                   positive y axis.
        xyangle -- (float) angle the side of the parallelepiped makes with
                   the x-y plane.
        yzangle -- (float) angle the side of the parallelepiped makes with
                   the y-z plane.
        flags --  (int) reserved for future use.

    Value Returned:  (OBJECT *) pointer to the new parallelepiped.

*************************************************************************/
OBJECT *GenParallelepiped(colr, length, width, height, xyangle, yzangle,
flags)
int colr, flags;
float length, width, height, xyangle, yzangle;
{
    float cpi, cpj, cpk, ni, nj, nk, vlength, xshift, zshift;
    VERTEX *vtx_000, *vtx_0y0, *vtx_xy0, *vtx_x00,
           *vtx_00z, *vtx_0yz, *vtx_xyz, *vtx_x0z;
    CORNER *corn;
    ATTRIBUTE *attr;
    OBJECT *RigidBody;

    RigidBody = NewRb(UniqueRbNum());

    xshift = height * tan(yzangle*PI/180.0);
    zshift = height * tan(xyangle*PI/180.0);

    vtx_000 = NewVertex ();
    vtx_0y0 = NewVertex ();
    vtx_xy0 = NewVertex ();
    vtx_x00 = NewVertex ();
    vtx_00z = NewVertex ();
    vtx_0yz = NewVertex ();
    vtx_xyz = NewVertex ();
    vtx_x0z = NewVertex ();
```

```
SetVertex (vtx_000, 0.0, 0.0, 0.0);
SetVertex (vtx_0y0, xshift, height, zshift);
SetVertex (vtx_xy0, length+xshift, height, zshift);
SetVertex (vtx_x00, length, 0.0, 0.0);
SetVertex (vtx_00z, 0.0, 0.0, width);
SetVertex (vtx_0yz, xshift, height, width+zshift);
SetVertex (vtx_xyz, length+xshift, height, width+zshift);
SetVertex (vtx_x0z, length, 0.0, width);

cpj = -tan(xyangle*PI/180.0);
vlength = sqrt(cpj*cpj + 1.0);

nj = cpj/vlength;
nk = 1.0/vlength;

/* Generate Front */
attr = NewAttribute ();
SetAttribute (attr, 0.0, nj, nk, colr, flags);

corn = NewCorn ();
SetCorner (corn, vtx_00z);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_0yz);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_xyz);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_x0z);
AddCorner (RigidBody, corn);

AddPolygon (RigidBody, attr);

/* Generate Back */
attr = NewAttribute ();
SetAttribute (attr, 0.0, -nj, -nk, colr, flags);

corn = NewCorn ();
SetCorner (corn, vtx_000);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_x00);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_xy0);
AddCorner (RigidBody, corn);
```

```
corn = NewCorn ();
SetCorner (corn, vtx_0y0);
AddCorner (RigidBody, corn);

AddPolygon (RigidBody, attr);

cpj = -tan(yzangle*PI/180.0);
vlength = sqrt(1.0 + cpj*cpj);

ni = 1.0/vlength;
nj = cpj/vlength;

/* Generate Right End */
attr = NewAttribute ();
SetAttribute (attr, ni, nj, 0.0, colr, flags);

corn = NewCorn ();
SetCorner (corn, vtx_x0z);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_xyz);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_xy0);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_x00);
AddCorner (RigidBody, corn);

AddPolygon (RigidBody, attr);

/* Generate Left End */
attr = NewAttribute ();
SetAttribute (attr, -ni, -nj, 0.0, colr, flags);

corn = NewCorn ();
SetCorner (corn, vtx_000);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_0y0);
AddCorner (RigidBody, corn);

corn = NewCorn ();
SetCorner (corn, vtx_0yz);
AddCorner (RigidBody, corn);

corn = NewCorn ();
```

```
    SetCorner (corn, vtx_00z);
    AddCorner (RigidBody, corn);

    AddPolygon (RigidBody, attr);

    /* Generate Top */
    attr = NewAttribute ();
    SetAttribute (attr, 0.0, 1.0, 0.0, colr, flags);

    corn = NewCorn ();
    SetCorner (corn, vtx_0y0);
    AddCorner (RigidBody, corn);

    corn = NewCorn ();
    SetCorner (corn, vtx_xy0);
    AddCorner (RigidBody, corn);

    corn = NewCorn ();
    SetCorner (corn, vtx_xyz);
    AddCorner (RigidBody, corn);

    corn = NewCorn ();
    SetCorner (corn, vtx_0yz);
    AddCorner (RigidBody, corn);

    AddPolygon (RigidBody, attr);

    /* Generate Bottom */
    attr = NewAttribute ();
    SetAttribute (attr, 0.0, -1.0, 0.0, colr, flags);

    corn = NewCorn ();
    SetCorner (corn, vtx_000);
    AddCorner (RigidBody, corn);

    corn = NewCorn ();
    SetCorner (corn, vtx_00z);
    AddCorner (RigidBody, corn);

    corn = NewCorn ();
    SetCorner (corn, vtx_x0z);
    AddCorner (RigidBody, corn);

    corn = NewCorn ();
    SetCorner (corn, vtx_x00);
    AddCorner (RigidBody, corn);

    AddPolygon (RigidBody, attr);

    return RigidBody;
}
```

```
/*****************************************************************
  GenBox -  Generates a winged edge database representation of a box.
            The box is oriented such that one corner is always at the
            origin with the sides of the box extending down the positive
            x, y, and z axes.

  Arguments:
      colr --  (int) base color value (should be one of the base values
               defined in file "defs.h".
      length --  (float) distance the box will extend along the positive
                 x axis.
      width --  (float) distance the box will extend along the positive
                z axis.
      height --  (float) distance the box will extend along the positive
                 y axis.
      flags --  (int) reserved for future use.

  Value Returned:  (OBJECT *) pointer to the new box.

*****************************************************************/
OBJECT *GenBox (colr, length, width, height, flags)
int colr, flags;
float length, width, height;
{
  return GenParallelepiped (colr, length, width, height, 0.0, 0.0, flags);
}

/*****************************************************************
  GenCube -  Generates a winged edge database representation of a cube.
             The cube is oriented such that one corner is always at the
             origin with the sides of the cube extending down the positive

             x, y, and z axes.

  Arguments:
      colr --  (int) base color value (should be one of the base values
               defined in file "defs.h".
      length --  (float) distance the cube will extend along each of the
                 positive x, y, and z axes.
      flags --  (int) reserved for future use.

  Value Returned:  (OBJECT *) pointer to the new cube.

*****************************************************************/
OBJECT *GenCube (colr, length, flags)
int colr, flags;
float length;
{
  return GenParallelepiped (colr, length, length, length, 0.0, 0.0, flags);
}
```

```
/*************************************************************************
  GenCone -  Generates a winged edge database representation of a cone.
The
             cone is oriented with its axis along the y-axis.  The base of

             the cone is in the plane y=0.  The tip of the cone is located

             at the point y=height.  The cone is represented with "polys"
             number of polygons around the outer (curved) side.

  Arguments:
      colr --  (int) base color value (should be one of the base values
               defined in file "defs.h".
      polys -- (int) number of polygons to use to represent the outside
               (curved part) of the cone.
      radius --  (float) radius of the base of the cone.
      height --  (float) height of the cone or the distance it will extend

                 along the y-axis.
      flags --  (int) reserved for future use.

  Value Returned:  (OBJECT *) pointer to the new cone.

*************************************************************************/
OBJECT *GenCone (colr, polys, radius, height, flags)
int colr, polys, flags;
float radius, height;
{
  register cnt;
  float inc, cpi, cpj, cpk, length, x1, x2, z1, z2;
  VERTEX *vtx[MaxPolys], *top;
  CORNER *corn;
  ATTRIBUTE *attr;
  OBJECT *RigidBody;

  RigidBody = NewRb(UniqueRbNum ());

  inc = (2.0*PI/(float)polys);
  cpi = height * radius * sin(inc);
  cpj = radius * radius * sin(inc);
  cpk = height * (radius - cos(inc)*radius);

  length = sqrt (cpi*cpi + cpj*cpj + cpk*cpk);

  vtx[0] = NewVertex();
  SetVertex (vtx[0], radius, 0.0, 0.0);
  top = NewVertex();
  SetVertex (top, 0.0, height, 0.0);

  for (cnt=0; cnt < polys; cnt++) {
    x1 = cos ((float)cnt * inc) * radius;
```

```
        x2 = cos ((float)(cnt+1) * inc) * radius;
        z1 = sin ((float)cnt * inc) * radius;
        z2 = sin ((float)(cnt+1) * inc) * radius;

        cpi = height * (z2 - z1) / length;
        cpj = (x1*z2 - x2*z1) / length;
        cpk = height * (x1 - x2) / length;

        attr = NewAttribute ();
        SetAttribute (attr, cpi, cpj, cpk, colr, flags);

        if (cnt<polys-1) {
          vtx[cnt+1] = NewVertex();
          SetVertex (vtx[cnt+1], x2, 0.0, z2);
        }

        corn = NewCorn();
        SetCorner (corn, top);
        AddCorner (RigidBody, corn);

        corn = NewCorn();
        SetCorner (corn, vtx[cnt]);
        AddCorner (RigidBody, corn);

        corn = NewCorn();
        SetCorner (corn, cnt<(polys-1) ? vtx[cnt+1] : vtx[0]);
        AddCorner (RigidBody, corn);

        AddPolygon (RigidBody, attr);
      }

      attr = NewAttribute();
      SetAttribute (attr, 0.0, -1.0, 0.0, colr, flags);

      for (cnt=polys-1; cnt>=0; cnt--) {
        corn = NewCorn();
        SetCorner (corn, vtx[cnt]);
        AddCorner (RigidBody, corn);
      }
      AddPolygon (RigidBody, attr);
      return RigidBody;
    }
    /*
    GenTriangle (colr, x1, y1, z1, x2, y2, z2, x3, y3, z3, sv, v)
    int colr;
    float x1, y1, z1, x2, y2, z2, x3, y3, z3;
    vector sv, v;
    {
      float ax, ay, az, bx, by, bz, cpi, cpj, cpk, length, shade;
      pushmatrix ();
      scale (sv.i, sv.j, sv.k);
      ax = x2-x1;
```

```
    ay = y2-y1;
    az = z2-z1;

    bx = x3-x1;
    by = y3-y1;
    bz = z3-z1;

    cpi = ay*bz - az*by;
    cpj = az*bx - ax*bz;
    cpk = ax*by - ay*bx;

    length = sqrt(cpi*cpi + cpj*cpj + cpk*cpk);

    shade = (cpi*v.i + cpj*v.j + cpk*v.k)/length*(float)(span-1);

    color (colr + abs((int)shade));
    pmv (x1, y1, z1);
    pdr (x2, y2, z2);
    pdr (x3, y3, z3);
    pclos ();

    popmatrix ();
}

GenParallelogram (colr, x1, y1, z1, x2, y2, z2, x3, y3, z3, sv, v)
int colr;
float x1, y1, z1, x2, y2, z2, x3, y3, z3;
vector sv, v;
{
  Genriangle (colr, x1, y1, z1, x2, y2, z2, x3, y3, z3, sv, v);
  GenTriangle (colr, x2-x1+x3, y2-y1+y3, z2-z1+z3, x3, y3, z3,
                  x2, y2, z2, sv, v);
}

GenRectangle (colr, x1, y1, x4, y4, sv,  v)
int colr;
float x1, y1, x4, y4;
vector sv, v;
{
  GenParallelogram (colr, x1, y1, 0.0, x4, y1, 0.0, x1, y4, 0.0, sv, v);
}
GenSquare (colr, x, y, length, sv, v)
int colr;
float x, y, length;
vector v;
{
  GenRectangle (colr, x, y, x+length, y+length, sv, v);
}
*/
```

Appendix A.7 Vector Manipulation Functions

```
/*************************************************************************
  Filename:  vectors.c

 by Timothy A. Thompson

  Purpose:  The purpose of this package is to provide a set of vector
            manipulation functions.  These functions include the rotation
            and translation of vectors and of the global viewing angle.
            This package also implements a stack of vectors so the viewing
            vector can be saved during these transformations.

  Functions Provided:  PopAll ()
                       PopVector ()
                       PushAll ()
                       PushVector ()
                       RotateAll ()
                       RotateMulti ()
                       RotateMultiEnv ()
                       RotateVector ()
                       TranslateMulti ()
                       TranslateVector ()

*************************************************************************/

#include "defs.h"

vector VectorStack[VectStackSize];
int TOP = 0;                              /* Top of vector stack */

/*************************************************************************
  PushAll - Pushes the vector, "v" on the vector stack AND saves the
current
            graphic transformations with a "pushmatrix".

  Arguments:
      v --  (vector) vector to be saved.

*************************************************************************/
PushAll (v)
vector v;
{
  pushmatrix ();
  PushVector (v);
}

/*************************************************************************
  PopAll - Pops a vector of the top of the vector stack and places it in
"v"
            AND retrieves a previously stored graphic transformation via a
            "popmatrix".
```

```
  Arguments:
      v --  (vector *) pointer to vector to be returned.

*****************************************************************/
PopAll (v)
vector *v;
{
  popmatrix ();
  PopVector (v);
}

/*****************************************************************
  PushVector - Pushes a vector on the vector stack.  An error message is
               printed to standard error if the stack is full.  (The stack

               is the same size as the transformation stack used by
               "pushmatrix" and "popmatrix".

  Arguments:
      v --  (vector) vector to save.

*****************************************************************/
PushVector (v)
vector v;
{
  register temp;

  if (TOP == VectStackSize - 1)
    fprintf (stderr, "\nVECTORS: Vector stack overflow\n");
  else {
    TOP++;
    VectorStack[TOP] = v;
  }
}

/*****************************************************************
  PopVector - Pops a vector off of the vector stack.  An error message is
              printed to standard error if the stack is empty.

  Arguments:
      v --  (vector *) pointer to the vector returned from stack.

*****************************************************************/
PopVector (v)
vector *v;
{
  if (TOP > 0) {
    *v = VectorStack [TOP];
    TOP--;
  }
  else
```

```
    fprintf (stderr, "\nVECTORS: Attempted to pop empty Vector Stack\n");
}

/*************************************************************************
  RotateMulti - Rotates a vector around all three vertices.

  Arguments:
      v --  (vector *) pointer to vector to modify.
      rotx, roty, rotz --  (short int) amount to rotate (in tenths of a
                           degree)  around  the  x,  y,  and  z  axes,
respectively.

*************************************************************************/
RotateMulti (v, rotx, roty, rotz)
vector *v;
short rotx, roty, rotz;
{
  RotateVector (v, rotx, 'x');
  RotateVector (v, roty, 'y');
  RotateVector (v, rotz, 'z');
}

/*************************************************************************
  RotateMultiEnv - Rotates the environment around all axes and rotates a
                   vector around each axis IN THE OPPOSITE DIRECTION.
                   This routine should be used to rotate something on the
                   screen instead of the built in "rotate" function when
                   there is a viewing vector or light-source vector which
                   must not change orientation.

  Arguments:
      v --  (vector *) pointer to vector to rotate (backwards).
      rotx, roty, rotz --  (short int) amount to rotate (in tenths of a
                           degree)  around  the  x,  y,  and  z  axes,
respectively.

*************************************************************************/
RotateMultiEnv (v, rotx, roty, rotz)
vector *v;
short rotx, roty, rotz;
{
  RotateAll (v, rotx, 'x');
  RotateAll (v, roty, 'y');
  RotateAll (v, rotz, 'z');
}

/*************************************************************************
  RotateAll - Rotates a vector the environment around a given axis and
              rotates a vector IN THE OPPOSITE DIRECTION.  This routine
              should be used to rotate something on the screen instead of
              the built in "rotate" function when there is a viewing vector
```

```
                or light-source vector which must not change orientation.

  Arguments:
       v --  (vector *) pointer to vector to rotate (backwards).
       rot --  (short int) amount to rotate (in tenths of a degree).
       axis --  (char) axis about which to rotate.

*************************************************************************/
RotateAll (v, rot, axis)
vector *v;
short rot;
char axis;
{
  rotate (rot, axis);
  RotateVector (v, -rot, axis);
}

/*************************************************************************
  RotateVector - rotates a vector about a given axis by a given amount.

  Arguments:
       v --  (vector *) pointer to vector to rotate.
       rot --  (short int) amount to rotate (in tenths of a degree).
       axis --  (char) axis about which to rotate.

*************************************************************************/
RotateVector (v, rot, axis)
vector *v;
short rot;
char axis;
{
  float ni, nj, nk, length;

  switch (axis) {
    case 'x':
      nj = v->j * cos((float)rot*convert) - v->k * sin((float)rot*convert);

      nk = v->j * sin((float)rot*convert) + v->k * cos((float)rot*convert);

      v->j = nj;
      v->k = nk;
      break;
    case 'y':
      ni = v->k * sin((float)rot*convert) + v->i * cos((float)rot*convert);

      nk = v->k * cos((float)rot*convert) - v->i * sin((float)rot*convert);

      v->i = ni;
      v->k = nk;
      break;
    case 'z':
```

```
        ni = v->i * cos((float)rot*convert) - v->j * sin((float)rot*convert);
        nj = v->i * sin((float)rot*convert) + v->j * cos((float)rot*convert);
        v->i = ni;
        v->j = nj;
        break;
      default:
        break;
    }
    length = sqrt(v->i * v->i + v->j * v->j + v->k * v->k);
    v->i = v->i / length;
    v->j = v->j / length;
    v->k = v->k / length;
}

/*****************************************************************
  TranslateMulti - translate a vector along all three axes.

  Arguments:
      v --  (vector *) pointer to vector to modify.
      dx, dy, dz --  (float) distance to translate vector along the
                     x, y, and z axes, respectively.

*****************************************************************/
TranslateMulti (v, dx, dy, dz)
vector *v;
float dx, dy, dz;
{
  TranslateVector (v, dx, 'x');
  TranslateVector (v, dy, 'y');
  TranslateVector (v, dz, 'z');
}

/*****************************************************************
  TranslateVector - translates a vector a given distance along a given
axis.

  Arguments:
      v --  (vector *) pointer to vector to modify.
      xlate --  (float) distance to translate vector.
      axis --  (char) axis along which to translate.

*****************************************************************/
TranslateVector (v, xlate, axis)
vector *v;
float xlate;
char axis;
{
  switch (axis) {
    case 'x':
      v->i += xlate;
```

```
            break;
          case 'y':
            v->j += xlate;
            break;
          case 'z':
            v->k += xlate;
            break;
          default:
            break;
        }
      }
```

Appendix A.8 Kinematic Database

```
/*****************************************************************
  Filename:  kindb.c

  by Timothy A. Thompson

  Purpose:  This package implements what is called the "kinematic
            database".
            It includes routines to Load and Save scenes and objects,
            routines to create new objects, routines to add subobjects
            to objects, and routines to rename and scale objects.  This
            package also has routines to position and move objects by
            creating and changing transforms for the objects.

  Functions Provided:  AddKObj ()
                       AddXform ()
                       LoadObj ()
                       NewKObj ()
                       NewXform ()
                       RenameKObj ()
                       SaveObj ()
                       SetKObj_Rbody ()
                       SetKObj_SubObj ()
                       SetScale ()
                       SetXformRot ()
                       SetXformRotMulti ()
                       SetXformTrans ()
                       SetXformTransMulti ()

*****************************************************************/

#include "defs.h"
#include "dbdefs.h"
#include "kindefs.h"

/*****************************************************************
  SaveObj -  Saves an object in one or more files.  The name of the file
             is the same as the name of the object.

  Arguments:
      obj --  (KOBJ *)  object to save.

  Value Returned:  (An error code may be returned in the future)

*****************************************************************/
SaveObj (obj)
KOBJ *obj;
{
  struct vertexrecord {
    VERTEX *vtx;
    int vertnum;
```

```
  struct vertexrecord *next;
};
typedef struct vertexrecord Vptr;

FILE *fp, *fopen();
char filename[30], tmpstr[71];
KOBJ *tmp;
XFORM *xtmp;
FACE *ply;
VERTEX *valid;
int vertcount, colr, flags;
float i, j, k, x, y, z;
Vptr *head, *tail, *listptr;

switch (obj->type) {
  case 'o':
    strcpy (filename, "objlib/");
    strcat (filename, obj->name);
    fp = fopen (filename, "w");
    fprintf (fp, "O\n");
    tmp = obj->obtype.subobj;
    while (tmp) {
      fprintf (fp, "@\n");
      fprintf (fp, "%s\n", tmp->name);
      fprintf (fp, "%f\n", tmp->scale);
      xtmp = tmp->xform;
      while (xtmp) {
        fprintf (fp, "%c\n", xtmp->type);
        fprintf (fp, "%c\n", xtmp->axis);
        switch (xtmp->type) {
          case 't':
            switch (xtmp->axis) {
              case 'x':
              case 'y':
              case 'z':
                fprintf (fp, "%f\n", xtmp->amt.dist);
                break;
              case 'a':
                fprintf (fp, "%f %f %f\n", xtmp->amt.trans.x,
                                           xtmp->amt.trans.y,
                                           xtmp->amt.trans.z);
                break;
              default:
                break;
            }
            break;
          case 'r':
            switch (xtmp->axis) {
              case 'x':
              case 'y':
              case 'z':
                fprintf (fp, "%d\n", xtmp->amt.angle);
```

```
                break;
              case 'a':
                fprintf (fp, "%d %d %d\n", xtmp->amt.rot.x,
                                           xtmp->amt.rot.y,
                                           xtmp->amt.rot.z);
                break;
              default:
                break;
            }
            break;
          default:
            break;
        }
        xtmp = xtmp->next;
      }
      tmp = tmp->nextkobj;
    }
    fclose (fp);
    tmp = obj->obtype.subobj;
    while (tmp) {
      SaveObj (tmp);
      tmp = tmp->nextkobj;
    }
    break;

  case 'r':
    vertcount = 0;
    ply = FirstFace (obj->obtype.rbody);
    if (ply) {
      valid = GetVert (&x, &y, &z);
      if (valid) {
        head = tail = (Vptr *) malloc (sizeof (Vptr));
        if (!head)
          fprintf (stderr, "kindb.c: malloc failed in SaveObj (1)\n");
        else
        {
          tail->vtx = valid;
          tail->vertnum = vertcount++;
          tail->next = NULL;
        }
      }
    }
    while (ply) {
      valid = GetVert (&x, &y, &z);
      while (valid) {
        for (listptr = head; listptr && listptr->vtx != valid;)
          listptr = listptr->next;
        if (!listptr) {
          if ((listptr = (Vptr *)malloc(sizeof(Vptr)))==NULL)
            fprintf (stderr, "kindb.c: malloc failed in SaveObj (2)\n");

          else
```


C 2

```
            {
              listptr->vtx = valid;
              listptr->vertnum = vertcount++;
              listptr->next = NULL;
              tail->next = listptr;
              tail = listptr;
            }
          }
          valid = GetVert (&x, &y, &z);
        }
        ply = NextFace (ply);
      }
      strcpy (filename, "/tmp/");
      strcat (filename, obj->name);
      fp = fopen (filename, "w");
      fprintf (fp, "R\n");
      fprintf (fp, "%d\n", vertcount);
      for (listptr = head; listptr;) {
        fprintf (fp, "%f %f %f\n", listptr->vtx->x, listptr->vtx->y,
listptr->vtx->z);
        listptr = listptr->next;
      }
      fclose (fp);
      strcpy (filename, "/tmp/.");
      strcat (filename, obj->name);
      fp = fopen (filename, "w");
      ply = FirstFace (obj->obtype.rbody);
      while (ply) {
        GetAttribute (ply, &i, &j, &k, &colr, &flags);
        fprintf (fp, "p\n");
        fprintf (fp, "%d\n", colr);
        fprintf (fp, "%f %f %f\n", i, j, k);
        fprintf (fp, "%d\n", flags);
        valid = GetVert (&x, &y, &z);
        while (valid) {
          for (listptr = head; listptr && listptr->vtx != valid;)
            listptr = listptr->next;
          if (listptr)
            fprintf (fp, "v\n%d\n", listptr->vertnum);
          else
            fprintf (stderr, "ERROR: Invalid VERTEX pointer in SaveObj\n");
          valid = GetVert (&x, &y, &z);
        }
        ply = NextFace (ply);
      }
      fclose (fp);
      sprintf (tmpstr, "/bin/cat /tmp/%s /tmp/.%s >objlib/%s",
                                          obj->name, obj->name,
obj->name);
      system (tmpstr);
```

```
        sprintf (tmpstr, "/bin/rm /tmp/%s /tmp/.%s", obj->name, obj->name);
        system (tmpstr);
        for (listptr = head; listptr;) {
          head = head->next;
          free (listptr);
          listptr = head;
        }
        break;
      case 'u':
        fprintf  (stderr,  "ERROR:  Undefined  object  record  found  in
database!\n");
        break;
      default:
        break;
    }
}

/*************************************************************************
  LoadObj -  Load an object from one or more files.

  Arguments:
      obj --  (KOBJ *) pointer to object in which information from file
              will be placed.  (The object should be 'undefined' before
              calling LoadObj.
      obname --  (char *) name of object to load.

  Value Returned:  (An error code may be returned in the future.)

*************************************************************************/
LoadObj (obj, obname)
KOBJ *obj;
char *obname;
{
  FILE *fp, *fopen();
  char filename[30], newname[20], obtype, token, axis;
  float scale, dist, xf, yf, zf, ni, nj, nk;
  int i, angle, numvert, index, colr, xi, yi, zi, flag;
  Boolean firstpoly;
  KOBJ *subobj;
  OBJECT *rbody;
  VERTEX **vtx;
  CORNER *corn;
  XFORM *xfm;
  ATTRIBUTE *attr;

  strcpy (filename, "objlib/");
  strcat (filename, obname);
  fp = fopen (filename, "r");
  fscanf (fp, "%c\n", &obtype);
  switch (obtype) {
    case 'O':
      while (feof (fp) == 0) {
```

```
    fscanf (fp, "%c\n", &token);
    switch (token) {
      case '@':
        fscanf (fp, "%s", newname);
        subobj = NewKObj (newname);
        fscanf (fp, "%f", &scale);
        obj->scale = scale;
        AddKObj (obj, subobj);
        break;
      case 't':
        xfm = NewXform ();
        fscanf (fp, "%c\n", &axis);
        switch (axis) {
          case 'x':
          case 'y':
          case 'z':
            fscanf (fp, "%f", &dist);
            SetXformTrans (xfm, axis, dist);
            break;
          case 'a':
            fscanf (fp, "%f %f %f", &xf, &yf, &zf);
            SetXformTransMulti (xfm, xf, yf, zf);
            break;
          default:
            break;
        }
        AddXform (subobj, xfm);
        break;
      case 'r':
        xfm = NewXform ();
        fscanf (fp, "%c", &axis);
        switch (axis) {
          case 'x':
          case 'y':
          case 'z':
            fscanf (fp, "%d", &angle);
            SetXformRot (xfm, axis, angle);
            break;
          case 'a':
            fscanf (fp, "%d %d %d", &xi, &yi, &zi);
            SetXformRotMulti (xfm, xi, yi, zi);
            break;
          default:
            break;
        }
        AddXform (subobj, xfm);
        break;
      default:
        break;
    }
  }
  fclose (fp);
```

```
        subobj = obj->obtype.subobj;
        while (subobj) {
          LoadObj (subobj, subobj->name);
          subobj = subobj->nextkobj;
        }
        break;
      case 'R':
        fscanf (fp, "%d", &numvert);
        if ((vtx = (VERTEX **) malloc (numvert*sizeof(VERTEX *))) == NULL) {
          fprintf (stderr, "malloc failed in LoadObj\n");
        }
        for (i=0; i<numvert; i++) {
          vtx[i] = NewVertex ();
          fscanf (fp, "%f %f %f", &xf, &yf, &zf);
          SetVertex (vtx[i], xf, yf, zf);
        }
        firstpoly = TRUE;
        rbody = NewRb(UniqueRbNum());
        SetKObj_Rbody (obj, rbody);
        while (feof (fp) == 0) {
          fscanf (fp, "%c\n", &token);
          switch (token) {
            case 'p':
              if (!firstpoly) {
                AddPolygon (rbody, attr);
              }
              fscanf (fp, "%d", &colr);
              fscanf (fp, "%f %f %f", &xf, &yf, &zf);
              fscanf (fp, "%d", &flag);
              attr = NewAttribute ();
              SetAttribute (attr, xf, yf, zf, colr, flag);
              firstpoly = FALSE;
              break;
            case 'v':
              fscanf (fp, "%d", &index);
              corn = NewCorn ();
              SetCorner (corn, vtx[index]);
              AddCorner (rbody, corn);
              break;
            default:
              break;
          }
        }
        if (!firstpoly) {
          AddPolygon (rbody, attr);
        }
        fclose (fp);
        free (vtx);
        break;
      default:
        fclose (fp);
```

```
        break;
    }
}

/*****************************************************************
  AddKObj -  Adds a sub object to an object.

  Arguments:
      parentkobj --  (KOBJ *) object to which child object is being added.

      childKobj --  (KOBJ *) child object which is being added to parent
                    object.

*****************************************************************/
AddKObj (parentkobj, childkobj)
KOBJ *parentkobj, *childkobj;
{
  KOBJ *tmp1, *tmp2;

  if (parentkobj && childkobj) {
    if (parentkobj->obtype.subobj == NULL) {
      parentkobj->type = 'o';
      parentkobj->obtype.subobj = childkobj;
/*      childkobj->parent = parentkobj; */
    }
    else
    {
      if (parentkobj->type == 'o') {
        tmp2 = NULL;
        for (tmp1 = parentkobj->obtype.subobj; tmp1;) {
          tmp2 = tmp1;
          tmp1 = tmp1->nextkobj;
        }
        tmp2->nextkobj = childkobj;
/*        childkobj->parent = parentkobj;  */
      }
    }
  }
}

/*****************************************************************
  NewKObj -  Creates a new and undefined object.

  Arguments:
      name --  (char *)

  Value Returned:

*****************************************************************/
KOBJ *NewKObj (name)
char *name;
{
```

```
    KOBJ *tmp;

    if ((tmp = (KOBJ *) malloc (sizeof (KOBJ))) == NULL)
      fprintf (stderr, "kindb.c:  malloc failed in NewKObj\n");
    else
    {
      strcpy (tmp->name, name);
/*    tmp->modified = FALSE;  */
      tmp->type = 'u';
      tmp->scale = 1.0;
      tmp->xform = NULL;
      tmp->nextkobj = NULL;
/*    tmp->parent = NULL; */
    }
    return tmp;
}

/*****************************************************************************
  SetKObj_SubObj -  Sets a object such that it is made up of the given
                    subobject.  Use of this routine will destroy the
                    previous definition of the object.

  (This routine may be deleted as it was written before "AddKObj" was
   written.  AddKObj has the effect that was needed while this routine
   did not.)

  Arguments:
      mainobj --  (KOBJ *) object which will contain subobject.
      subobj --  (KOBJ *) object which will be the only subobject of
                 the indicated main object.

*****************************************************************************/
SetKObj_SubObj (mainobj, subobj)
KOBJ *mainobj, *subobj;
{
  if (mainobj && subobj) {
    mainobj->type = 'o';
    mainobj->obtype.subobj = subobj;
/*    subobj->parent = mainobj;  */
  }
}

/*****************************************************************************
  SetKObj_Rbody -  Sets a object such that is is made up of the given
                   rigid body.  Use of this routine will destroy the
                   previous definition of the object.

                   If the definition of an object is viewed as a tree,
                   then this routine is used to create a leaf node as a
                   rigid body should be the termination point of any
                   object's sub-components.
```

```
    Arguments:
        mainobj --  (KOBJ *) object which will be made up of a single rigid
                    body.
        rbody --  (OBJECT *) rigid body which will be the sole component of
                  the main object.

*****************************************************************************/
SetKObj_Rbody (mainobj, rbody)
KOBJ *mainobj;
OBJECT *rbody;
{
  if (mainobj) {
    mainobj->type = 'r';
    mainobj->obtype.rbody = rbody;
  }
}

/*****************************************************************************
  SetScale -  Sets the scale factor of an object.  The coordinates of all
              vertices which make up a scaled object or any of its sub-
              objects should be multiplied by this scaling factor.

    Arguments:
        obj --  (KOBJ *) object to scale
        sval --  (float) scaling factor (1.0 makes it the same size).

*****************************************************************************/
SetScale (obj, sval)
KOBJ *obj;
float sval;
{
  obj->scale = sval;
}

/*****************************************************************************
  RenameKObj -  Renames an existing object.

    Arguments:
        obj --  (KOBJ *) object to rename.
        name --  (char *) new name of the object.

*****************************************************************************/
RenameKObj (obj, name)
KOBJ *obj;
char *name;
{
  strcpy (obj->name, name);
}

/*****************************************************************************
  NewXform -  Creates a new transformation to be associated with an object.
```

```
  Value Returned:  (XFORM *) pointer to the new transform.
**************************************************************************/
XFORM *NewXform ()
{
  XFORM *tmp;

  if ((tmp = (XFORM *) malloc (sizeof (XFORM))) == NULL)
    fprintf (stderr, "kindb.c:  malloc failed in NewXform\n");
  else
    tmp->next = NULL;
  return tmp;
}

/*************************************************************************
  SetXformRot -  Sets a transform to be a rotation about a single axis.

  Arguments:
      xfrm --  (XFORM *) pointer to transform to set.
      axis --  (char) axis about which to rotate.
      angle --  (short) amount to rotate (in tenths of a degree).

*************************************************************************/
SetXformRot (xfrm, axis, angle)
XFORM *xfrm;
char axis;
short angle;
{
  xfrm->type = 'r';
  xfrm->axis = axis;
  xfrm->amt.angle = angle;
}

/*************************************************************************
  SetXformRotMulti -  Sets a transform to be a rotation about all axes.

  Arguments:
      xfrm --  (XFORM *) pointer to transform to set.
      x, y, z --  (short) amounts to rotate about each axis (in tenths of a

                  degree).

*************************************************************************/
SetXformRotMulti (xfrm, x, y, z)
XFORM *xfrm;
short x, y, z;
{
  xfrm->type = 'r';
  xfrm->axis = 'a';
  xfrm->amt.rot.x = x;
  xfrm->amt.rot.y = y;
```

```
    xfrm->amt.rot.z = z;
}

/*****************************************************************
  SetXformTrans -  Sets a transform to be a translation about a single
axis.

  Arguments:
      xfrm --  (XFORM *) pointer to transform to set.
      axis --  (char) axis along which to translate.
      dist --  (float) distance to translate.

*****************************************************************/
SetXformTrans (xfrm, axis, dist)
XFORM *xfrm;
char axis;
float dist;
{
  xfrm->type = 't';
  xfrm->axis = axis;
  xfrm->amt.dist = dist;
}

/*****************************************************************
  SetXformTransMulti -  Sets a transform to be a translation about all
axes.

  Arguments:
      xfrm --  (XFORM *) pointer to transform to set.
      x, y, z -- (float) distance to translate along each axis.

*****************************************************************/
SetXformTransMulti (xfrm, x, y, z)
XFORM *xfrm;
float x, y, z;
{
  xfrm->type = 't';
  xfrm->axis = 'a';
  xfrm->amt.trans.x = x;
  xfrm->amt.trans.y = y;
  xfrm->amt.trans.z = z;
}

/*****************************************************************
  AddXform -  Adds a transformation to an object.  All vertices in the
              object or any sub-object will be affected by the
              transformation.

  Arguments:
      obj --  (KOBJ *) object to which transformation is to be added.
      xfrm --  (XFORM *) pointer to transform to add to the object.
```

```
*****************************************************************************/
AddXform (obj, xfrm)
KOBJ *obj;
XFORM *xfrm;
{
  XFORM *ptr;

  if (obj->xform == NULL)
    obj->xform = xfrm;
  else
  {
    for (ptr=obj->xform; ptr->next; ptr=ptr->next);
    ptr->next = xfrm;
  }
}
```

```
/*************************************************************************
  Filename:  lp.c

  Written by:   Allan Rideout
  Modified by:  Timothy A. Thompson

  Purpose:  This module implements the database which is responsible for
            storing rigid bodies.  For more information on how these rigid
            bodies are stored, see the file "obj.h".

  Functions Provided:

*************************************************************************/
/* lp.c 01.11.89*/

#include  "defs.h"
#include  "dbdefs.h"

FACE *fce;
BEDGE *bedg;
EDGE  *edg;
VERTEX *vtx;
IEDGE *iedg;
CORNER *corn;
OBJECT *obj;
ATTRIBUTE *attr;

/*******************************************************
  loop_poly -  Traverses the winged edge database and creates a list of
               corners (pointers to vertices) contained in the polygon
               pointed to by the global variable "fce".  The list is
               accessed by the "rcorn" field of the structure pointed to by

               the global variable "obj".

*************************************************************************/
loop_poly()
{
 /*loop polygon specified by fce
  and place corners in a clockwise sequence*/

  EDGE *edg1;
  bedg = fce->bedg;
  edg = bedg->edg;
  bedg = bedg->nextbedg;

  /*first vtx is that found an both current and next edg.*/

  edg1 = bedg->edg;
  if((edg->vtx1 == edg1->vtx1)||
```

```
      (edg->vtx1 == edg1->vtx2)) vtx = edg->vtx1;
    else vtx = edg->vtx2;

    if ((corn = (CORNER *) malloc (scorn)) == NULL) {
      fprintf (stderr, "lp: malloc failed in loop_poly\n");
    }
    corn->nextcorn = NULL;
    corn->vtx = vtx;

    add_corner();

    /*find the sequence: next vtx is the different vtx found on
    the next edg*/

  /*  bedg = bedg->nextbedg; */
    while(bedg)
      {
       edg = bedg->edg;
       if(vtx == edg->vtx1) vtx = edg->vtx2;
       else vtx = edg->vtx1;

       if ((corn = (CORNER *) malloc (scorn)) == NULL) {
         fprintf (stderr, "lp: malloc failed in loop_poly\n");
       }
       corn->nextcorn = NULL;
       corn->vtx = vtx;

       add_corner();
       bedg = bedg->nextbedg;
      }
  }


  /*********************************************************************
    add_corner -   Adds the corner pointed to by "corn" to the list of
  vertices
                 which will define the new polygon being built for the
  object
                 pointed to by "obj".

  *********************************************************************/
  add_corner()
  {
   /*current corner (corn) becomes the new root of the current
   polygon*/

   CORNER *q;

   q=obj->rcorn;
   obj->rcorn=corn;
   corn->nextcorn=q;
  }
```

```
/*************************************************************************
  unjoin -  (not currently used)

*************************************************************************/
unjoin()
{
 /*last corner removed from current polygon to become a
 free point*/

 CORNER *q;

 q=obj->rcorn;
 obj->rcorn=q->nextcorn;
 free(q);
}

/*************************************************************************
  abandon_polygon -  Destroys the temporary linked list pointed to by the
                     "rcorn"  field  of  the  structure  pointed  to  by  the
global
                     variable "obj".

*************************************************************************/
abandon_polygon()
{
 /*all points in current polygon are returned to previous
 status*/

 CORNER *q;
 for(corn=obj->rcorn;corn;)
 {
     vtx=corn->vtx;
        q=corn;
    corn=corn->nextcorn;
     free(q);
        obj->rcorn = NULL;
 }
}

/*************************************************************************
  add_polygon -  Build a database representation of a polygon out of the
                 corners pointed to by the "rcorn" field of the structure
                 pointed to by the global variable "obj".

*************************************************************************/
add_polygon()
{
 /*current polygon (point list with root, obj->rcorn) is
 inserted into object data structure*/

 FACE *q;
```

```
    int ncorn=0;
    for(corn=obj->rcorn;corn;)
    {
        ncorn++;
        corn=corn->nextcorn;
    }
    if(ncorn<3)
    {
        printf("\n%d  corners,\n\n\nno polygon.",ncorn);
        return(0);
    }
    q=obj->fce;
    if ((fce=(FACE*)malloc(sfce)) == NULL) {
      fprintf (stderr, "lp: malloc failed in add_polygon\n");
    }
    fce->nextfce = NULL;
    fce->bedg = NULL;
    fce->attr = attr;
    add_bedge();
    fce->bedg=bedg;
    fce->nextfce=q;
    obj->fce=fce;

    abandon_polygon();
}

/*****************************************************************************
  add_bedge -  Adds a bounding edge pointed to by "bedg" to the database.
*****************************************************************************/
add_bedge()
{
 BEDGE *q;
 bedg=NULL;
 for(corn=obj->rcorn;corn;)
 {
  add_edge();
  q=bedg;
  if ((bedg=(BEDGE*)malloc(sbedg)) == NULL) {
    fprintf (stderr, "lp: malloc failed in add_bedge\n");
  }
  bedg->edg=edg;
  bedg->nextbedg=q;
  corn=corn->nextcorn;
  }
}


/*****************************************************************************
  add_edge -  Adds the edge pointed to by "edg" to the database.
*****************************************************************************/
```

```
add_edge()
{
 VERTEX *vtx1;
 CORNER *corn1;
 EDGE *edg1;
 IEDGE *iedg1;

 vtx=corn->vtx;
 corn1=corn->nextcorn;
 if(corn1==NULL) corn1=obj->rcorn;
 vtx1=corn1->vtx;

          /*search for vtx,vtx1 edge*/
 for(iedg=vtx->iedg; iedg;)
 {
  edg=iedg->edg;
  for(iedg1=vtx1->iedg;iedg1;)
  {
     edg1=iedg1->edg;
     if((edg==edg1)&&(edg->fce2==NULL))
     {
      add_iedge();
      vtx=vtx1;
      add_iedge();
      edg->fce2=fce;
         return(1);
     }
     iedg1=iedg1->nextiedg;
  }
  iedg=iedg->nextiedg;
 }
    /*must add a new edge*/

 if ((edg=(EDGE*)malloc(sedg)) == NULL) {
   fprintf (stderr, "lp: malloc failed in add_edge\n");
 }
 edg->fce1=fce;
 edg->fce2=NULL;
 edg->vtx1=vtx;
 edg->vtx2=vtx1;
 add_iedge();
 vtx=vtx1;
 add_iedge();
}

/*************************************************************************
  add_iedge -  Adds to the incident edge list when a new edge is added.
               (There is currenty a bug which causes each incident edge to

               appear on the lists twice.)
```

```
*****************************************************************************/
add_iedge()
{
 IEDGE *q,*p;

 if ((iedg=(IEDGE*)malloc(siedg)) == NULL) {
   fprintf (stderr, "lp: malloc failed in add_iedge\n");
 }
 iedg->edg=edg;
 iedg->nextiedg = NULL;

 if(vtx->iedg==NULL)
 {
  vtx->iedg=iedg;
  return(1);
 }
 for(q=vtx->iedg;q;)
 {
  p=q;
  q=q->nextiedg;
 }
 p->nextiedg=iedg;
}


/*****************************************************************************
  remove_face -  (Currently unused and untested.)

*****************************************************************************/
remove_face()
{
 /*fce is removed from object data structure*/

 FACE *q;
 remove_bedge();
 q=fce;
 fce=q->nextfce;
 free(q);
}

/*****************************************************************************
  remove_bedge -  (Currently unused and untested.)

*****************************************************************************/
remove_bedge()
{
 BEDGE *q;

 for(bedg=fce->bedg;bedg;)
 {
  edg=bedg->edg;
       if(edg->fce1==fce) edg->fce1=NULL;
```

```
    else if(edg->fce2==fce) edg->fce2=NULL;

         if(edg->fce1==edg->fce2)    /* edg is now empty*/
        {
         vtx=edg->vtx1; remove_iedge();
         vtx=edg->vtx2; remove_iedge();
        }
    q=bedg;
    bedg=q->nextbedg;
    free(q);
   }
  }

  /*************************************************************************
    remove_iedge -  (Currently unused and untested.)

  *************************************************************************/
  remove_iedge()
  {
   IEDGE *p,*q;

   q=vtx->iedg;
   if(q->edg==edg)
   {
       vtx->iedg=q->nextiedg;
          free(q);
       return(1);
   }
   while(q->edg!=edg)
   {
       p=q;
       q=q->nextiedg;
   }
   p->nextiedg=q->nextiedg;
   free(q);
  }
```