ARMY RESEARCH LABORATORY

BRL-CAD Tutorial Series: Volume II – Introduction to MGED

by Lee A. Butler, Eric W. Edwards, and Betty J. Schueler

ARL-SR-102 April 2001

Approved for public release; distribution is unlimited

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5068

ARL-SR-102 **April 2001**

BRL-CAD Tutorial Series: Volume II – Introduction to MGED

Lee A. Butler
Survivability/Lethality Analysis Directorate, ARL

Eric W. Edwards
SURVICE Engineering Company

Betty J. Schueler
Quantum Research International

Approved for public release; distribution is unlimited.

Abstract

Since 1979, the U.S. Army Research Laboratory has been developing and distributing the BRL-CAD constructive solid geometry (CSG) modeling package for a wide range of military and industrial applications. The package includes a large collection of tools and utilities including an interactive geometry editor, ray-tracing and generic framebuffer libraries, a network-distributed image-processing and signal-processing capability, and an embedded scripting language.

As part of this effort, a multivolume tutorial series is being developed to assist users in the many features of the BRL-CAD package. The "Introduction to MGED," which is the second volume in the series, is intended to provide new users with a basic understanding of the Multi-Device Geometry Editor (MGED), which is the heart of BRL-CAD. Other volumes focus on installation procedures, advanced features, and programming.

Preface

Since 1979, the U.S. Army Research Laboratory (formerly the Ballistic Research Laboratory) has been developing the BRL-CAD constructive solid geometry (CSG) modeling package for a wide range of military and industrial applications. The strength of the package lies in its ability to build realistic models of complex objects from a relatively small set of "primitive shapes" by employing the basic Boolean operations of union, subtraction, and intersection and by assigning real-world material attributes.

The package comprises a large collection of tools, utilities, and libraries including an interactive geometry editor, ray-tracing and generic framebuffer libraries, a network-distributed image-processing and signal-processing capability, and an embedded scripting language.

Although BRL-CAD has continued to mature in performance and utility, developers have strived to keep the package approachable and easy to use, as evidenced by the package's dual command formats, its newly renovated graphical user interface (GUI), and its customization potential through user scripting.

In addition, a multivolume tutorial series is being developed to assist users in a variety of BRL-CAD areas and applications. The "Introduction to MGED," which is the second volume in the series, is intended to provide new users with a basic understanding of the Multi-Device Graphics Editor (MGED), which is the heart of the BRL-CAD package. Other volumes focus on installation procedures, advanced features, and programming.

Intentionally Left Blank

Acknowledgments

The authors would like to thank the members of the Advanced Computer Systems Team, who reviewed this document in a timely manner and made many helpful suggestions to improve its accuracy and presentation. Team members include Paul Tanenbaum, John Anderson, TraNese Christy, Bob Parker, Ron Bowers, Sean Morrison, and Chris Johnson.

In addition, the authors would like to especially acknowledge team member Mike Muuss, who passed away while this volume was in preparation. Mike was the original architect of the BRL-CAD package and guided its development for 20 years until his death on 20 November 2000. He embodied a unique blend of unparalleled intellect, unquenchable curiosity, and unending energy to advance the capabilities of everything and everyone he touched. A natural-born troubleshooter, Mike approached every job, big or small, with a passion for excellence and a child-like enthusiasm, which helped drive BRL-CAD far beyond expectations.

Although he never got a chance to review this document, this work would certainly not have been possible without his vision and attention to detail. Therefore, the BRL-CAD Tutorial Series is dedicated to his memory. His sharp mind, his warm spirit, and his loyal friendship will be greatly missed.

Intentionally Left Blank

Table of Contents

		Page
	Preface	iii
	Acknowledgments	v
Lesson 1:	Creating Primitive Shapes	1
Lesson 2:	Learning the Viewing Options	11
Lesson 3:	Using the Insert Command to Size and Place Shapes	23
Lesson 4:	Assigning Material Properties and Raytracing	31
Lesson 5:	Learning About Boolean Expressions	37
Lesson 6:	Creating a Goblet	47
Lesson 7:	Assigning Material Properties to Your Goblet	55
Lesson 8:	Assigning More Material Properties to Your Goblet	61
Lesson 9:	Creating a Globe in a Display Box	69
Lesson 10:	Creating a Mug	77
Lesson 11:	Refining the Mug	83
Lesson 12:	Creating the Mug Through the GUI	87
Lesson 13:	Placing Shapes in 3-D Space	93
Lesson 14:	Gaining More Practice Placing Shapes in Space	103
Lesson 15:	Creating a Toy Truck	113
Lesson 16:	Learning Modeling Techniques and Structures	129
Appendix A:	MGED Commands	147
Appendix B:	Emacs and Vi Commands	247
Appendix C:	Primitive Shapes	251
	Index	261
	Distribution List	267
	Report Documentation Page	269

Intentionally Left Blank

Supplementary Tutorial Boxes

Note that throughout this document, tutorial boxes have been used to supplement the information presented in the text. Each box is labeled with one of the following icons for ease of recognition:

Point of Caution

Further Information

Key Point

Alternative Idea/Method

Intentionally Left Blank

Lesson 1: Creating Primitive Shapes

In this lesson, you will:

- Launch the MGED program.
- Enter commands at the MGED prompt in the Command Window.
- Use the MGED Graphical User Interface (GUI).
- Open or create a new database when launching MGED.
- Use the GUI to open or create a new database.
- Title a database.
- Select a unit of length for your design.
- Select a primitive shape.
- Create a primitive shape using the **make** command.
- Use the **Z** command to clear the Graphics Window.
- Draw a previously created shape using the **draw** command.
- Use the **erase** command to delete an item in the Graphics Window display.
- Create a sphere using the GUI menu.
- Use the **l** command to list a shape's attributes or parameters.
- Use the **ls** command to list the contents of the database.
- Eliminate a shape or object from the database using the **kill** command.
- Edit a command.
- Use the **q** or **exit** commands to quit the program.

1. Launching the MGED Program

To launch the MGED program, type **mged** at the Terminal (tty) prompt and then press the **ENTER** key. This brings up two main windows: the MGED Command Window and the MGED Graphics Window (sometimes called the Geometry Window). Both windows will initially be blank, awaiting input from you. To leave the program at any time, at the Command Line type either the letter **q** or the word **quit** and then press the **ENTER** key.

2. Entering Commands in the Command Window

You can type in commands at the **`mged>`** prompt. Many experienced UNIX users prefer this method because it allows them to quickly create a model (which we sometimes refer to as a "design") without having to point and click on a lot of options. The complete listing of editing commands and what they do can be found in Appendix A.

	Check all typed entries before you press the **ENTER** key. If you find you made a mistake, simply press the **BACKSPACE** key until you have erased over the mistake and then re-type the information. Later you will get more experience editing text using **vi** and **emacs** command emulation.

3. Using the GUI

Users who are more familiar with Microsoft Windows may prefer to use the GUI pull-down menus at the top of the Command or Graphics Window (they are the same in either window). The menus are divided into logical groupings to help you navigate through the MGED program.

Before you can create a model, you need to open a new database either through the Terminal Window when starting MGED or through the GUI after starting MGED.

4. Opening or Creating a New Database when Launching MGED

When launching MGED, you can open or create a database at the same time. At the shell prompt (usually a $ or %), in the Terminal Window, type **mged** followed by a new or existing database name with a **.g** extension. For example:

```
mged sphere.g<ENTER>
```

Terminal Window

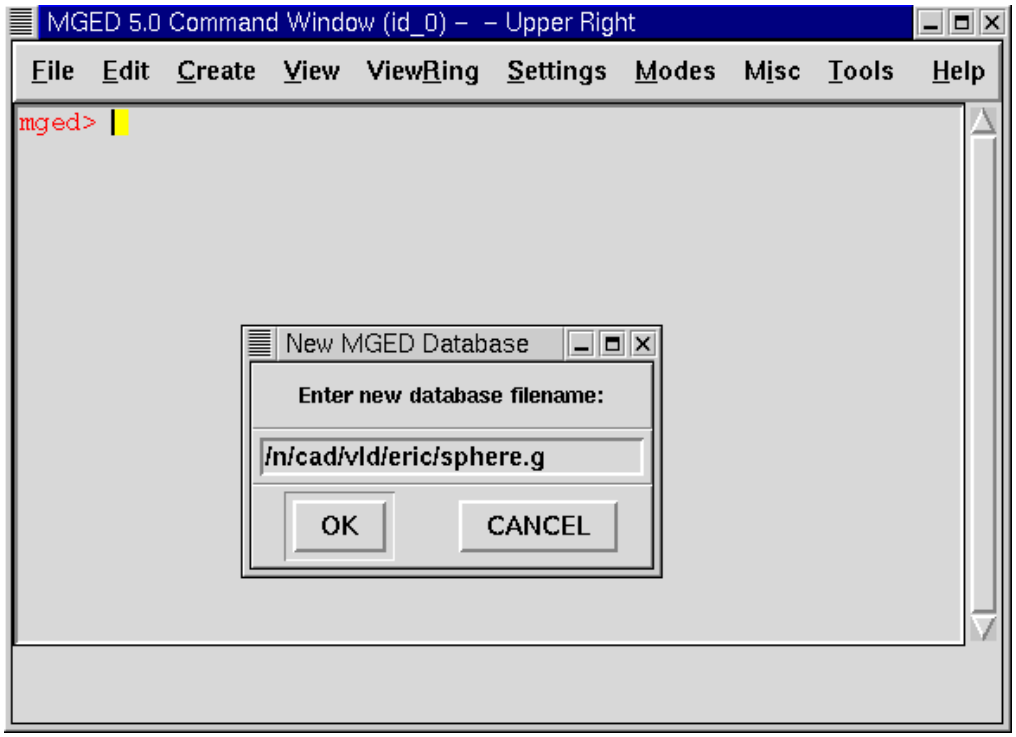
If you are creating a new database, a small dialog box asking if you want to create a new database named **sphere.g** will appear. Click on **Yes**, and a new database will be created.

If **sphere.g** already exists, MGED will open the **sphere.g** database as the program is launched.

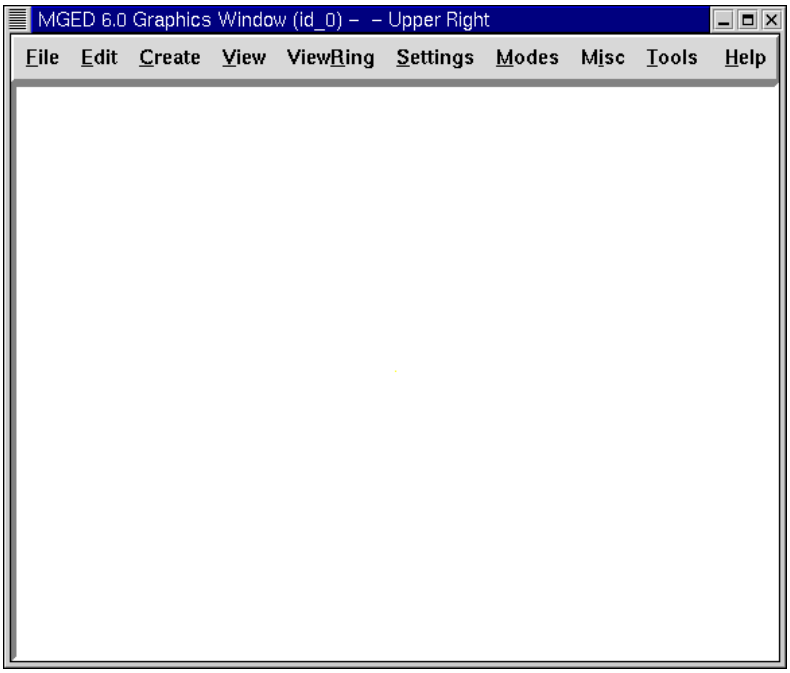
5. Using the GUI to Open or Create a Database

Alternatively, once you have launched MGED, you can open an existing database or create a new database using the GUI menus (at the top of the Command or Graphics Window) by clicking on **File** and then either **Open** or **New**. Both options bring up a small dialog box. The **Open** dialog box will ask you to type in the name of an existing database. The **New** dialog box will ask you to type in the name of a new database. Click on **OK** to accept the database.

For this lesson, create a new database called **sphere.g**. To do this, type **sphere.g** at the end of the path name, as shown in the following illustration. Click on **OK** to accept the database name.



MGED Command Window with Database Dialog Box



MGED Graphics Window

One advantage to using the GUI, if you aren't familiar with UNIX file management, is that this will show you your current path name, so you will know exactly where your database is going to be located. This can be especially helpful if you have a lot of directories or files to manage.

6. Assigning a Title to Your Database

You can title your new database to provide an audit trail for you or others who might use your database. After the prompt, in the Command Window, type **title** followed by a space and a name that reflects the database you are going to make. When you are done, press the **ENTER** key. For example:

```
mged> title MySphere<ENTER>
```

Note that in BRL-CAD versions prior to release 6.0, the title is limited to 72 characters.

7. Selecting a Unit of Length

MGED uses millimeters for all internal mathematical processes; however, you can create your design using some other unit, such as feet. For this lesson, inches is used. To select inches, move your mouse pointer to the **File** menu at the top of the Command Window. Click on **File** and then **Preferences.** A new menu will appear. Select **Units** and then **Inches**. If you are not a "point-and-click" type of person and prefer a Command Line,

then just type **units in** after the MGED prompt in the Command Window, followed by the **ENTER** key. The Command Line looks like:

```
mged> units in<ENTER>
```

8. Selecting a Primitive Shape

MGED provides a variety of *primitive shapes* (sometimes referred to as simply *shapes* or *primitives*) that you can use to build models. Each type of shape has parameters that define its position, orientation, and size. A listing of these shapes and their respective parameters is given in Appendix C.

	Historically, the word *solid* was used for what we now refer to as a *primitive shape*. This older terminology was frequently difficult for new users to understand. If you see the word *solid* used in any BRL-CAD programs or documentation, think *primitive shape*.

9. Creating a Sphere from the Command Line

For this lesson, you are going to create a single sphere. There are two ways you can create a primitive shape. You can create all shapes through the Command Window and most shapes through the GUI.

You can easily create a sphere from the prompt in the Command Window by typing just a few commands. At the MGED prompt, type:

```
make sph1.s sph<ENTER>
```

[Note: Use the digit 1*, not the letter* l*]*

This command tells the MGED program to:

make	sph1.s	sph
Make a primitive shape	Name it sph1.s	Make the shape a sphere

A default sphere will be created, and a wireframe representation of the primitive shape will appear in the Graphics Window. In Lesson 4, you will give your sphere a solid, three-dimensional look.

This command will draw the primitive shape in the Graphics Window.

10. Clearing the Graphics Window

To build another object or work on another primitive shape, you can easily clear the Graphics Window through the Command Window. At the Command Line prompt, type an uppercase **Z** (for zap) followed by **ENTER**.

	Note: Before using the **zap** option, make sure you "activate" (i.e., set the focus on) the Command Window. If you type a **z** and your cursor is still in the Graphics Window, you will send your design spinning. Typing a zero (0) will stop the spin.

11. Drawing a Previously Created Object

To recall the sphere, type the command on the Command Line as follows:

```
draw sph1.s<ENTER>
```

This command tells the MGED program to:

draw	sph1.s
Draw a previously created primitive shape	named sph1.s

12. Erasing an Item from the Graphics Window

You may occasionally want to erase a particular item from the display in the Graphics Window. You can use the **erase** command to remove the item without any file operation being performed; the item remains in the database. To delete the **sph1.s** object from the display, at the Command Window prompt, type:

```
erase sph1.s<ENTER>
```

13. Creating a Sphere Using the GUI

Another way to create a sphere is to use the GUI menu system duplicated at the top of the Command and Graphics windows. Clear your Graphics Window by using the previously described **Z** command. Then, in the Graphics Window, click on **Create** and then **Make Solid**. A drop-down menu will appear with the various primitive shape types available.

Select **sph** for sphere. This will bring up a dialog box. Click in the empty text box and type **sph2.s**. Click on **Apply** or press **ENTER**. A new sphere will be created and drawn in the Graphics Window. When you create a shape through the GUI, the shape will automatically be in *edit mode* so that you can change it as needed, and the shape's parameters—which define its position, orientation, and size—will be in view.

14. Viewing a Shape's Parameters

Sometimes when you are making a model, you might want to view a shape's parameters, such as height, width, or radius, in the Command Window. You can easily list the attributes of a shape by typing the **l** (for "list") command at the Command Window prompt as follows:

l *shape_name***<ENTER>** *[Note: The command is the lowercase letter* l*]*

	Note: If you attempt to type in the Command Window and you see no words appearing there, chances are the focus has not been set on that window (i.e., keyboard input is still directed to another window). Depending on your system's configurations, the focus is set to a window either by moving the cursor into the window or clicking on the window.

An example of the dialog that might occur in the Command Window for the parameters or attributes of the first sphere you created is as follows:

```
mged> l sph1.s
sph1.s:   ellipsoid  (ELL)
     V  (1, 1, 1)
     A  (1, 0, 0)  mag=1
     B  (0, 1, 0)  mag=1
     C  (0, 0, 1)  mag=1
     A  direction cosines=(0, 90, 90)
     A rotation angle=0, fallback angle=0
     B direction cosines=(90, 0, 90)
     B rotation angle=90 fallback angle=0
     C direction cosines=(90, 90, 0)
     C rotation angle=0, fallback angle=90
```

Don't be concerned if you notice in the preceding output that MGED stores your sphere as an ellipsoid. In actuality, the sphere is just a special case of the ellipsoid (see Appendix C). Also, note that it is not important if the numbers in your output do not match what is shown in this example.

Use the **l** command to list both **sph1.s** and **sph2.s** before continuing with this lesson.

15. Listing the Contents of a Database

In addition to viewing a shape's contents, you might also want to list the contents of the database to see what items have been created. To view the database contents, type at the Command Window prompt:

```
ls<ENTER>
```

16. Killing a Shape or Object

Sometimes when creating a model, you may need to eliminate a shape or object from the database. The **kill** command is used to do this. For example, if you wanted to **kill** the **sph1.s** shape, you would type at the Command Window prompt:

```
kill sph1.s<ENTER>
```

Make another sphere through either the Command Window or the GUI and name it **sph3.s**. Once the sphere is made, use the **kill** command to eliminate it from the database by typing at the Command Window prompt:

```
kill sph3.s<ENTER>
```

You can tell the shape has been eliminated by using the **ls** command in the Command Window to list the contents of the database. At the Command Window prompt, type:

```
ls<ENTER>
```

You should see two shapes listed: **sph1.s** and **sph2.s**.

	Note: All changes are immediately applied to the database, so there is no "save" or "save as" command. Likewise, there is presently no "undo" command to bring back what you have deleted, so be sure you really want to permanently delete data before using the **kill** command.

17. Editing Commands in the Command Window

Occasionally, when you enter commands in the Command Window, you will make a mistake in typing. MGED can emulate either the *emacs* or *vi* syntax for Command Line editing. By default, the emacs syntax is used. See Appendix B for a list of keystrokes, effects, and ways to select between the two command sets.

You can also use the arrow keys to edit commands. The left and right arrow keys move the cursor in the current Command Line. Typing **ENTER** at any location on the

Command Line executes the command. Note that both the **BACKSPACE** and **DELETE** keys will delete one character to the left of the cursor.

MGED keeps a history of commands that have been entered. The up and down arrow keys allow you to select a previously executed command for editing and re-execution.

18. Quitting MGED

Remember, to leave the program at any time, type from the Command Line either the letter **q** or the word **quit** and then press the **ENTER** key. You may also quit the program by selecting **Exit** from the **File** menu.

Review

In this lesson, you:

- Started the MGED program.
- Entered commands in the Command Window.
- Used the MGED GUI.
- Created or opened a database using MGED naming conventions.
- Used the GUI to create or open a database.
- Titled a database.
- Selected a unit of measure for a design.
- Selected a primitive shape.
- Created a primitive shape using the **make** command in the Command Window.
- Cleared the screen of a design using the **Z** command.
- Drew a previously created shape using the **draw** command.
- Used the **erase** command to delete a shape from the Graphics Window display.
- Used the GUI to create a primitive shape.
- Used the **l** command to view a shape's parameters.
- Used the **ls** command to list the contents of the database.
- Used the **kill** command to eliminate a shape from the database.
- Edited commands in the Command Window.
- Used the **q** or **Exit** commands to quit the program.

Intentionally Left Blank

Lesson 2: Learning the Viewing Options

In this lesson, you will:

- Create a model radio.
- Locate viewing information in the Command Window.
- Identify elements of the MGED viewing system.
- View your radio from different angles.
- Work with Shift Grips.

Models in BRL-CAD are constructed in a single *xyz* coordinate system, which we sometimes refer to as *model space*. The Graphics Window of MGED displays a portion of this space. The *xyz* coordinate system is used for specifying both the geometry and the view of the geometry that is presented in the Graphics Window.

MGED offers a default view and a variety of optional views. You can switch back and forth between these views during and after model creation. This lesson is designed to help you understand the viewing process and options.

1. Creating a Radio

To gain practice viewing actual geometry, let's build a simple geometric object—a "walkie-talkie" radio. Note that the commands we use to do this are not discussed here because our current concern is on applying the principles of viewing. Later lessons on creating geometry address these commands in detail. Begin by launching MGED and creating a new database named **radio.g**. Remember that one way to do this is to type the following command in a Terminal Window:

```
$ mged radio.g
```

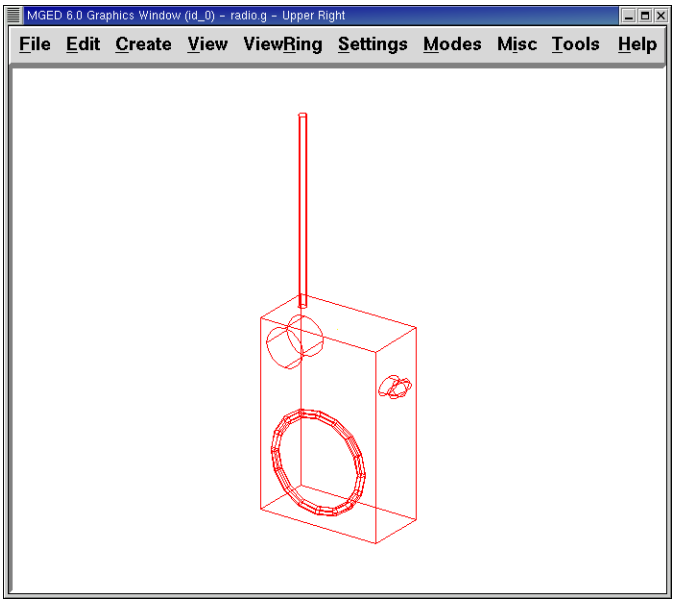
Type the following in the Command Window, carefully checking each line before pressing **ENTER**. If you make a mistake, use **BACKSPACE** or the left/ right arrow keys to make corrections before pressing **ENTER** (see Appendix B for the editing command list).

	Be especially careful to note the difference between the numeral one [`1`] and the letter l [`l`] in `ell1` on the third line.

```
in body.s rpp  0 16   0 32   0 48<ENTER>
in btn.s  rec  8 30 36  0 3  0   4 0 0  0 0 2<ENTER>
in btn2.s ell1 8 33 36  4 0  0   2<ENTER>
in spkr.s tor 16 16 16  1 0  0  12 1<ENTER>
in ant.s  rcc  2  2 46  0 0 48   1<ENTER>
in knob.s rcc  4  4 40  8 0  0   5<ENTER>
```

	Note that in the preceding Command Line expressions, **btn** is an abbreviation for **button**, **ant** is an abbreviation for **antenna**, and **spkr** is an abbreviation for **speaker.** Also note that the numbers could have been separated by single spaces. The extra spaces were inserted simply to improve readability. For some usages (e.g., the **r** and **comb** commands, which are discussed later), the number of spaces has to be exact.

An image similar to the following should now appear in the Graphics Window.



Default View of a Radio

2. Locating Viewing Information in the Command Window

Now take a minute to look at the Command Window. Even if nothing is in the window, enclosed in the bottom border is a string of information about the Graphics Window. An example string might read:

```
cent=(8.000 16.000 24.000) sz=96.000 mm az=35.00 el=25.00
tw=-0.00 ang=(0.00 0.00 0.00)
```

As detailed in the following table, this information contains four groups of viewing data about the Graphics Window.

Viewing Data at the Bottom of the Command Window

Screen Designation	Viewing Information	Location of Variables	Default Units	Default Values
cent=	Center of View	First 3 numbers	Millimeters	0.000 0.000 0.000
sz=	Size of View	4th number	Millimeters	Dependent upon size selected
az= el=	Viewing Angle	5th and 6th numbers	Degrees	35.00 25.00
tw= ang=	Twist and Angle of View	7th–10th numbers	Degrees	0.00 0.00 0.00 0.00

3. Identifying Elements of the MGED Viewing System

Center of View

The first set of information tells you the center of what you are viewing. You can change the center of where you are looking through both the GUI and the Command Window.

To change the center of your view of the radio using the GUI, press the **SHIFT** key and *any* mouse button while dragging the mouse. (This is an example of a Shift Grip, which is described later in this chapter.) You can also change the center of view by placing the mouse pointer where you want the center to be and clicking the *middle* mouse button.

To change the center of view using the Command Window, simply type at the prompt the word **center** followed by three values for *x*, *y*, and *z* (which is the 3-D coordinate system mentioned previously). For example:

```
center 0 15 325.735<ENTER>
```

As you change your view of the geometry, notice that the numbers in the brackets after the *cent*= title will change to reflect the new center of the view.

Size of View

The size of the view is the amount of model space that is shown in the Graphics Window. For example, consider using a camera with a zoom lens to photograph a rose. As shown in the following figures, if you zoom in on the rose, it will *appear* large in relation to your viewing field. If you zoom out, it will *appear* smaller. In actuality, the view size for the rose image on the left might represent only 15 mm across while the view size for

the image on the right might represent 100 mm across. In both cases, however, the actual size of the rose is the same.

Zoom In to View Details
(small size of view)

Zoom Out to View Object
in Relation to Environment
(large size of view)

To change the view size of your radio through the GUI, click the *right* mouse button to zoom in and the *left* mouse button to zoom out. Each time you click the left or right mouse button, the view of the design will increase or decrease in size by a factor of 2 (i.e., two times larger or two times smaller than the previous size).

You can also zoom in or out on your design by going to the **View** menu and selecting **Zoom In** or **Zoom Out**. A drawback to this method is that you can only zoom in or out one time because the drop-down menu closes once you make a selection.

If you get lost at any point while creating a model, you can use the **zap** (**Z**) command to clear the geometry from the Graphics Window and then recall the shape with the **draw** command. When drawing in an empty Graphics Window, MGED automatically sizes the view to fit what you draw into the window.

You can control the view size of your radio more accurately with the Command Window. To set the size to 100 (of whichever units you have selected), type at the prompt:

```
size 100<ENTER>
```

You can also zoom in or out on a design by typing **zoom** on the Command Line. To make your radio appear 50% smaller, you would type:

```
zoom 0.5<ENTER>
```

To make your radio appear twice as large, you would type:

zoom 2<ENTER>

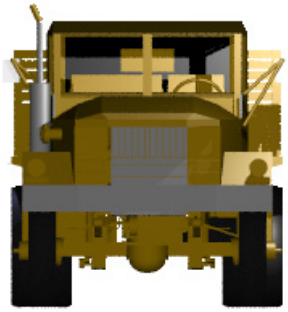
	Remember that changing the view size does NOT affect the size of the object. You will change the size of an object in Lesson 6.

Angle of View

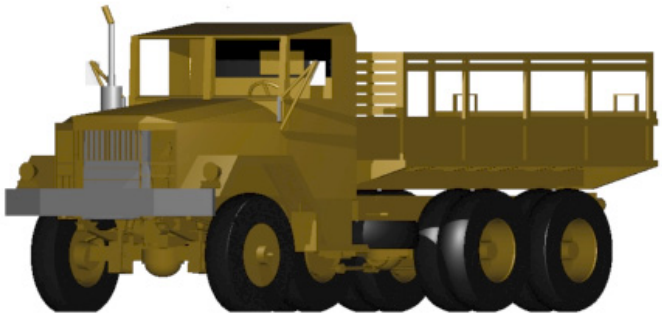
Azimuth, *elevation*, and *twist* (all measured in degrees) determine where *you* are in relation to the object you are viewing. Azimuth determines where you are around the *sides* of it (i.e., to the front, left, right, behind, or somewhere in between), elevation determines where you are *above* or *below* it, and twist determines the *angle* you are rotated about the viewing direction.

To better understand azimuth, imagine walking around a truck with a camera to photograph it. As shown in the following illustrations, you would be at 0˚ azimuth if you stood directly in front of the truck to take its picture. If you circled around slightly to your right, you would be at 35˚ azimuth. If you moved further around until you were looking directly at the driver's side (in U.S. trucks), you would be at 90˚ azimuth. Standing behind it would put you at 180˚ azimuth. If you were facing the passenger's side, you would be at 270˚ azimuth.

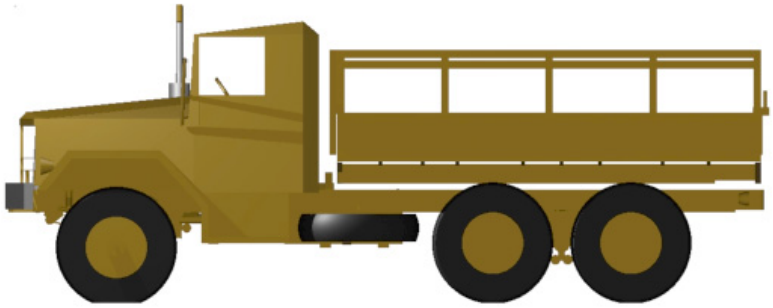
	The terms *azimuth*, *elevation*, and *twist* are similar to the terms *yaw*, *pitch*, and *roll*, respectively, which are common terms in the aerospace industry.



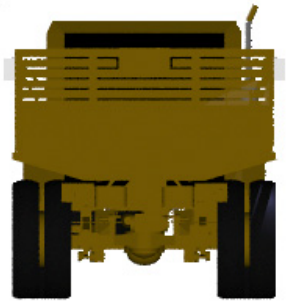
Front (az=0, el=0)



az=35, el=0



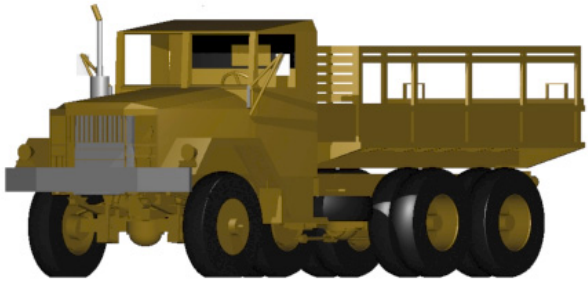
Left (az=90, el=0)



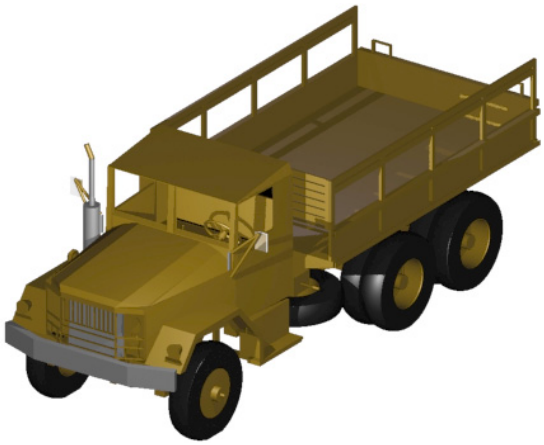
Rear (az=180, el=0)

Right (az=270, el=0)

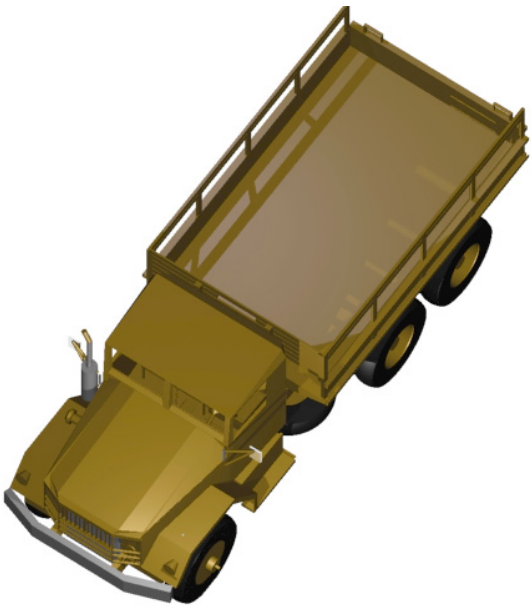
Elevation, on the other hand, involves the viewer's position above or below an object. In the preceding example, you circled around a truck without changing your relative *height*. You had an elevation of 0˚, which means you were level with it. As the following figures illustrate, however, imagine stopping at the 35˚ azimuth position and then climbing a ladder to photograph the truck from 25˚ elevation. Climbing higher, you would be at 60˚ elevation. If you were directly above it with the camera facing down, you would be at 90˚ elevation. If you crawled under the truck and looked directly up at it, you would be at –90˚ elevation.



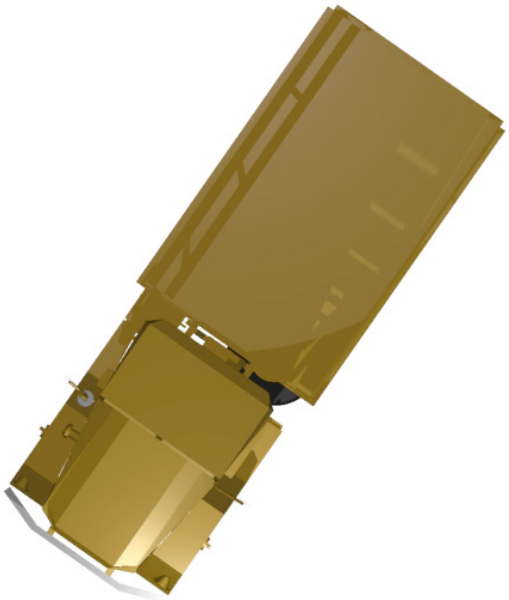
az=35, el=0



az=35, el=25

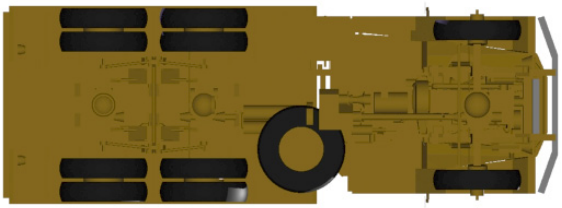


az=35, el=60



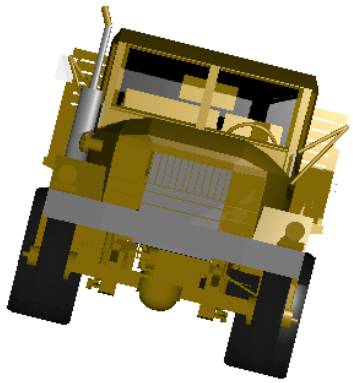
az=35, el=90

Top (az=270, el=90)



Bottom (az=270, el=-90)

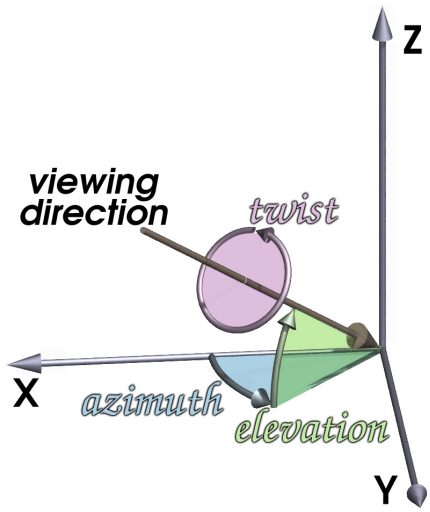
Finally, twist (which is an optional setting in MGED) specifies a rotation about the viewing direction. This rotation is applied to the view after azimuth and elevation have been designated. So, returning to our truck example, imagine standing in front of the vehicle (az=0, el=0) and then tilting your camera counterclockwise 14˚. This would give your view a 14˚ twist angle, as shown in the following figure (on the left). Note again that it is not the truck that is tipped up, but simply your view of it. For more information on specifying twist, see the **ae** command in Appendix A.



Front (az=0, el=0, tw=14)

4. Summing up on Azimuth and Elevation and the *xyz* Coordinate System

As mentioned at the start of this lesson, MGED operates in a three-dimensional coordinate system (determined by the *x*, *y*, and *z* axes). Azimuth is measured in the *xy* plane with the positive *x* direction corresponding to an azimuth of 0˚. Positive azimuth angles are measured from the positive *x* axis toward and past the positive *y* axis. Negative azimuth angles are measured in the other direction.



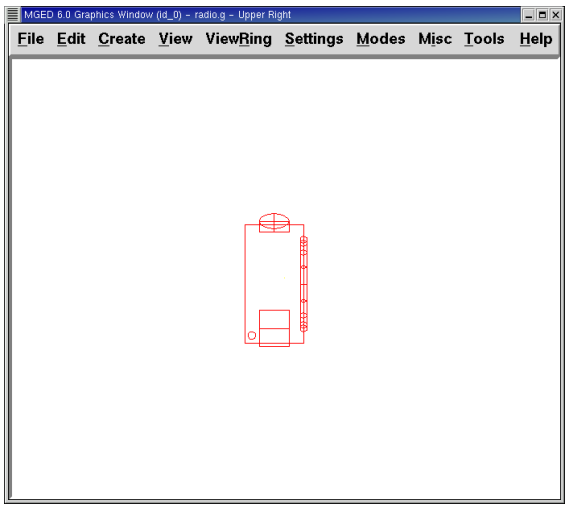
Azimuth, Elevation, and the xyz Coordinate System

If the azimuth angle is 0, then elevation is measured in the *xz* plane with +90˚ corresponding to the positive *z* direction and –90˚ corresponding to the negative *z* direction. However, if azimuth is not 0, these angles are in a plane aligned with the azimuth direction.

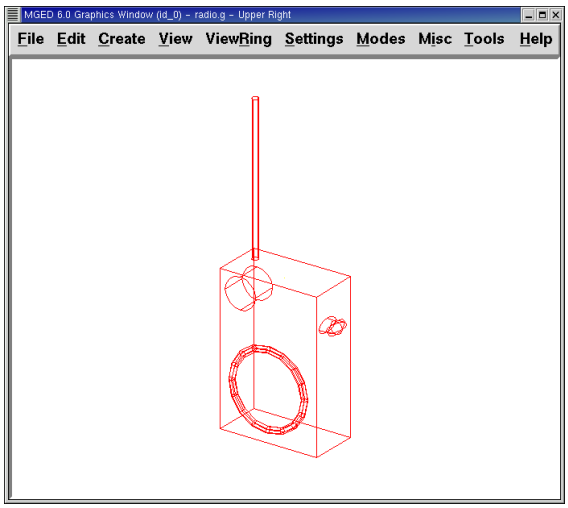
5. Viewing Your Radio from Different Angles

Let's now experiment with different views of your radio. MGED has several standard default views, which you've already seen in the preceding truck example. They include **Top** (az270, el90); **Bottom** (az270, el-90); **Right** (az270, el0); **Left** (az90, el0); **Front** (az0, el0); **Rear** (az180, el0); **az35, el25**; and **az45, el45**.

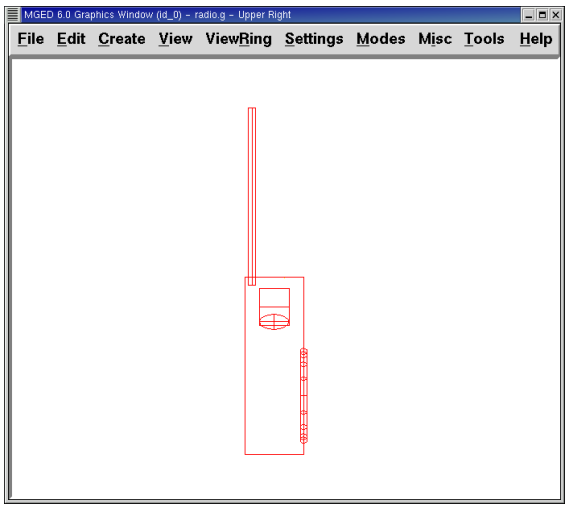
Go to the **View** menu and try viewing your radio from different angles.



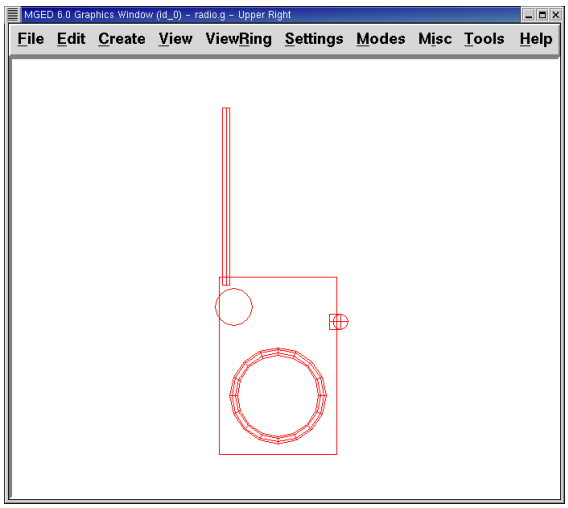
Top



az35,el25



Right



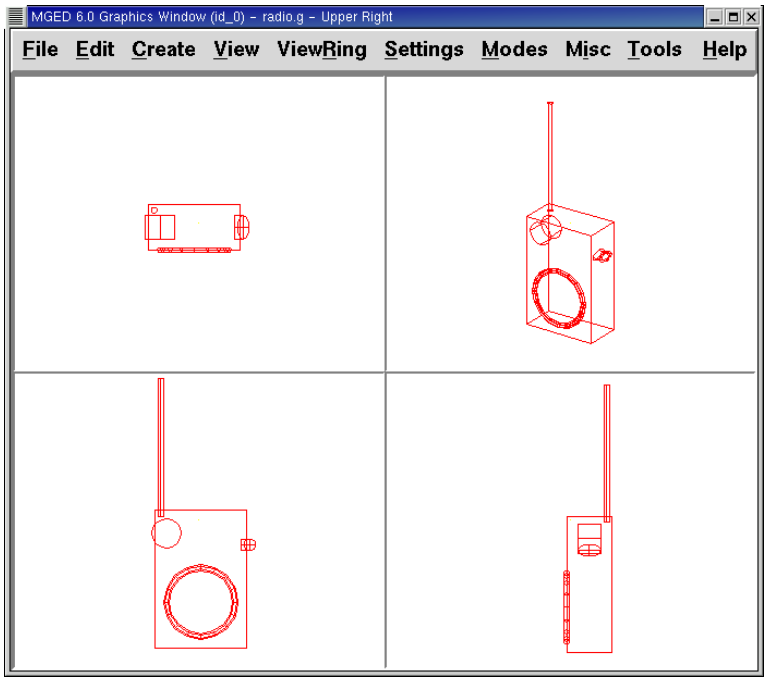
Front

You can also select any azimuth-elevation combination from the Command Line. For example, at the prompt type

```
ae 128 17<ENTER>
```

As with many of the Command Line options, this method of selecting views provides a finer degree of control/precision when you need it.

MGED can also display multiple views simultaneously. Go to the **Modes** menu and click on **Multipane**. Four small panes with different views should appear in your Graphics Window, as shown in the following illustration.



Multipane View of the Radio

6. Working with Shift Grips

The Shift Grip options of MGED are handy hot-key and mouse button combinations that can be used in two different ways. With regard to our present discussion on viewing, the Shift Grips can, in effect, "drag" the world around in front of the viewer (but without actually changing the coordinates of the viewed objects). The same Shift Grips can also be used in Edit mode to actually move or alter the geometry of your objects. In both cases, the Shift Grips appear to do the same thing, so it is important always to know the mode in which you are operating.

In general, the **SHIFT** key *translates* (moves), the **CTRL** key *rotates*, and the **ALT** key *constrains* (or limits) translation or rotation to a particular axis (*x*, *y*, or *z*). These axes

correspond to the three mouse buttons as follows: the *left* button represents the *x* axis, the *middle* button represents the *y* axis, and the *right* button represents the *z* axis. In addition, the **SHIFT** and **CTRL** keys can be used in conjunction with *any* mouse button to *scale* an object (although the **ALT** key will not constrain this action). The following table lists all of the key bindings and their functions.

Shift Grip Keys and Effects

Function	**Key Combination**	**Effect in Normal Viewing**	**Effect in Edit Mode**
Translate (Move)	SHIFT + *any* mouse button + mouse drag	Moves view in any direction	Translates object in any direction
Rotate	CTRL + *any* mouse button + mouse drag	Rotates view in any direction	Rotates object in any direction
Constrain Translation	SHIFT + ALT + *left* mouse button + mouse drag	Moves view in the *x* direction	Translates object in the *x* direction
	SHIFT + ALT + *middle* mouse button + mouse drag	Moves view in the *y* direction	Translates object in the *y* direction
	SHIFT + ALT + *right* mouse button + mouse drag	Moves view in the *z* direction	Translates object in the *z* direction
Constrain Rotation	CTRL + ALT + *left* mouse button + mouse drag	Rotates view about the *x* axis	Rotates object about the *x* axis
	CTRL + ALT + *middle* mouse button + mouse drag	Rotates view about the *y* axis	Rotates object only about the *y* axis
	CTRL + ALT + *right* mouse button + mouse drag	Rotates view about the *z* axis	Rotates object about the *z* axis
Scale	SHIFT + CTRL + *any* mouse button + mouse drag	Scales view larger or smaller	Scales object larger or smaller

Depending on your window manager or desktop environment settings, some key combinations may already be designated to perform other tasks (e.g., resizing or moving a window). If so, you may need to adjust settings to allow the Shift Grip options to function. Furthermore, left-handed users may have switched the behavior of the left and right mouse buttons in their system configurations. In such instances, the terms *left mouse button* and *right mouse button* should be switched throughout this document.

Probably the easiest way to familiarize yourself with the Shift Grip options is to try them out on your radio. Using the preceding table as a guide, experiment with translating, rotating, constraining translation and rotation to particular axes, and sizing your radio view.

	Remember, although the Shift Grip options may appear to be manipulating objects, unless you are in Edit mode they are only manipulating *your view* of the objects.

Review

In this lesson, you:

- Created a model radio.
- Located viewing information in the Command Window.
- Identified elements of the MGED viewing system.
- Viewed your radio from different angles.
- Worked with Shift Grips.

Lesson 3: Using the Insert Command to Size and Place Shapes

In this lesson, you will:

- Create a sphere and a right circular cylinder using the **make** command.
- Create the same two shapes using the **in** (insert) command.
- Combine arguments on the Command Line to streamline the entry of variables.
- Develop a combined-command form to help manage Command Line variables.
- Consider conventions for choosing names for your objects.
- View your shapes from different perspectives using options of the View menu.
- Quit the MGED program.

This lesson focuses on creating shapes from the Command Window using the **make** and **in** commands. You will create a sphere (sph) and a right circular cylinder (rcc) using both commands so that you can see how each command works. Later in the lesson, you will practice viewing your model from different angles.

Creating a New Database from the Command Window

Create a new database and name it **shapes.g**. Title your database **myShapes**.

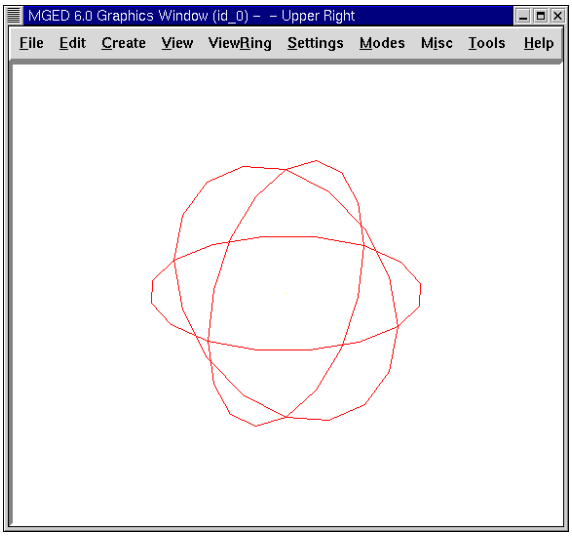
1. Creating a Sphere Using the Make Command

Begin by making the Command Window active (usually by clicking anywhere in the window). Then, at the MGED prompt, type in the command:

```
make sph1.s sph <ENTER>
```

As noted in Lesson 1, this command tells MGED to:

make	sph1.s	sph
Create a shape	Name it sph1.s	Make it a sphere



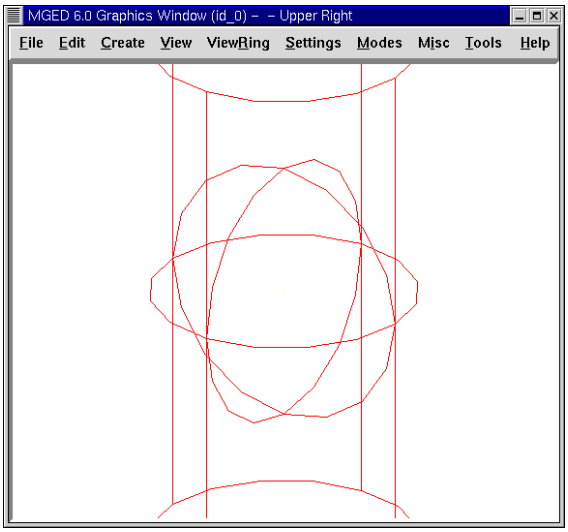
Wireframe Sphere

A sphere shape has now been created, and a wireframe drawing should appear in your Graphics Window.

To make the rcc from the Command Window prompt, type:

```
make rcc1.s rcc<ENTER>
```

Your Graphics Window should now display a large rcc that, from the default view of az35, el25, looks as if it intersects the sphere you previously created.



Wireframe Sphere and Right Circular Cylinder

Using the **make** command is a fast and easy way to create a shape; however, most models are going to require shapes that have specific parameters, such as height and radius. So, a more precise way to create these shapes is to use the **in** (insert) command.

2. Using the In Command to Create Shapes

Begin by making the Command Window active (usually done by clicking anywhere in the window). Then, use the **Z** (zap) command to clear the Graphics Window. You are now ready to create a sphere using the **in** command. At the MGED prompt type:

```
in sph2.s sph<ENTER>
```

MGED will respond with:

```
Enter X, Y, Z of vertex:
```

You must tell MGED where to position the vertex (center) of your sphere in space. Type at the MGED prompt:

```
4 4 4<ENTER>
```

	As you work in MGED, you will often be asked to enter a value for a *vector* or a *vertex*. In MGED, a vector represents the distance and direction from one point in space to another, and a vertex is one single point in space. The values entered for a vector are typically used to create an object with specific dimensions. The values entered for a vertex place the object in space.

Your sphere will now be placed at (x,y,z)=(4,4,4), as measured in millimeters. Notice that the numbers are separated by spaces followed by the **ENTER** key. MGED will now ask you to:

```
Enter radius:
```

Type in:

```
3<ENTER>
```

The radius of your sphere will be 3 mm. The following is the dialog that should appear in your Command Window (including the appropriate responses).

```
mged> in sph2.s sph
Enter X, Y, Z of vertex: 4 4 4
Enter radius: 3
51 vectors in 0.000543 sec
```

The last line of this dialog is simply a record of the computer's speed in drawing the shape. It has no real usefulness to the user at this point.

A sphere has now been created, and a wireframe drawing similar to the one created using the **make** command should appear in your Graphics Window.

To make the right circular cylinder, type at the Command Window prompt:

```
in rcc2.s rcc<ENTER>
```

MGED will ask you to enter values for *x*, *y*, and *z* of the vertex (where you want the center of one end of the rcc placed in space). Type:

```
4 4 0<ENTER>
```

Be sure to leave spaces between each of these numbers.

MGED will now ask you to enter the *x*, *y*, and *z* values of the height (H) vector (i.e., how long you want the rcc to be). Type:

```
0 0 4<ENTER>
```

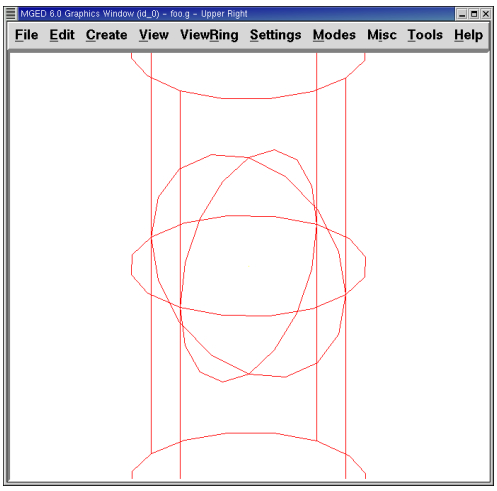
The last value you will need to enter is the radius of the rcc. Type:

```
3<ENTER>
```

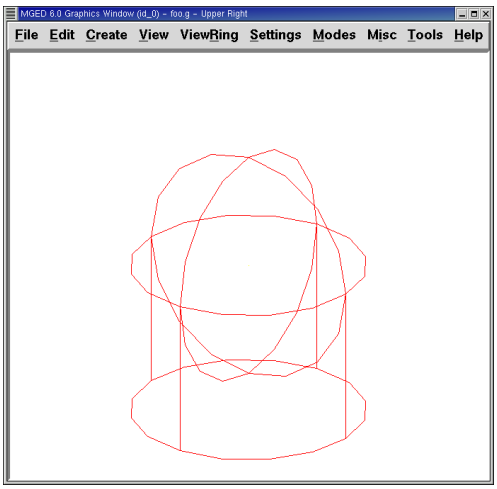
The dialog in the Command Window for the creation of the rcc should look like this:

```
mged> in rcc2.s rcc
Enter X, Y, Z of vertex: 4 4 0
Enter X, Y, Z of height (H) vector: 0 0 4
Enter radius: 3
42 vectors in 0.000214 sec
```

You should now have new versions of the sphere and rcc shapes. Notice how these two shapes compare in size to the first two you created. The rcc is now in proportion to the sphere and is placed in space off to the left in your Graphics Window. By specifying the dimensions of the shapes and their locations in space, you were able to create the model more precisely.



Shapes Created with Make Command



Shapes Created with In Command

3. Combining Arguments on One Line

Another way to use the **in** command is to combine all of the required information on one line. Once you become familiar with using the **in** command, you will probably prefer to use this method as it allows you to input all the parameter values more quickly.

Clear the Graphics Window by using the **Z** command. Now make another sphere by typing after the MGED prompt:

```
in sph3.s sph 4 4 4 3<ENTER>
```

The meaning of this longer form of the command is:

in	**sph3.s**	**sph**	**4**	**4**	**4**	**3**
Insert a primitive shape	Name it sph3.s	Make the primitive shape a sphere	Make the *x* of the vertex a value of 4	Make the *y* of the vertex a value of 4	Make the *z* of the vertex a value of 4	Make the radius a value of 3

To make the right circular cylinder using this method, type after the MGED prompt:

```
in rcc3.s rcc 4 4 0 0 0 4 3<ENTER>
```

The meaning of this command is:

in	**rcc3.s**	**rcc**	**4**	**4**	**0**	**0**	**0**	**4**	**3**
Insert a primitive shape	Name it rcc3.s	Make the primitive shape a right circular cylinder	Make the *x* of the vertex a value of 4	Make the *y* of the vertex a value of 4	Make the *z* of the vertex a value of 0	Make the *x* of the height vector a value of 0	Make the *y* of the height vector a value of 0	Make the *z* of the height vector a value of 4	Make the radius a value of 3
						Make the shape four units long, pointing straight toward positive *z*			

4. Making a Combined-Command Form for the In Command

When you are first starting to use MGED, if you want to use the Command Window rather than the GUI, you may want to make yourself some blank, combined-command forms for each type of primitive shape you will be creating. This can speed up the design process and help remind you of which values must be entered for each shape. A form for the sphere might be:

in	?	**sph**	?	?	?	?
Insert a shape	Name of primitive shape	Type of shape is a sphere	Value of *x*	Value of *y*	Value of *z*	Radius of sph
			Center			

A Combined-Command Form for the rcc might be:

in	?	**rcc**	?			?			?
Insert a primitive shape	Name of shape	Type of shape is a right circular cylinder	Value of *x*	Value of *y*	Value of *z*	Value of *x*	Value of *y*	Value of *z*	Radius of rcc
			Vertex			Height vector			

5. Considering MGED Naming Conventions

You may have noticed that each time you have created a sphere, or rcc, you have given it a different name. MGED doesn't care what name you give a shape, but you will find as you develop models that it helps to have some formula, or conventions, when naming shapes. Note also that each name must be unique in the database, and for BRL-CAD releases prior to 6.0, names are limited to 16 characters in length.

In this lesson, we sometimes assigned names to the shapes based on their shape type and the order in which we created them. We did this because the shapes had no real function, except to be examples.

When you create real-life models, however, you will probably want to assign names as we did for the radio component names, which were based on their functions (e.g., **btn** for button, **ant** for antenna, etc.).

If you work with more experienced modelers, check with them to see what set of conventions they use. If you work alone, develop a set of naming conventions that works for you and then use it consistently.

6. Viewing the Shapes

Practice viewing your new shapes using the **View** menu. Manipulate your view using the various mouse-key combinations identified in the previous lesson.

7. Quitting MGED

If you wish to quit MGED, at this point, type either the letter **q** or the word **quit** after the Command Window prompt and then press **ENTER**. You may also quit the program by selecting **Exit** from the **File** menu.

Review

In this lesson, you:

- Created a sphere and a right circular cylinder using the **make** command.
- Created the same two shapes using the **in** (insert) command.
- Combined commands to streamline the entry of variables.
- Developed a combined-command form to help manage Command-Line variables.
- Considered MGED naming conventions.
- Viewed your shapes from different perspectives using options of the View menu.
- Quit the MGED program.

Intentionally Left Blank

Lesson 4: Assigning Material Properties and Raytracing

In this lesson, you will:

- Recall primitive shapes made previously.
- Make a region of two primitive shapes.
- Assign material properties to your primitive shapes from the Command Window.
- Clear the Graphics Window and draw the new region.
- Raytrace your design from the GUI.
- Use the GUI to change layers of the Graphics Window.
- Clear the Graphics Window after raytracing a model.

1. Opening the Database

To recall the primitive shapes made in the previous lesson, start MGED and go to the **File** menu and select **Open**. A control panel will appear with a list of folders and files. Select **shapes.g** and **Open**. A new box will appear, and you should click on **OK**.

You should now have two windows prominently displayed on your screen. At the MGED prompt in the Command Window, type:

```
draw sph2.s rcc2.s<ENTER>
```

2. Creating a Region

Before you can raytrace your design, you have to make a *region* of the two shapes. A region is an object that has uniform material properties. Most applications that use BRL-CAD models consider regions as the basic components of the model. Regions are constructed using the basic Boolean operations of *union*, *intersection*, and *subtraction*, which are discussed in the next chapter.

At the MGED prompt, type:

```
r shapes2.r u sph2.s u rcc2.s<ENTER>
```

*[Note: Make sure you key in the information correctly before you press **ENTER**. Spaces, or the lack thereof, are important.]*

This command tells MGED to:

r	shapes2.r	u	sph2.s	u	rcc2.s
Make a region	Name it shapes2.r	Add the volume of the shape	sph2.s	Add the volume of the shape	rcc2.s

3. Assigning Material Properties to a Region

Now type in:

mater shapes2.r<ENTER>

MGED will respond with:

```
shader=
Shader?  ('del' to delete, CR to skip)
```

Type in:

plastic<ENTER>

MGED will ask:

```
Color = (no color specified)
Color R G B (0..255)? ('del' to delete, CR to skip)
```

Type in:

0 255 0<ENTER>

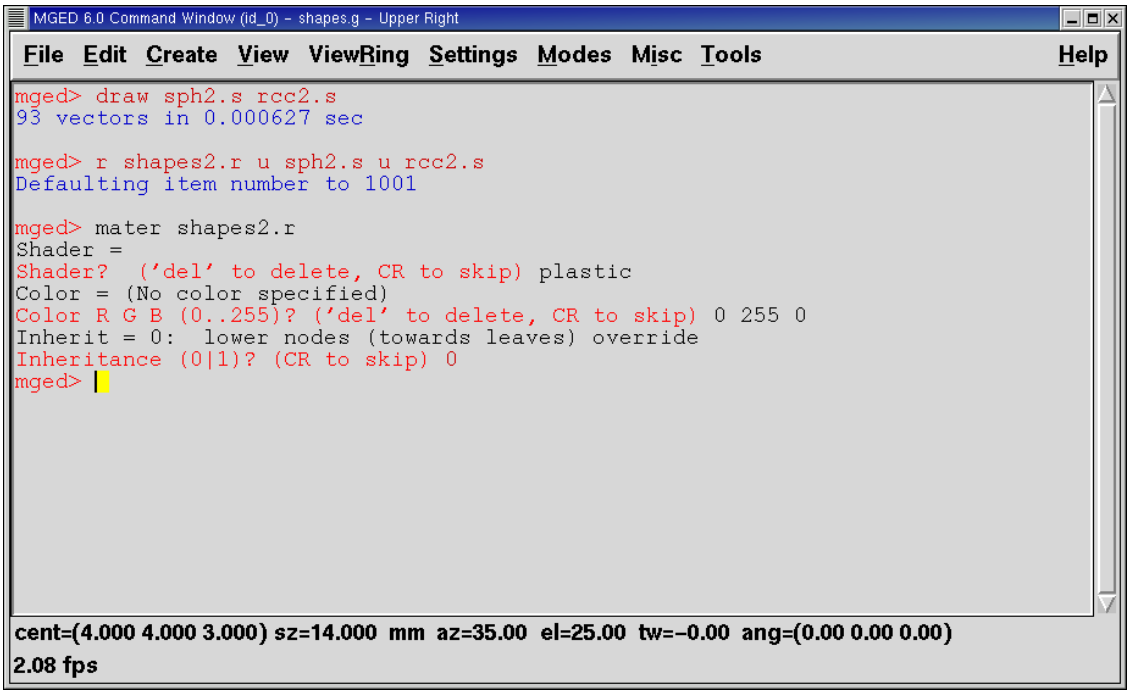
This will assign a light green color to the region. MGED will now ask:

```
Inherit = 0  lower nodes  (towards leaves)  override
Inheritance (0/1)? (CR to skip)
```

Type:

0<ENTER>

When you have completed this set of commands, your Command Window should look like the following example:



The Command Window Screen

The one-line version of this set of commands would be:

`mater shapes2.r plastic 0 255 0 0<ENTER>`

Diagrammed, this command says to:

mater	shapes2.r	plastic	0 255 0	0
Assign material properties to:	the region called shapes1.r	Make the region of plastic	Give it a color of light green	Do not inherit colors or material type

4. Clearing the Graphics Window and Drawing the New Region

An easy way to clear the Graphics Window of the old design and draw the new region is to type at the MGED prompt:

`B shapes2.r<ENTER>`

This command tells MGED to:

B	shapes2.r
Blast (clear) the Graphics Window and draw	The region named shapes2.r

The **Blast** command is shorthand for the combination of the **Z** and **draw** commands.

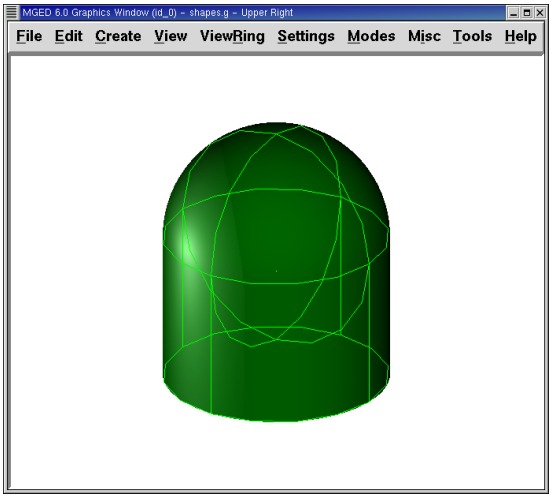
5. Raytracing Your Model

Now go to the **File** menu and select **Raytrace**. A dialog box called the **Raytrace Control Panel** will appear. At the top are menus for **Framebuffer** and **Objects**. Select **Framebuffer**. A drop-down menu will appear with six choices: **Active, All, Rectangle Area, Overlay, Interlay**, and **Underlay**. Make sure the **Active**, **All**, and **Underlay** options are activated (as shown by the presence of a red indicator to the left of each choice). Select **OK**.

	Note: When you select **Raytrace** or **OK** from this dialog window, you start an auxiliary program (**rt**) of the BRL-CAD package. The program only raytraces objects that have been drawn in the Graphics Window. You may have many shapes, regions, or combinations in a database, but if they aren't currently drawn in the Graphics Window, the raytracer will ignore them.

Change the background color produced by the raytracer by selecting **Background Color** in the **Raytrace Control Panel**. A drop-down menu will appear with some predefined color choices plus a color tool. Select **white**. The select button should now appear white, in accordance with your selection.

Next select **Raytrace** from the four options along the bottom of the box. The Graphics Window should start changing, and you will soon see your design in shades of green with the wireframe superimposed on the design, as shown in the following example:



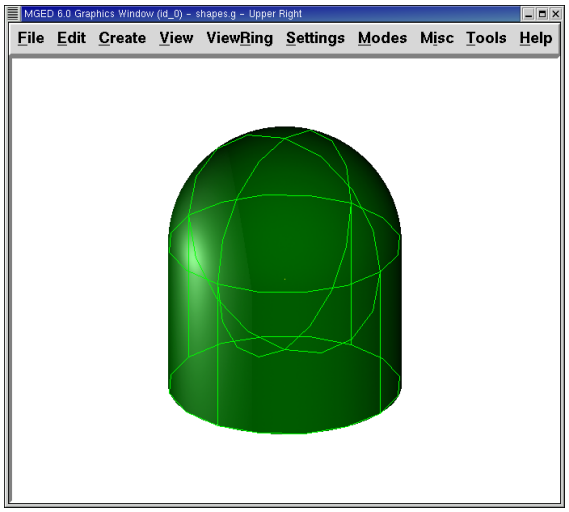
Raytraced Shapes

As we have seen, you can specify the background color for the raytraced image. You can also fill the entire framebuffer with the background color. To do this, select the desired color and then click the **fbclear** (framebuffer clear) button at the bottom of the Raytrace Control Panel.

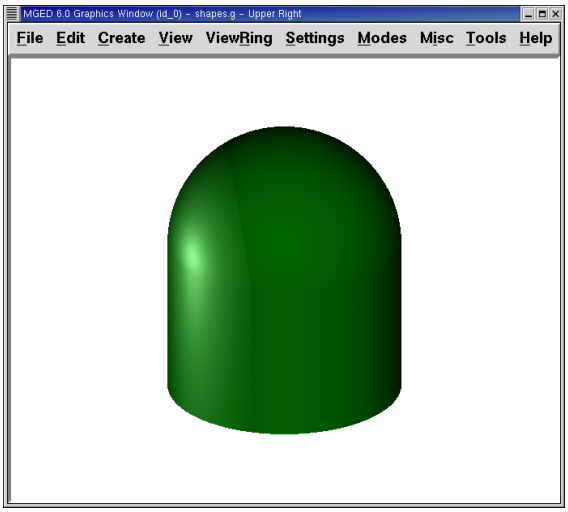
6. Changing Layers of the Graphics Window

The Graphics Window of MGED is used to display different types of graphical information: 3D wireframes and 2D pixels (or images). Conceptually, each type of data occupies a separate *layer* in the display. The 3D wireframes occupy the wireframe layer, while the 2D pixels (images) occupy the framebuffer layer. These layers can be thought of as transparencies, and the order in which they are stacked and displayed can be changed.

Within the Raytrace Control Panel, there is a **Framebuffer** menu. At the top of this menu is a toggle button labeled **Active**. This turns the display of the framebuffer layer on and off. Near the bottom of the same menu are three radio buttons: **Overlay**, **Interlay**, and **Underlay**. When the underlay mode is selected, the pixel data are displayed under or behind the vector data. Conversely, when the overlay mode is selected, the pixel data are in front of the vector data. The interlay option is similar to the overlay mode. The subtle difference is an advanced topic not covered here.



Framebuffer in Underlay Mode



Framebuffer in Overlay Mode

To see how this works, go to the framebuffer menu and select **Overlay**. Notice that the wireframe representation disappears. Where does it go? If you answered "behind the framebuffer," you would be correct. To view the model's geometry, you would have to make the framebuffer inactive or select underlay mode.

The wireframe layer has a yellow dot in the center that marks the center of the view talked about in Lesson 2. This allows you to determine whether the framebuffer is in

overlay or underlay mode. If you can see the yellow dot, the framebuffer is in underlay mode. If you've told MGED to draw some geometry and the Graphics Window seems to remain blank, you are probably seeing a blank framebuffer masking the wireframe layer.

Note that you can change the view in the wireframe, but the view in the framebuffer does not automatically update to match. It is not possible to directly manipulate the view in the framebuffer. You must raytrace again in order to update the framebuffer image.

7. Clearing the Graphics Window

To completely clear the Graphics Window, you have to handle both the wireframe and framebuffer layers. Recall that you can clear the wireframe layer with the **Z** command. For the framebuffer layer, there is the **fbclear** button on the Raytrace Control Panel.

In some instances, you may prefer to turn off the framebuffer instead of clearing it. When the framebuffer is turned off, MGED runs faster because it doesn't have to redraw the framebuffer each time it updates the display. You can turn the framebuffer on and off by toggling the **Active** item in the Raytrace Control Panel's framebuffer menu.

	Note that in BRL-CAD versions 5.1 and later, turning off the framebuffer does not destroy the image it contains. Turning it back on displays the same image. However, in earlier versions of the package, the contents of the framebuffer are lost when it is turned off.

Review

In this lesson you:

- Recalled primitive shapes made previously.
- Made a region of two primitive shapes.
- Assigned material properties to your primitive shapes from the Command Window**.**
- Cleared the Graphics Window and draw the new region.
- Raytraced your design from the GUI.
- Used the GUI to change layers of the Graphics Window.
- Cleared the Graphics Window after raytracing a model.

Lesson 5: Learning About Boolean Expressions

In this lesson, you will:

- Learn about combinations and regions.
- Learn about Boolean operations.
- Make regions with Boolean operations.

	This is an important lesson because Boolean operations are critical to the modeling process. The order in which shapes are combined and the operators used to combine the shapes will determine how the MGED program interprets your model. The correct use of Boolean expressions to modify geometric shapes is a key skill in constructive solid modeling. It important to review these concepts as many times as necessary. If it is difficult to absorb them all now, come back to them later.

1. Combinations and Regions: Boolean Tools

There are conceptually two objects in MGED that support Boolean operations. One is called a *combination,* the other is called a *region*.

As mentioned earlier, a typical geometric shape in MGED is called a *primitive*. However, single primitives are often insufficient to fully describe the complex shape of the object being modeled. So, combining two or more primitive shapes into other shapes (called *combinations*) using Boolean operators allows you to artfully imitate the shape and form of most complicated objects.

The previous chapter noted that material properties are associated with regions. Like combinations, regions use Boolean operations to create complex shapes. The difference is that regions are shapes that have material properties. They occupy three-dimensional space, rather than simply defining a shape in space.

You can think of primitives and combinations as a blueprint for an object. The actual object is created when a region is made. For example, you might make a blueprint of an object such as a coffee mug, but then create that mug from different types of material (e.g., ceramic or glass). Regardless of the material, the blueprint is the same.

When Boolean operations are used to build up complex shapes from simpler shapes, we can call the result a *shape combination*. When they are used to define other logical or hierarchical structure within the database, the result may be referred to as a *group* or an *assembly combination*.

2. Boolean Operations

The three Boolean operators employed by the MGED program are *union, subtraction*, and *intersection*. You can use Boolean operations to combine shapes to produce more complex shapes.

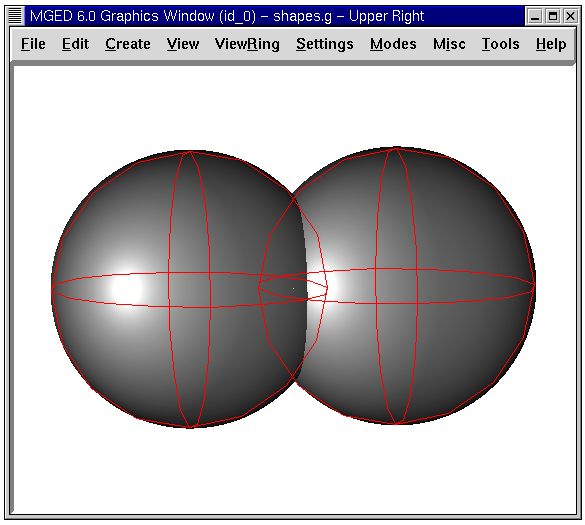
1. Union Shapes: Merge two shapes.

2. Subtract Shapes: Remove the volume of one shape from another.

3. Intersect Shapes: Use only the parts of the two shapes that overlap.

Union

The union operator, **u**, joins shapes so that any point in at least one of them will be part of the result. Union is a powerful and frequently used operator.

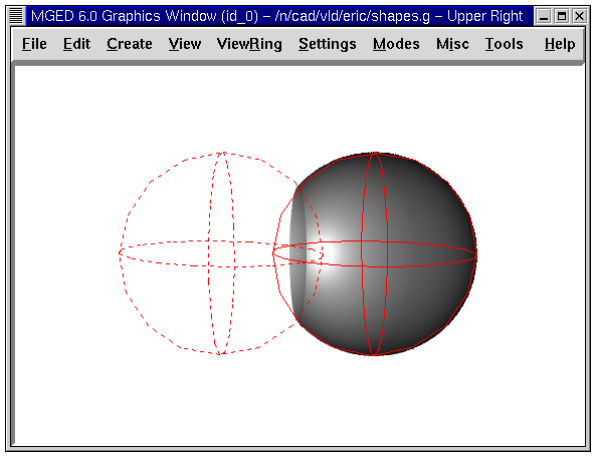


The Union of Two Spheres

Subtraction

When a primitive shape has a second, overlapping shape subtracted from it, the result is that the second shape disappears, together with any common volume it had with the first shape. The – (minus sign) operator signifies subtraction or difference. This operation is especially useful in hollowing a body, removing an oddly shaped piece of a primitive shape, or accounting for edge intersections of walls, plates, piping, or other connected shapes.

In the following example, a dotted red line indicates that the sphere being subtracted extends inside the sphere on the right. This overlapping portion is partially out of view in the raytraced image.



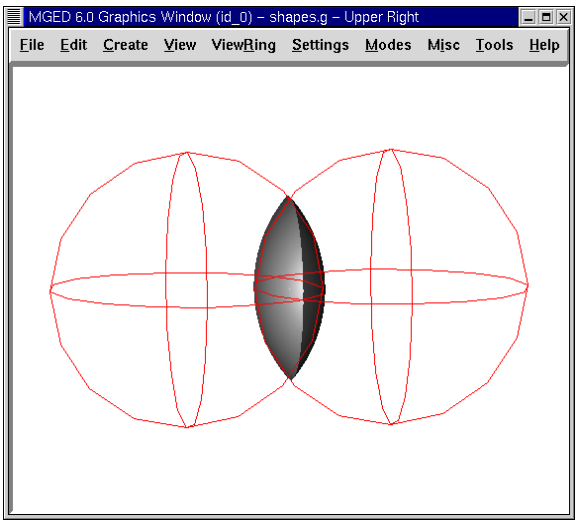
Subtraction of One Sphere from Another Sphere

Intersection

The Boolean intersection operation, signified by a + (plus sign) operator, combines two primitive shapes that overlap each other, saving only their common volume (the nonoverlapped areas will not be present). An easy way to understand intersections is to think of shapes as roads. The intersection is the place where two roads overlap.

Although many people find intersection operations harder to understand than unions and subtractions, unusual/complex shapes can be expressed using the intersection operator. For example, you can model a magnifying lens as the intersection of two spheres.

The intersection operation is rarely useful unless, as shown in the following figure, at least two shapes overlap. The intersection of two shapes having no common points (i.e., no overlap) is the null set, so it includes no points of space at all.



Intersection of Two Spheres

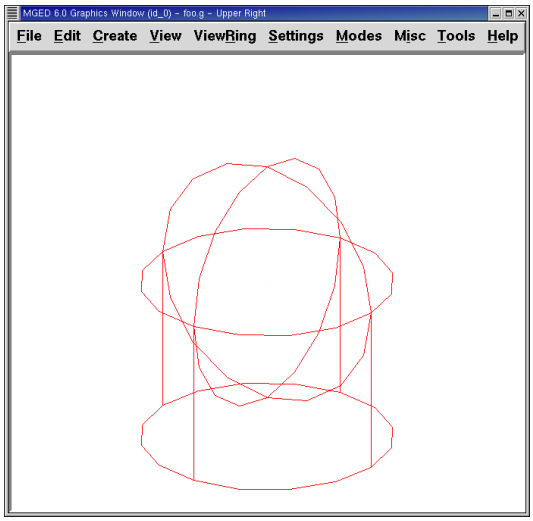
There is one important restriction when using the Boolean subtraction and intersection operators. There must be a first shape from which a second shape can be subtracted or intersected. If you have only one shape within a region or combination, the operator will be ignored and the union operator will always be used.

3. Making Regions with Boolean Operations

Begin by opening the database **shapes.g** that you created in Lesson 3. At the Command Window prompt, type:

```
draw sph2.s rcc2.s<ENTER>
```

This lets us see the shapes we will be using to create our regions. As seen earlier, the two shapes should look something like the following:



Two Primitive Shapes

In this lesson, we will create different shapes to demonstrate the function of Boolean operations. In the Command Window, type the following:

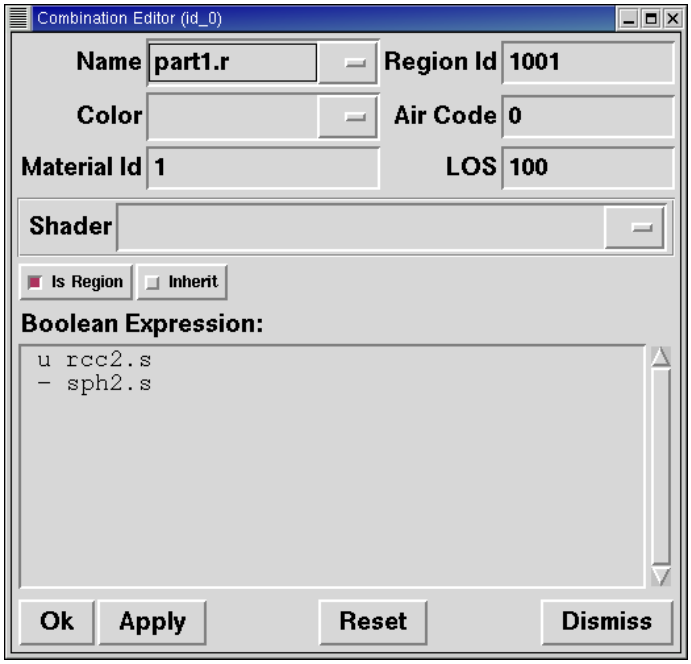
```
r part1.r u rcc2.s - sph2.s<ENTER>
```

This command tells MGED to:

r	part1.r	u	rcc2.s	-	sph2.s
Make a region	Call it part1.r	Merge…	The shape named rcc2.s	Subtract…	The shape named sph2.s

Note: The first member always has a lowercase **u** for an operator. The second and subsequent members can use –, +, or **u** as needed. The process of determining which operators to use, and in what order, is discussed in a more advanced tutorial.

In the previous lesson, we applied material properties to objects from the Command Line. Now we are going to use the graphical interface to do the same thing. From the **Edit** menu, choose **Combination Editor.** This will pop up a dialog box. Click the button to the right of the **Name** entry box and then click on **Select from All**. A drop-down menu will appear with the regions you have created. Select **part1.r**. The result should look like the following:



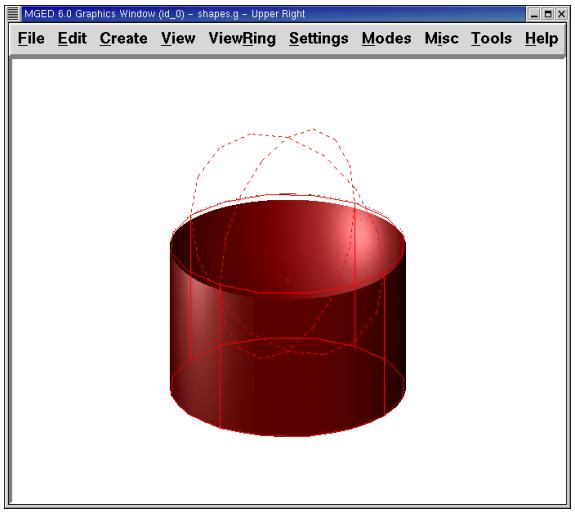
Combination Editor

Click on the button next to **Color** and select **red** from the pull-down menu.
Now click the **OK** button at the bottom left of the dialog window. This will apply your changes and close the panel.

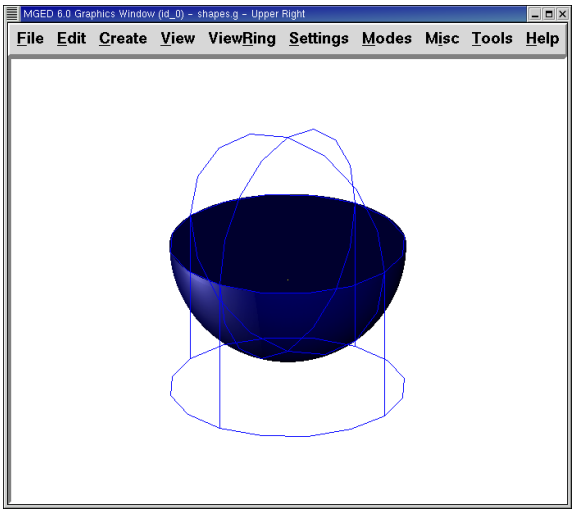
At the moment, we have only the primitive shapes displayed, not the region. Before we can raytrace, we must remove the primitive shapes from the display, and draw the region. Otherwise, we will not be able to see the region with the color properties we applied. We can do this by typing:

```
B part1.r
```

We are now ready to raytrace this object. From the **File** menu, bring up the **Raytrace Control Panel** and click the **Raytrace** button. The image you get should look similar to the left-hand image that follows. Note that it may take several minutes to raytrace the window, depending on the speed of your particular system.



Raytraced part1.r



Raytraced part2.r

You should see that a spherical "bite" has been taken out of the top of the cylinder.

Next we will make a blue region using the intersection operator instead of subtraction. Once again, we start by creating a region:

```
r part2.r u rcc2.s + sph2.s<ENTER>
```

For comparison to the GUI approach used to make **part1.r**, let's use the Command Line to assign the color to **part2.r**:

```
mater part2.r plastic 0 0 255 0<ENTER>
```

Finally, Blast this new region onto the display as follows:

```
B part2.r<ENTER>
```

Now raytrace the object. It should look similar to the preceding right-hand image.

	Note: Remember to clear the Graphics Window and draw your new region or combination before trying to raytrace the model. The raytracer ignores a region or combination that is not drawn in the Graphics Window. The color of the wireframe is your clue. If it doesn't reflect the colors you've assigned (e.g., everything is drawn in red even though you've assigned other colors), then you haven't cleared the screen of the primitive shapes and drawn the new region or combination since the time you made it.

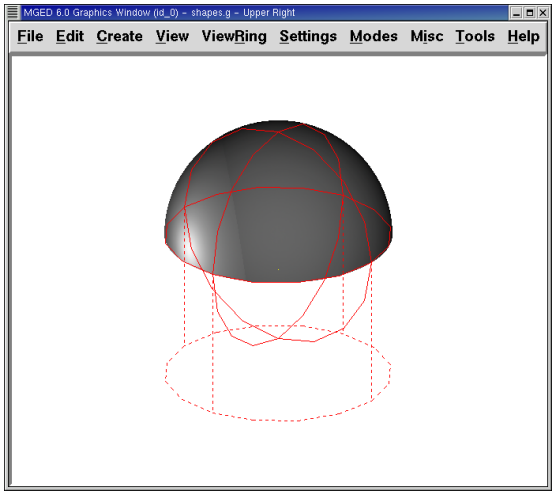
When you use the intersection operator, the order in which you specify the shapes doesn't matter. We would have gotten the same results if we had specified the Boolean operation as

```
r part2.r u sph2.s + rcc2.s
```

However, when using the subtraction operator, the order of the two shapes is very important. Let's make a region with the order of the shapes reversed from that used for **part1.r**:

```
r part3.r u sph2.s - rcc2.s
```

This time we won't bother to set a color. (When no color is set for objects, the raytracer (rt) will use a color of white. However, these objects may appear gray because of the amount of light in the scene.) Blast this design to the display and raytrace it:



Raytrace part3.r

Now let's raytrace all three objects we have created together. To draw the three regions at once, we could type:

```
B part1.r part2.r part3.r
```

Doing this once is no problem. However, if these were three parts that made up some complex object, we might like to be able to draw all of them more conveniently. To make drawing a collection of objects together easier, we create an *assembly combination* to gather them all together. We will create one called **dome.c** for our three regions. This is accomplished by the following command:

```
comb dome.c u part1.r u part2.r u part3.r
```

Notice the similarity between this command and the **r** command we used to create the regions.

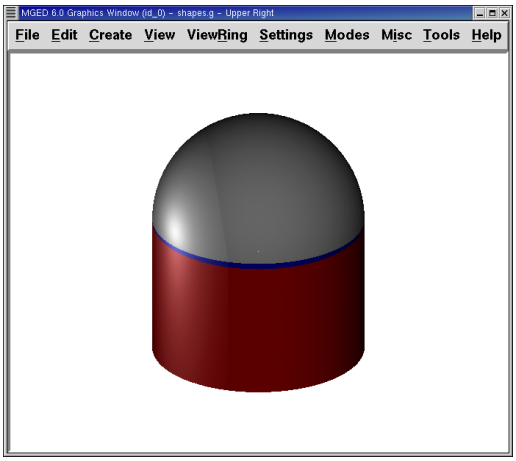
Remember from the discussion at the beginning of this lesson, the difference between a region and a combination is that combinations are not necessarily composed of only one kind of material. Several objects of different materials can make up an *assembly combination* such as the one we have just created.

	Because creating assembly combinations is a very common task, there is a shortcut command—the **g** (for group) command—to help make the task easier. Creating **dome.c** using this command would look as follows: **`g dome.c part1.r part2.r part3.r`** Notice that you don't have to type the **u** Boolean operators. The **g** command unions all of its arguments.

All that is necessary to draw all three objects is the much simpler command:

`B dome.c`

Now we can raytrace the collected set and get the following image:



Raytraced dome.c

5. Operator Precedence

The shapes we have created here are fairly simple. In each case, a single primitive shape is unioned, and subtraction or intersection operations are performed on that single primitive shape. You should know that it is possible to use much more complex Boolean equations to create the shape of objects. When you want to make such objects, keep in mind the precedence of the Boolean operations. In the Boolean notation we are using, the subtraction and intersection operators both have higher precedence than the union operator has. So, for example:

```
comb demo.c u shape1 - shape2 u shape3 - shape4 + shape5
```

This would result in the following Boolean expression:

```
(shape1 - shape2) u ( (shape3 - shape4) + shape5)
```

Review

In this lesson, you:

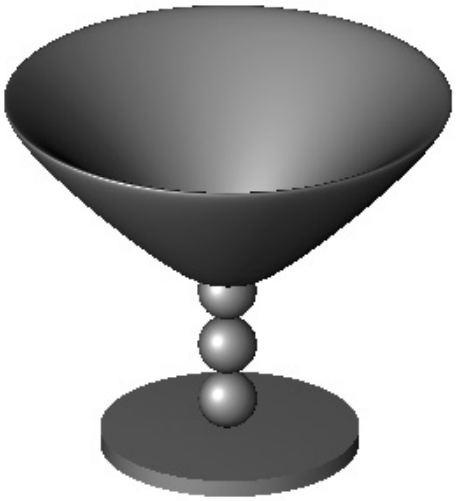
- Learned about combinations and regions.
- Learned about Boolean operations.
- Made regions with Boolean operations.

Lesson 6: Creating a Goblet

In this lesson, you will:

- Create a new database.
- Create, edit, and copy primitive shapes to make the parts of the goblet.
- Make regions of the parts.
- Make a combination of the regions.
- View a data tree.
- Raytrace your completed goblet.

In this lesson, you will create a goblet similar to the one in the following example.



Raytraced Goblet

1. Creating a New Database

First, start MGED from the shell prompt. Select **File** from the menu bar and then **New**. A dialog box will appear, and it will ask you for a new database name. Type in **goblet.g** at the end of the path name and click on **OK** to create the new database. The program should tell you that the database was successfully created and it is using millimeters for its unit of measure.

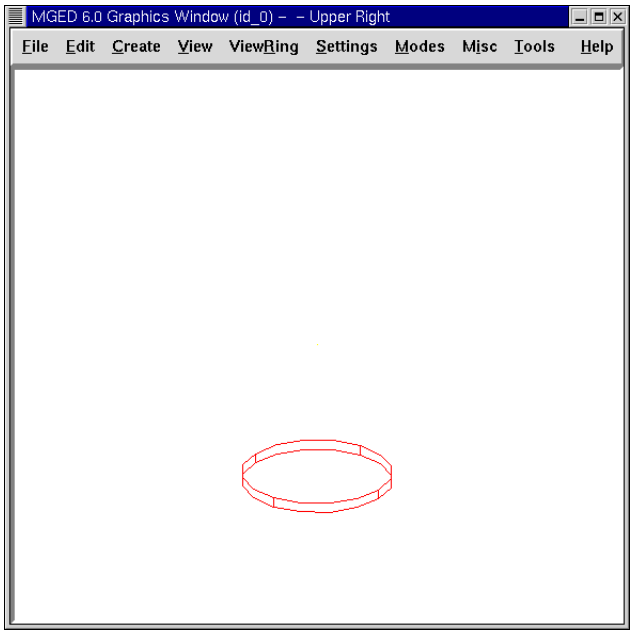
2. Creating, Editing, and Copying the Parts of the Goblet

Creating the Goblet Base

Go to the menu bar and select **Create** and then **Make Solid**. Next, select **rcc** for right circular cylinder. A dialog box will appear asking you to name the rcc. Type in **base1.s** and then click on **Apply** (or press **ENTER**). A tall cylinder will appear in the Graphics Window that is ready for you to edit.

Editing the Base

Go to the menu bar and select **Edit** and then **Set H**. Place the mouse pointer in the *lower half* of the Graphics Window and click on the *middle* mouse button several times. The cylinder will become shorter as you click. (Note that the closer your pointer is to the midpoint of the Graphics Window, the smaller the change will be. As you click farther away from the middle, the changes will be greater.) Continue clicking until the cylinder looks like a flat disk, as shown in the following figure. Click on **Accept** when done.

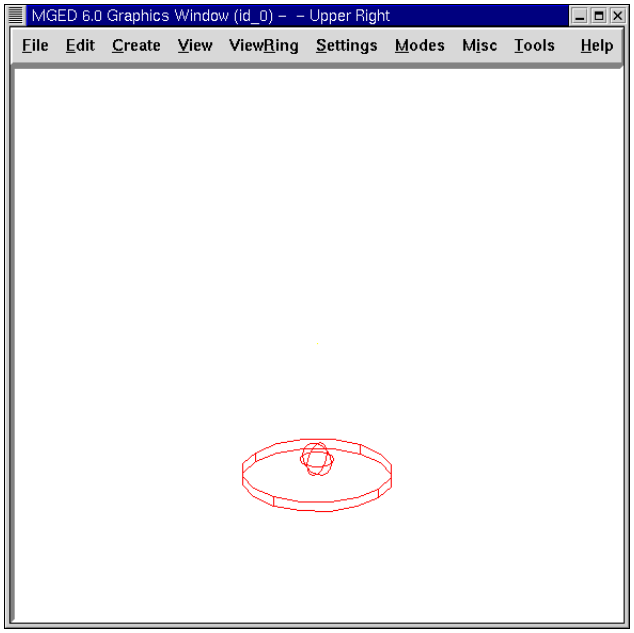


The rcc Goblet Base

Creating the Goblet Stem

Go to the menu bar and select **Create** and then **Make Solid**. Click on **sph** to select a sphere. You will be asked to provide a name for the sphere. Type **ball1.s** in the name box and then click on **Apply**. A large sphere will appear in your Graphics Window.

Go to the **Edit** menu and click on **Scale**. Place the mouse cursor/pointer in the *lower half* of the Graphics Window and click the *middle* mouse button until your sphere is about one-quarter the diameter of the base, as shown in the illustration that follows.



First Sphere on Goblet Stem

To move the ball on top of the goblet base, press the **SHIFT** key and *left* mouse button to drag the sphere into place. You can check your placement by going to the **View** option of the menu bar and selecting a **Front** view. In this view, you can align the center line of the sphere with the center line of the rcc. Repeat this process from a **Left** view. When you believe the sphere is correctly aligned with the rcc, go back to the **Edit** option and click on **Accept**.

Adding Additional Balls to the Goblet Stem

The next step is to add more spheres to your goblet stem. An easy way to do this is to go to the **Edit** menu and select **Solid Editor**. A dialog box will appear. Enter the name for the first sphere you created, **ball1.s**. Next, click on **Reset** (to reset the values of the dialog box to those of **ball1.s**) or hit return in the **Name** box. Again in the **Name** box, change **ball1.s** to **ball2.s** by using the **BACKSPACE** key to erase the **1**. Type in a **2** and then click on **Apply**.

Repeat this process with an sph named **ball3.s**. When you are done, click on **OK** to close the Solid Editor box. You now have three balls for your stem, but you won't be able to see them until you edit them because they are in the same place.

An even easier way to make the copies is to use the **cp** (copy) command as follows:

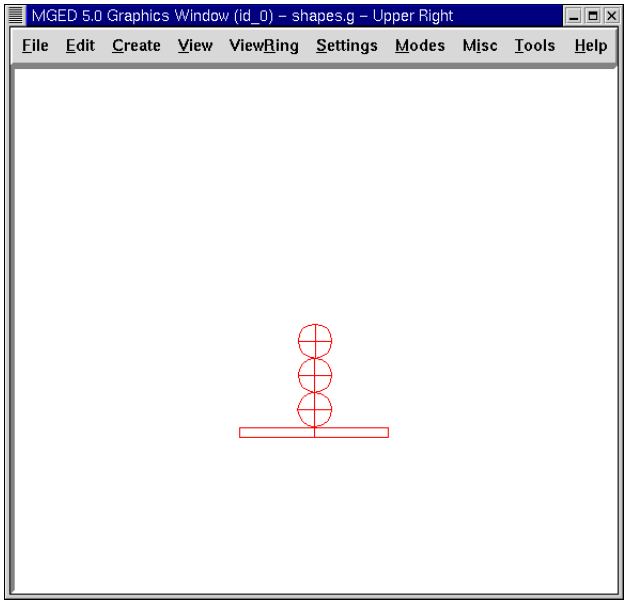
```
cp ball1.s ball2.s<ENTER>
cp ball1.s ball3.s<ENTER>
```

Editing the Balls of the Goblet Stem

To edit the new balls you have created, go to the **Edit** menu and click on **Solid Selection**. A box will appear with the names of your base and balls. Double click on **ball2.s** to select it. You will see the first ball change color to white. Use the **SHIFT** key and *any* mouse button to drag this ball (which is really **ball2.s**) so that it rests on top of (and slightly overlaps) **ball1.s** . Check your views to align the ball as you did with the first ball. (Note that this alignment is even easier if you drag using the **SHIFT** and **ALT** keys and the *right* mouse button, which will constrain the movement of the ball to the Z direction.) Click on **Accept** under the Edit option when finished.

If you were modeling a real goblet, you would want the balls of the stem to overlap slightly. If they barely touch, the stem would be very weak. If they do not touch, then the stem would be made of separate pieces of material suspended in space.

Repeat these steps to move **ball3.s** into position. When you are finished, your goblet should look as follows from a front view:

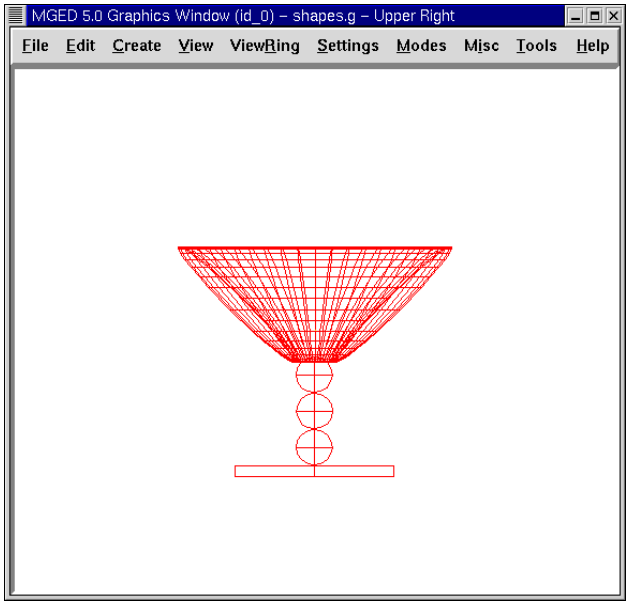


Goblet with Ball Stem

Making the Goblet Basin

The next step is to make the goblet's basin. Go to the **Create** menu and select **Make Solid**. Click on **eto** to select an elliptical torus. Name the torus **basin1.s**. Click on **Apply**. A large eto will appear in your Graphics Window.

Go to the **Edit** menu and select **Set C**. Place the mouse arrow in the *upper half* of the Graphics Window and click on the *middle* mouse button until your eto is approximately the size of the one in the following figure. If you need to, use **Scale** to make the basin more proportional to the rest object and use the Shift Grips and multiple views to position the basin.



Goblet Base, Stem, and Basin – Front View

3. Making Regions of the Goblet's Base, Stem, and Basin

In order for MGED to know what primitives to raytrace, you must first designate these areas through Boolean operations. In this example, the two Boolean operations used will be the union (u) and the subtraction (-).

To make the stem a region, type at the Command Window prompt:

```
r stem1.r u ball1.s u ball2.s u ball3.s<ENTER>
```

To make the base a region, type at the prompt:

```
r base1.r u base1.s - ball1.s<ENTER>
```

To make the basin a region, type at the prompt:

`r basin1.r u basin1.s - stem1.r<ENTER>`

Note that when creating **base1.r**, we subtracted a primitive shape from another primitive shape. When creating **basin1.r**, we subtracted an entire region from a primitive shape.

4. Making a Combination of the Regions

To combine all the regions into one object, you will need to perform one last Boolean operation. At the prompt in the Command Window, type:

`comb goblet1.c u basin1.r u stem1.r u base1.r<ENTER>`

This operation tells the MGED program to:

comb	goblet1.c	u	basin1.r	u	stem1.r	u	base1.r
Make a combination	Name it goblet1.c	The combination will be made of a union of	the region basin1.r	and	the region stem1.r	and	the region base1.r

5. Viewing a Data Tree

MGED requires a certain logical order to the model data tree so it knows how to raytrace the various elements. In the goblet, the base and basin consist of regions composed of only one primitive shape each. The stem, in contrast, consists of a region composed of the union of three spheres. The three regions were combined to form a complex object. To view the data tree for this combination, type at the Command Window prompt:

`tree goblet1.c<ENTER>`

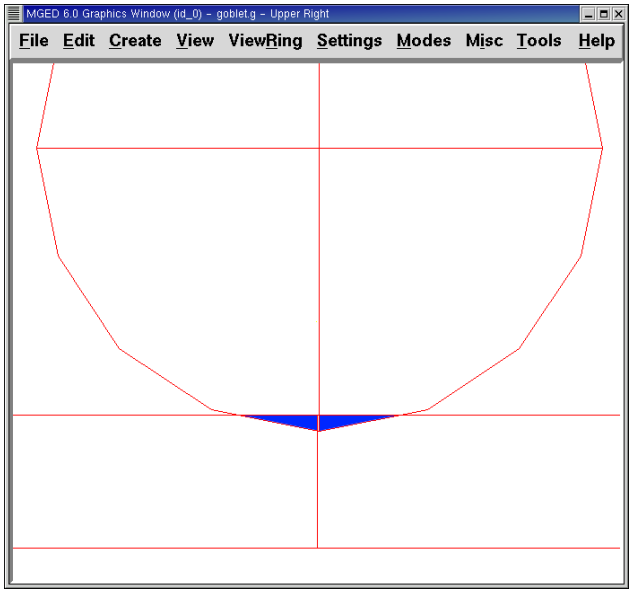
MGED will respond with:

```
goblet1.c/
        u basin1.r/R
                u basin1.s
                - stem1.r/R
                        u ball1.s
                        u ball2.s
                        u ball3.s
        u stem1.r/R
                u ball1.s
                u ball2.s
                u ball3.s
        u base1.r/R
```

```
u base1.s
- ball1.s
```

The name of the overall combination of this design is goblet1.c. It is composed of the three regions: base1.r, stem1.r, and basin1.r. The region base1.r is composed of the primitive shape named base1.s minus ball1.s. The region stem1.r is composed of three primitive shapes named ball1.s, ball2.s, and ball3.s. The region basin1.r is composed of the primitive shape named basin1.s minus the region stem1.r.

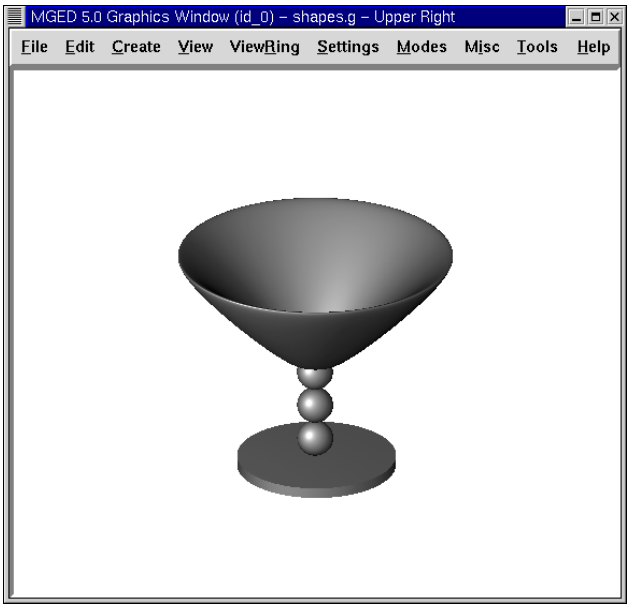
Remember that regions define volumes of uniform material. In the real world (and in BRL-CAD), no two objects can occupy the same space. If two regions occupy the same space, they are said to *overlap*. To avoid having the base and stem overlap, we subtract ball1.s from base1.s when we create base1.r. We also subtract the stem1.r from basin1.s when we create basin1.r. This removes material from one region that would otherwise create an overlap with another. The following figure shows the overlap between ball1.s and base1.s in blue. This is the volume that is removed from base1.r.



6. Raytracing the Goblet

To raytrace the goblet using the default material properties of gray plastic, go to the **File** menu and click on **Raytrace**. When the Raytrace Control Panel appears, change the color of the background by clicking on the button to the right of the **Background Color** box and then clicking on **white** in the drop-down menu. Next, click on **Raytrace**.

When you have finished viewing the goblet from the front view, go to the **View** option of the menu bar and select **az35, el25** and then raytrace. If you want to view the goblet without the wireframe, go to the **Framebuffer** option of the Raytrace Control Panel and click on **Overlay**. The goblet should look similar to the following illustration:



The Raytraced Goblet from an az35, el25 View

Review

In this lesson, you:

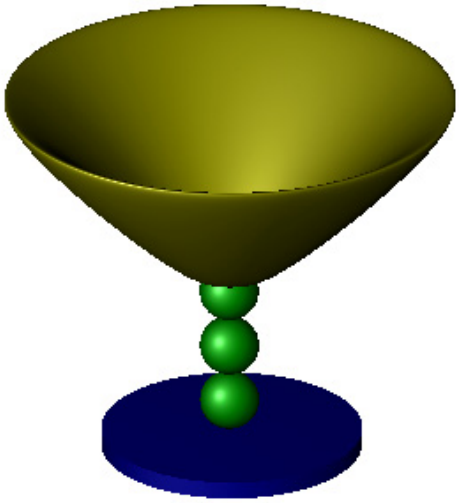
- Created a new database.
- Created, edited, and copied primitive shapes to make the parts of the goblet.
- Made regions of the parts.
- Made a combination of the regions.
- Viewed a data tree.
- Raytraced your completed goblet.

Lesson 7: Assigning Material Properties to Your Goblet

In this lesson, you will:

- Review how to open an existing database.
- Assign colors and the plastic shader to regions of the goblet.
- Use the transparency and mirror reflectance options of the shader.
- Raytrace various forms of your goblet.

In this lesson, you will add material properties to the goblet you created in the previous lesson. The finished goblet should appear similar to the one in the following example.



The Completed Goblet with Material Properties Assigned

1. Review of Opening an Existing Database

If you exited after the last lesson, open your goblet database (**goblet.g)** again. The easiest way to do this is to open the database from the Terminal Window when first starting MGED. To do this, type at the prompt:

```
mged goblet.g<ENTER>
```

Alternatively, you could start the MGED program, and select **File** from the menu bar and then **Open**. A dialog box will appear and ask you to enter an existing database name. Type in **goblet.g** (or click on it in the directory listing) and then click on **Open**. The

program should tell you that the database was successfully opened and it is using millimeters for its unit of measure. Click on **OK.**

Drawing the Goblet in the Graphics Window

To draw the goblet you made in the previous lesson, move the mouse pointer to the Command Window and type at the prompt:

```
draw goblet1.c<ENTER>
```

A wireframe representation of the goblet should appear in the Graphics Window.

2. Assigning Colors and the Plastic Shader to Regions of the Goblet

Go to the **Edit** menu and click on **Combination Editor**. To select the various regions of the goblet you made in the last lesson, go back to the **Name** box and click on the button to the right of the entry box. A submenu will appear. Double click on **Select From All Regions**. A list of regions created for this database will appear, including **base1.r**, **basin1.r**, and **stem1.r**. Double click on **base1.r** to select that region.

Click on the button to the right of **Color** in the Combination Editor, and a drop-down menu will appear with a list of available colors along with a color tool that will let you create more colors. Click on **blue**. Next, click on the button to the right of the **shader** box. A list of available shaders will appear. Click on **plastic**. A new set of options will appear. You will use two of these options in this lesson. Click on **Apply** to assign the color blue and the plastic shader to the goblet base.

Repeat this process to assign the color **green** and **plastic** shader to the **stem1.r** region and the color **yellow** and **plastic** shader to the **basin1.r** region. When you are finished, click on **OK** to dismiss the Combination Editor box.

Although the changes have been made to the database, the display in the Graphics Window doesn't reflect them yet. So, return the mouse pointer to the Command Window and type at the prompt:

```
B goblet1.c<ENTER>
```

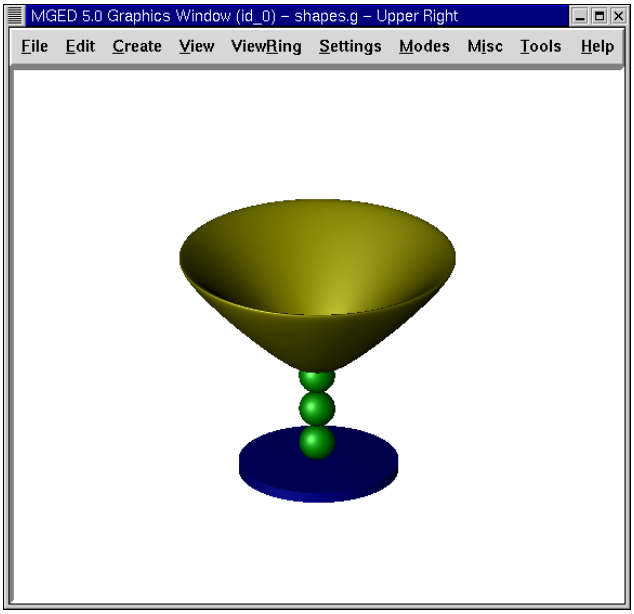
This command clears the screen and redraws the goblet with the color selections applied.

Raytracing the Goblet

To raytrace the goblet, go to the **File** menu and click on **Raytrace**. The Raytrace Control Panel will appear. Move your mouse pointer to the button to the right of **Background Color** and click on **white**. To make the raytracing go faster, you can resize the Graphics Window to make it smaller before you open the raytrace panel. When the window is resized, click on **Raytrace** to start the raytrace process.

	Note: As mentioned previously, it is undesirable to have regions that overlap. Although having overlaps may not always affect the raytracing process, if the model were going to be statistically analyzed, overlaps would create problems.

While the goblet is raytracing, move your mouse cursor to the **Framebuffer** option of the Raytrace Control Panel menu bar and click on **Overlay**. When the raytrace process is finished, you should have a goblet similar to the following example:



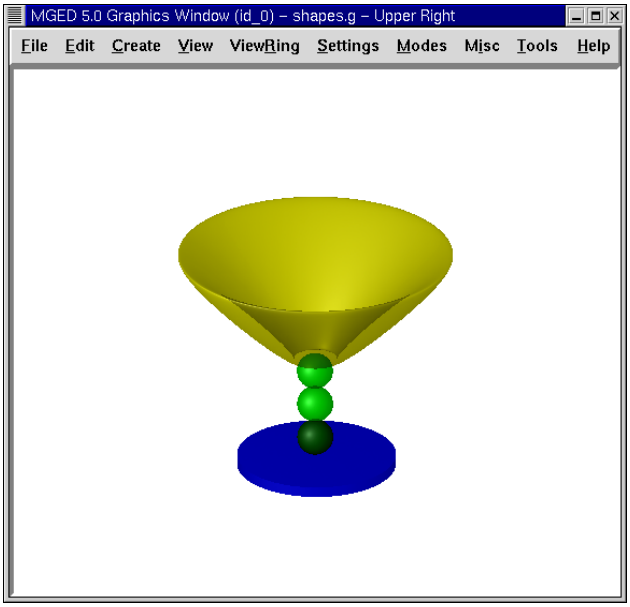
The Raytraced Goblet

3. Using the Transparency and Mirror Reflectance Shader Options

The raytraced goblet looks fairly realistic, but it could be enhanced by using other options of the Combination Editor. When you selected the plastic shader, a new set of options appeared, allowing you to choose various properties or attributes of the shader. Among the options was **Transparency**. You can adjust this property on individual regions by entering any value between 0.0 (opaque) and 1.0 (transparent).

Just as you applied color and a shader to each of the goblet's three regions, you can adjust the transparency of each region by (1) selecting the region in the Combination Editor, (2) left clicking on the box next to **Transparency**, and (3) entering any value between 0.0 and 1.0.

For this lesson, open the Combination Editor, click on the button to the right of the **Name** box, choose **Select From All Regions** in the drop-down menu, and then choose the **base1.r** region. Make sure **plastic** is the shader selected and type in **.5** to make your region semi-transparent. Click on **Apply** and repeat this process for each of the other two regions. Then **Raytrace** the goblet, which should look similar to the following:

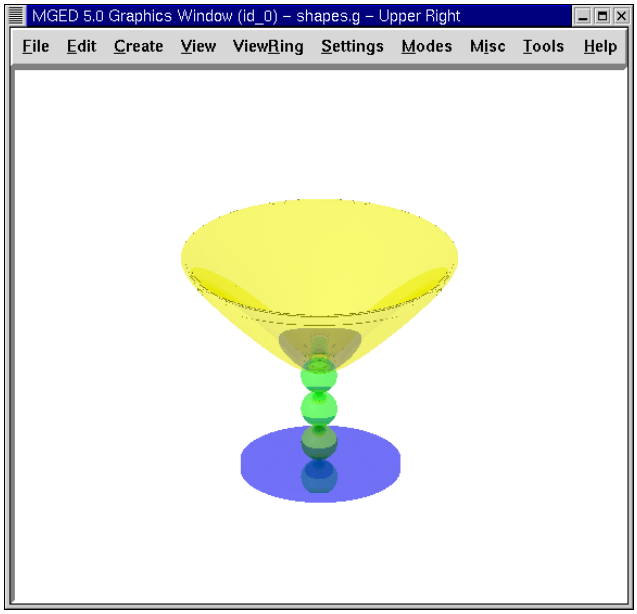


The Semi-Transparent Raytraced Goblet

The colors of the semi-transparent goblet are brighter than those of the opaque goblet because more light is allowed to penetrate the plastic material. You could make the goblet more realistic in appearance by returning to the Combination Editor and adding a **Mirror Reflectance**. For each region, place your mouse cursor in the box next to this option, click the left mouse button, and type in **.45**. This will cause about half of the available light to be reflected off the surface of the goblet.

4. Raytracing the New Forms of the Goblet

Click on **Apply** and **Raytrace** the design. The new image should appear similar to the following example:



The Raytraced Goblet with Mirror Reflectance Added

The new image is substantially different in appearance from the original image. Continue changing the values of transparency and mirror reflectance to see how they impact the resulting image.

Remember that when using these options, the combined value of both options should be less than 1.0. The following table shows you just some of the many possible combinations you could use:

Transparency Value	Mirror Reflectance Value
.50	.49
.35	.64
.20	.57
.10	.89
.89	.10

Review

In this lesson, you:

- Reviewed how to open an existing database.
- Assigned colors and the plastic shader to regions of the goblet.
- Used the transparency and mirror reflectance options of the shader.
- Raytraced various forms of your goblet.

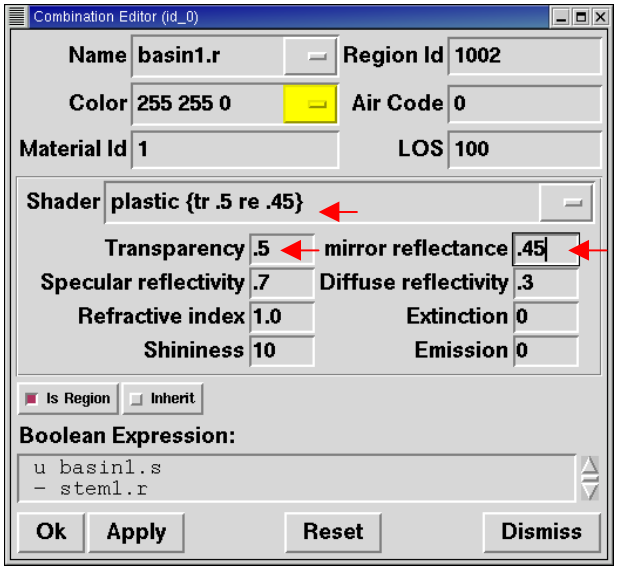
Lesson 8: Assigning More Material Properties to Your Goblet

In this lesson, you will:

- Use the specular and diffuse reflectivity options of the plastic shader.
- Assign values to the refractive index of the plastic shader.
- Assign values to the shininess option of the plastic shader.
- Assign values to the extinction option of the plastic shader.
- Experiment with various combinations of plastic shader options.

Open the **goblet.g** database using whichever method you prefer. Go to the **Edit** menu and click on **Combination Editor**. Select **basin1.r**.

In the last lesson, we assigned values for two shader attributes—transparency and mirror reflectance. In this lesson, we will assign values for still other shader properties. When the plastic shader is selected for region **basin1.r**, eight attribute entry boxes currently appear in the Combination Editor. These boxes contain either the values that the user has previously set (e.g., those we previously set for transparency and mirror reflectance) or the default values that the raytracer will use if no others are specified. When any of these values is modified, the change can be seen in braces in the shader string box and in the appropriate attribute entry boxes, as indicated by the arrows in the following example:



The Combination Editor

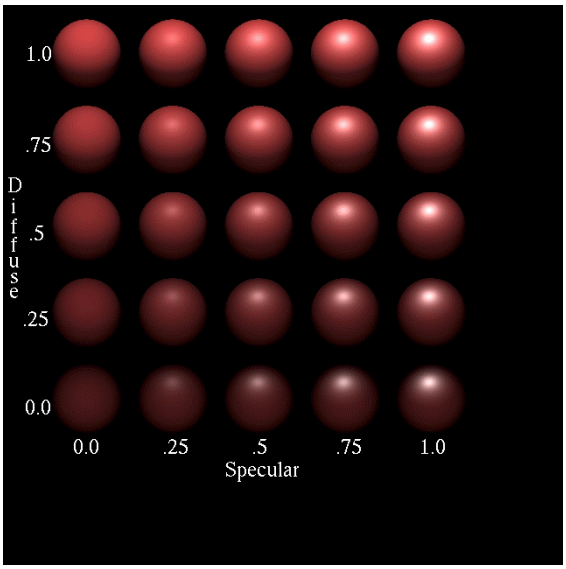
	Note that in BRL-CAD versions prior to 5.2, the default values are used, but they are not displayed in the shader attribute boxes.

In this example, the shader entry box indicates that the transparency (**tr**) is set at .5 and the mirror reflectance (**re**) is set at .45. The eight abbreviations currently used in the shader entry box are as follows:

tr - transparency	**sp** - specular reflectivity	**ri** - refractive index	**ex** - extinction
re - mirror reflectance	**di** - diffuse reflectivity	**sh** - shininess	**em** - emission

Specular and Diffuse Reflectivity

When light reflects off of a shiny surface, it produces two types of reflections. The most noticeable highlights are caused by *specular reflectivity*. The rest of the surface produces *diffuse reflectivity*. The shinier (or glossier) the surface is, such as on a crystal vase, the more specular reflectivity that is produced. The duller the surface is, such as with a wall painted with flat paint, the more diffuse reflectivity that is produced. A model of the relationship between these reflectivities is shown in the following illustration:



Specular vs. Diffuse Reflectivity Model

As seen in the illustration, diffuse reflectivity shows an object's color by reflecting ambient light off the object. The upper left ball exhibits the maximum value for diffuse reflectivity (1.0), and as a result, its surface color is uniform.

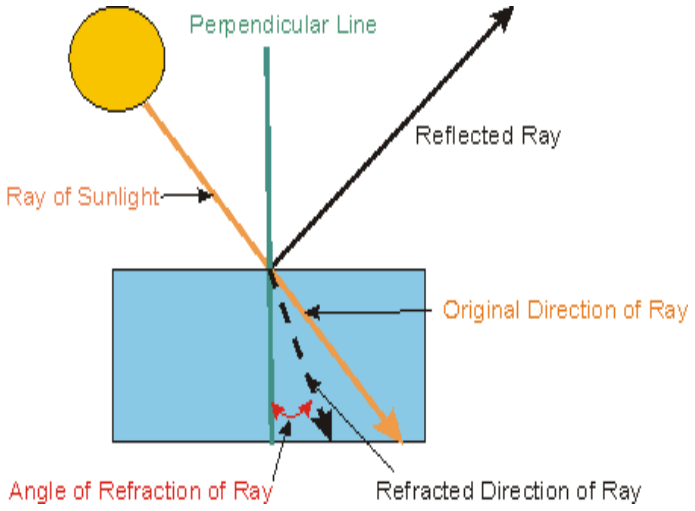
Specular reflectivity, on the other hand, reflects the color of a light source. The lowest right ball, with the maximum value for specular reflectivity (1.0), shows a white light source being reflected off the surface of the ball.

The range for both specular and diffuse reflectivity is 0.0 through 1.0. However, combined values for these are typically equal to 1.0. Remember, if you are going to set values for one of these attributes, you need to assign a corresponding value to the complementary attribute so that the combination of the values equals 1.0.

Refractive Index

When light passes through one medium (e.g., air) into another medium (e.g., water), it bends from its original path. The degree to which the light bends is called the *refractive index*. The more dissimilar the media are, the greater the degree of refraction that will occur. For example, sunlight passing through a diamond will bend more than the same sunlight through optical glass. The diamond would have a higher refractive index (approximately 2.42) whereas optical glass would have a lower refractive index (approximately 1.71).

The range of index of refraction for MGED is 1.0 (the index for air) or greater. This parameter is only useful for materials that have a transparency greater than 0. The following drawing of sunlight passing through water shows how refraction works.



A Ray of Sunlight Passing Through a Body of Water

Shininess

The *shininess* of an object affects the size of the specular component of the plastic shader. The shinier an object's surface is, the smaller the reflection of the light source on the object's surface will be. The range for shininess is typically an integer value from 1 to 10.

Extinction

The term *extinction* applies to the transmissive component of the plastic shader, and it indicates the amount of light absorbed by the object's material. The default value is 0.0, and the range can be any nonnegative number. Using this attribute can dramatically impact other attributes of the shader, especially the refractive index.

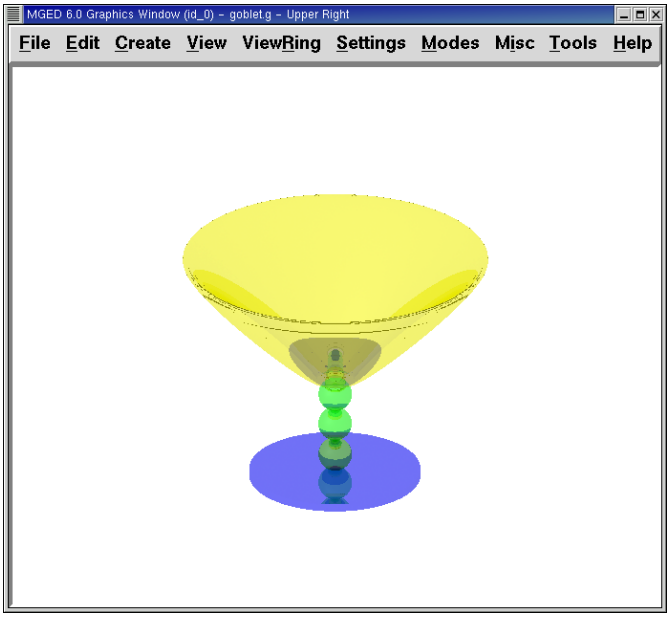
Emission

Emission is a relatively new feature that has been added to the BRL-CAD package. It concerns the amount of artificial brightness of the object.

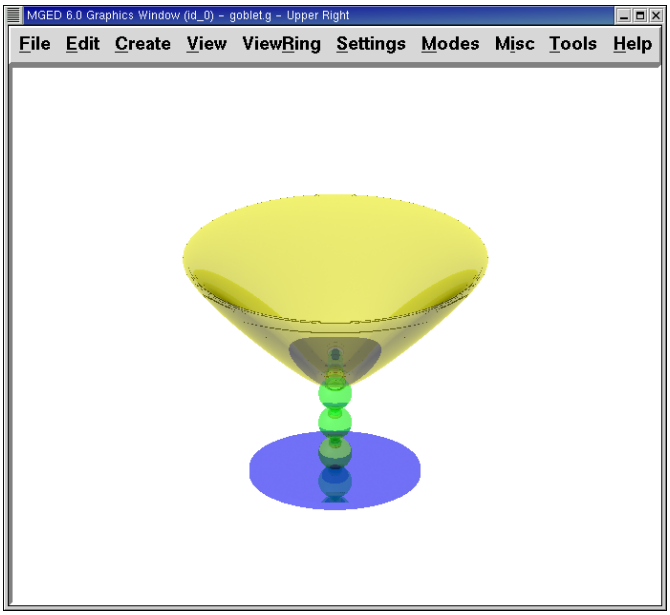
Applying Attributes of the Plastic Shader to the Goblet

Now that you understand the various attributes of the plastic shader, it is time to experiment with how they affect the final product—the goblet you created in the previous two lessons. You have already assigned values for transparency and mirror reflectance. Now add the attributes of specular reflectivity and diffuse reflectivity to **basin1.r**. Once you see how these two attributes affect your design, add the refractive index, then shininess and extinction. You might want to capture some of these changes so that you can refer to them later, when you are creating other models using the plastic shader. Remember to click **Apply** in the Combination Editor to actually incorporate the changes.

As you change the values for the attributes of the plastic shader, you will notice that some changes do not significantly alter the design. This is because there are a variety of ways to produce a particular look on an object. The following are two examples of the goblet with various values of the plastic attributes (which do create a noticeable difference) applied to **basin1.r**.

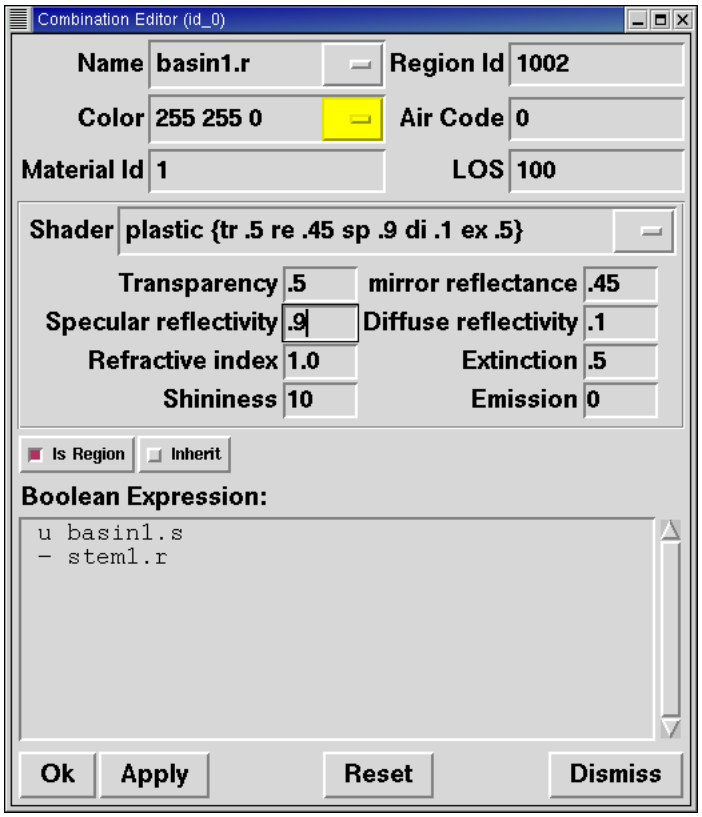


Goblet Assigned .9 for Specular Reflectivity and .1 for Diffuse Reflectivity



Same Goblet Also Assigned a Value of .5 for Extinction

By the time you have finished experimenting with changing attributes of the plastic shader, your Combination Editor window might look something like this:



The Combination Editor Window

Notice that while the shader string entry box reflects the values set by the user in the attribute entry boxes (e.g., the Transparency of .5), it does not reflect the default values (e.g., the Shininess of 10).

Review

In this lesson, you:

- Used the specular reflectivity and diffuse reflectivity options of the plastic shader.
- Assigned values to the refractive index of the plastic shader.
- Assigned values to the shininess option of the plastic shader.
- Assigned values to the extinction option of the plastic shader.
- Experimented with various combinations of plastic shader options.

Intentionally Left Blank

Lesson 9: Creating a Globe in a Display Box

In previous lessons, you combined various shapes into new objects. These new objects have been created using solid building blocks, much like those used in a wooden toy truck. However, in real life, most of the objects that you will design will consist of an outside shell and various inside parts. Therefore, in this lesson you will:

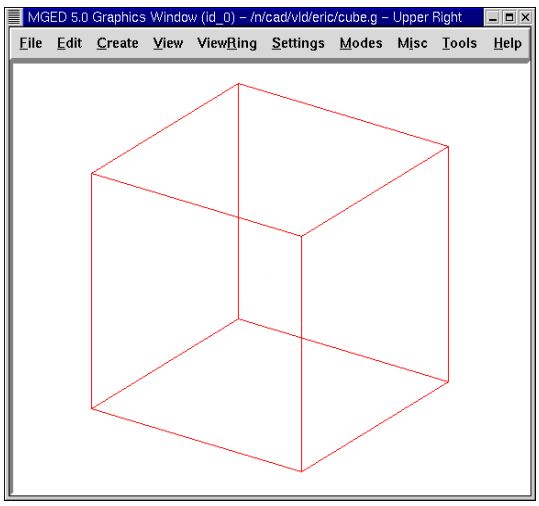
- Use the GUI to create a display box using arb8's.
- Create a globe inside the display box.
- Assign material properties to make the objects appear more realistic.
- Rotate an object 180˚ using the rotate option of the **Edit** menu.
- Use the color option of the Combination Editor to produce customized colors.

Create a New Database

Begin by creating a new database. Name your new file **cube.g**.

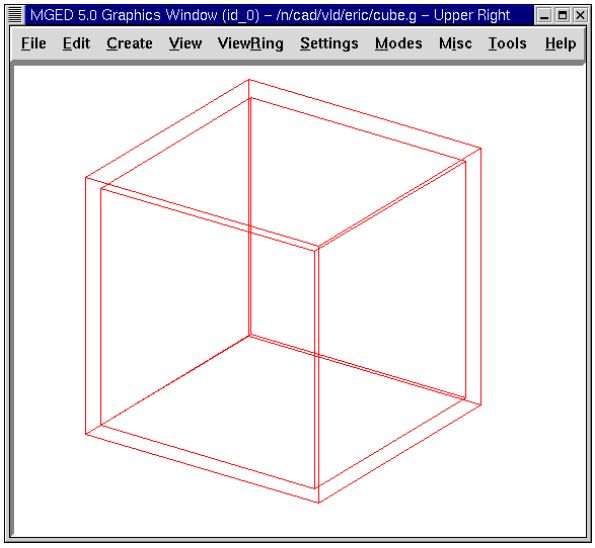
1. Creating the Display Box

Go to the **Create** menu, click on **Make Solid**, and select **arb8**. When asked to provide a name for the arb8, name it **cube1.s.** Click on **Apply**. Go to the **Edit** menu and click on **Accept**. You now have a cube for the outside of the display box, as in the following:



The Outside of the Display Box

Repeat the first part of this process to produce another arb8, this time calling this shape **cube2.s**. Go to the **Edit** menu and click on **Scale**. Place the mouse pointer in the *lower half* of the Graphics Window screen and click the *middle* mouse button until the second cube is slightly smaller than the first cube, as follows:



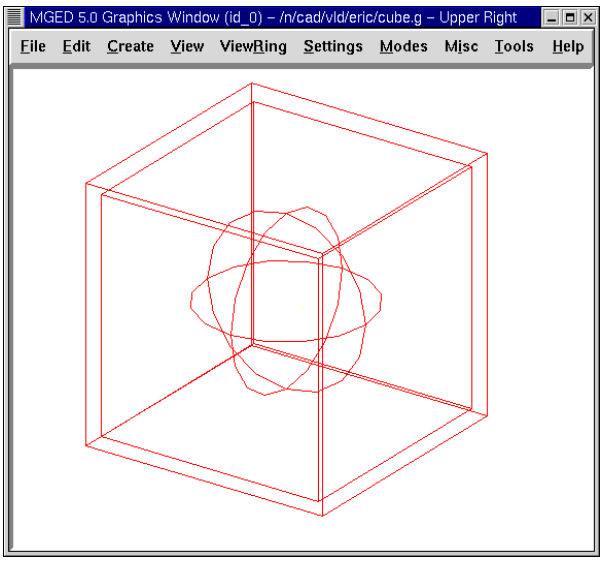
The Inside and Outside of the Display Box

Go to the **View** menu and change view to **Front**. Go to the **Edit** menu and click on **Translate** (move). Hold down the **SHIFT** key and drag the inside cube into position in the center of the outside cube. Repeat this process from the **Top** view and **Left** view until the smaller cube is placed in the center of the outside cube when viewed from all perspectives. When you are finished, go back to **Edit** and click on **Accept**.

2. Create a Globe Inside the Display Box

Go to the **Create** menu and click on **Make Solid**. Select **sph** from the list of shapes. Name the shape **globe1.s** and click on **Apply**.

A sphere should appear inside the cube in the Graphics Window. Change **View** to **Front**. Go to the **Edit** menu and select **Scale**. Reduce the size of the sphere until it will fit inside of the cube and then drag it into the center of the cube. Go to **Edit** and **Accept** your changes. Your globe and box should appear similar to the following in the az35, el25 view:



Wireframe Representation of Globe and Display Box

To view contents of the database, type at the Command Window prompt:

```
ls<ENTER>
```

You should see **cube1.s**, **cube2.s**, and **globe1.s** listed as shapes you have created. To make regions of these shapes, type at the prompt:

```
r cube1.r u cube1.s - cube2.s<ENTER>
r globe1.r u globe1.s<ENTER>
```

3. Using the Combination Editor to Assign Material Properties that Make the Objects Appear More Realistic

Go to **Edit** and select **Combination Editor**. In the dialog box, click the button next to the **Name** entry box. Select **globe1.r** from **Select From All** or **Select From All Regions**. Double click on the **globe1.r** name**.** Assign this region a **Shader** of **cloud**. Check the **Boolean Expression** box to make sure the region is made up of **u globe1.s**. Click on **Apply** to accept your choices. Go to **View** and select **az35, el25**.

Go back to **Name** and select **cube1.r** from the **Select From All** menu. Assign this region a **Shader** of **glass**. The glass shader is a shortcut to individually changing the attributes of the plastic shader to make it appear like glass.

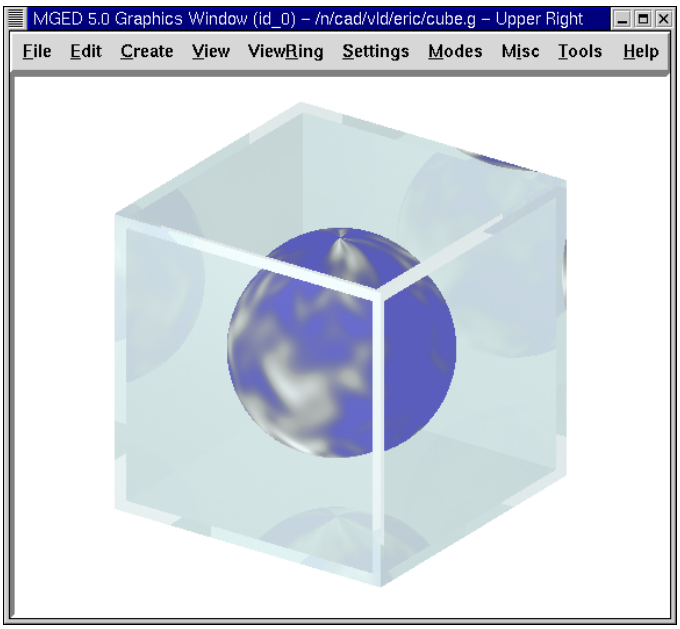
Go to the **Color** option and enter the values **244 255 255**. This will give your glass box a light cyan color. Click on **Apply** to accept your changes.

Before you can raytrace your design, you need to clear the Graphics Window by typing **Z** at the Command Window prompt because both shapes and regions are being displayed at this point. Next, type in the Command Window:

```
draw cube1.r globe1.r<ENTER>
```

The display box and globe should reappear in the Graphics Window. Go to the **File** menu and select **Raytrace**. Next, select **white** for **Background Color**. Click on **Raytrace**.

Your design should show a light cyan-colored glass cube with a blue globe inside. To eliminate the wireframing, go to **Framebuffer** (in the Raytrace Panel) and select **Overlay**. The display should appear similar to the following illustration:



Raytraced Display Box and Globe

To make this design more interesting, you can place the globe on a base. Do this by going back to **Framebuffer** and clicking on **Active** to deactivate the framebuffer. Next, go to the **Create** menu and select **Make Solid** and then **trc** (truncated right cone). Name the shape **base1.s**. Working from a **Front** view, go to the **Edit** menu and select **Scale**. Click the *middle* mouse button to reduce the trc in size until the bottom of it appears to be an appropriate size for the globe base. (You may need to increase the size of your Graphics Window or decrease your geometry view size to see the bottom of the trc.) Next, reduce the height of the shape by selecting **Set H** from the **Edit** menu and clicking with the *middle* mouse button. You may need to switch back and forth between these two options a few times to get an acceptable size. When finished, however, do not click on **Accept** yet, as we have more changes to make.

4. Moving and Rotating an Object

As with other features in MGED, moving and rotating objects can be accomplished in several ways, according to the amount of precision desired. As previously described, the Shift Grip functions can be used as a quick way to change an object when its exact angle and location do not necessarily matter. Alternatively, to achieve greater precision, the **Translate** (move) and **Rotate** commands under **Edit** can be selected and specific parameter numbers can be entered in the Command Window. In this lesson, we will experiment with both methods.

With your trc still in edit mode and still in a **Front** view, use the **SHIFT** key and the *left* mouse button to drag your shape (the bottom of the base) and sit it on the floor of the cube (just touching the *inside* box). Note that you could have selected **Translate** under **Edit** and entered parameters on the Command Line to move the trc to an exact location; however, in this case, aligning the shape with the drag-and-drop "eyeballing" method was appropriate. At this point, you may notice that your trc needs to be resized a little to better fit the globe. Use **Scale** and **Set H** as needed and then **Accept** your changes.

Now we need to make a second trc named **base2.s**, which we will use for the top of the base. On the Command Line, type:

```
cp base1.s base2.s
```

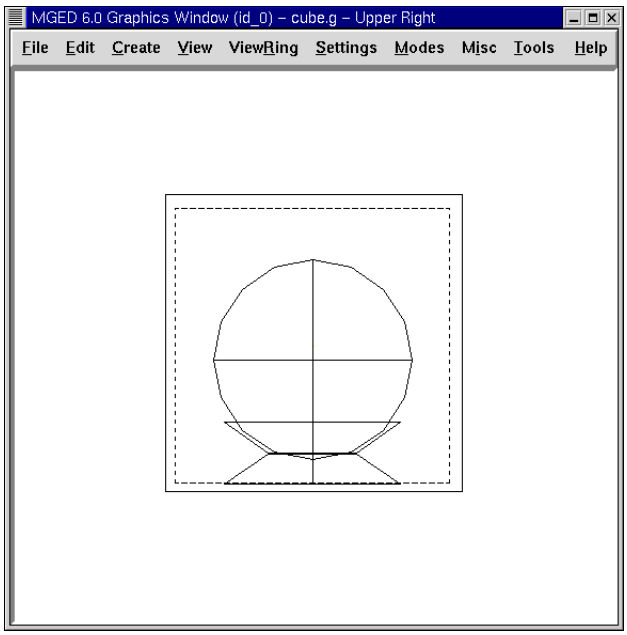
The second trc will appear directly on top of the first trc, so we will have to use **Solid Selection** under the edit menu to put **base2.s** in edit mode so that we can flip it upside down and then drag it to the top of the first trc.

To do this, we could use the **CTRL** and **ALT** (constrained rotation) keys and *left* mouse button and then move the mouse up or down until the trc is upside down. (If this method is used, note that you can release the mouse button and regrab the object if you need to.) However, because we know we want to rotate the shape an exact amount (180˚) about the *x* axis, let's use a more precise method to flip the shape. Select **Rotate** under **Edit** and then type in the following parameters (abbreviated as **p)** on the Command Line:

```
p 180 0 0<ENTER>
```

Our shape should have flipped upside down and jumped to the bottom of the first trc. (The two zeros you input indicate no rotation along the *y* and *z* axes.) Now use the **SHIFT** key and the *left* mouse button to drag **base2.s** upward and sit it on top of **base1.s**. The two shapes should form a base in which to hold your globe. Check your alignment using multiple views and then **Accept** your changes.

Go to **Edit** and **Solid Selection** and click on **globe1.r/globe1.s**. As you did with the trc's, use the Shift Grips to drag the globe down until it is in place on the base. Go back to **Edit** and click on **Accept**. Your design should look as follows:



Wireframe Representation of Globe and Base in Display Box

To make a region of the base, type in the Command Window:

```
r base1.r u base1.s u base2.s<ENTER>
```

5. Use the Color Tool of the Combination Editor to Produce Customized Colors.

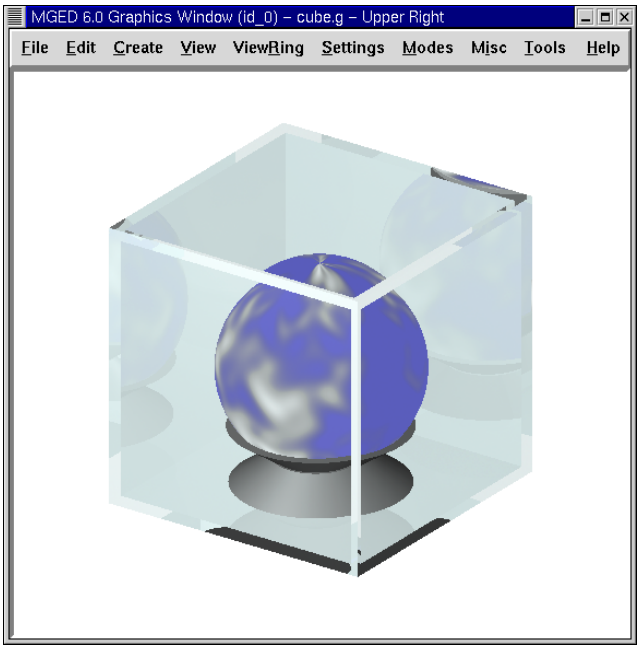
In the Combination Editor window, click the button to the right of the **Name** entry box and then **Select From All**. Choose **base1.r**. Assign the base a **Shader** of **plastic**. In the **Color** box, enter the numbers:

```
217 217 217
```

Apply your changes. Before you can raytrace your completed design, you must first clear the Graphics Window and rebuild your design by typing at the Command Window prompt:

```
Z<ENTER>
draw cube1.r globe1.r base1.r<ENTER>
```

Change your view to **az35, el25** and then raytrace your design, which should appear similar to the following:



Review

In this lesson, you:

- Used the GUI to create a display box using arb8's.
- Created a globe inside the display box.
- Used the Combination Editor to assign material properties that make the objects appear more realistic.
- Rotated an object 180˚ using the rotate option of the **Edit** menu.
- Used the color option of the Combination Editor to produce customized colors.

Intentionally Left Blank

Lesson 10: Creating a Mug

In this lesson, you will:

- Create an outside cylinder using the **in** command.
- Create an inside cylinder for hollowing out the larger shape cylinder.
- Create a handle for your mug.
- Create a combination to produce the body of your mug.
- Create a combination to join the handle to the body.
- Create a region of combined shapes with the same material and color.

In this lesson, you will be continuing your work creating real-life objects—in this case, the basic body shape of a coffee mug. In the next lesson, you will refine the body so that it is more realistic.

Creating a New Database

Create a new database and call it **mug.g**. Go back to the **File** menu and select **Preferences**, then **Units**, and then **inches**. This will create your body using inches. (Note: You could also do this by typing **units in** at the Command Window prompt.)

1. Creating the Outside Cylinder Using the In Command

To begin making the body, you will need to create an outside right circular cylinder. At the MGED prompt, type:

```
in bodyout.s rcc
```

The diagram of this command is as follows:

in	bodyout.s	rcc
Make a shape	Call it bodyout.s	The shape type is right circular cylinder

MGED will ask you the following questions about the cylinder you want to make. Type in the values given in bold. Make sure you leave spaces between variable values.

```
Enter X, Y, Z of vertex: 0 0 0<ENTER>
Enter X, Y, Z of height (H) vector: 0 0 3.5<ENTER>
Enter radius: 1.75<ENTER>
```

Note that the streamlined way to do this would be to type:

```
in bodyout.s rcc 0 0 0 0 0 3.5 1.75<ENTER>
```

The diagram of this command is as follows:

in	**bodyout.s**	**rcc**	**0 0 0**	**0 0 3.5**	**1.75**
Make a shape	Call it bodyout.s	The shape type is right circular cylinder	The *x*, *y*, and *z* of vertex is 0 0 0	The *x*, *y*, and *z* of the height vector is 0 0 3.5	The radius is 1.75

A shape of a cylinder, in wireframe form, will appear in the Graphics Window.

2. Creating the Inside Cylinder

Using this same method, type in the information for the inside right circular cylinder. This cylinder will be used to hollow out the outside cylinder. Whenever you are creating a hole in the surface of an object, make sure the shape creating the hole protrudes through the surface. This will ensure that you don't inadvertently leave a thin film of material where the two surfaces meet.

```
in bodyin.s rcc 0 0 0.25 0 0 3.5 1.5<ENTER>
```

The diagram of this command is:

in	**bodyin.s**	**rcc**	**0 0 0.25**	**0 0 3.5**	**1.5**
Make a shape	Call it bodyin.s	The shape type is right circular cylinder	The *x*, *y*, and *z* of vertex is 0, 0, and 0.25	The *x*, *y*, and *z* of the height vector is 0, 0, and 3.5	The radius is 1.5

A second cylinder, inside the first cylinder, should now appear in the Graphics Window.

3. Creating the Handle

Now you will want to enter some information about the body's handle. The shape type for the handle is an elliptical torus. At the Command Window prompt, type:

```
in handle.s eto 0 2.5 1.75 1 0 0<ENTER>
```

The diagram of this command is:

in	handle.s	eto	0 2.5 1.75	1 0 0
Make a shape	Name it handle.s	The shape type is elliptical torus	The *x*, *y*, and *z* of the vertex is 0, 2.5, and 1.75	The *x*, *y*, and *z* of the normal vector is 1, 0, and 0

The program will ask you to enter more values for the elliptical torus you are creating. Type in the values shown in bold.

```
Enter X, Y, Z, of vector C: .6 0 0<ENTER>
Enter radius of revolution, r: 1.45<ENTER>
Enter elliptical semi-minor axis, d: 0.2<ENTER>
```

A doughnut shape should appear on the right-hand side of the body. If you look carefully, you can see that about one-third of the elliptical torus intersects the body.

4. Creating the Bodyout.s-Bodyin.s Combination

The next step is to combine the two cylinders into the body of the mug. To do this, type:

```
comb body.c u bodyout.s - bodyin.s<ENTER>
```

You have told the program to make the combination **body.c** out of the union of **bodyout.s** minus **bodyin.s**.

comb	body.c	u	bodyout.s	-	bodyin.s
Make a combination	Call it body.c	Create a union of	bodyout.s cylinder	and subtract	bodyin.s cylinder

5. Creating the Handle.s - Bodyout.s Combination

To combine the handle with the outside cylinder, type:

```
comb handle.c u handle.s - bodyout.s<ENTER>
```

comb	handle.c	u	handle.s	-	bodyout.s
Make a combination	Call it handle.c	Create a union of	the handle.s torus	and subtract	the bodyout.s cylinder

6. Creating the Region Mug.r

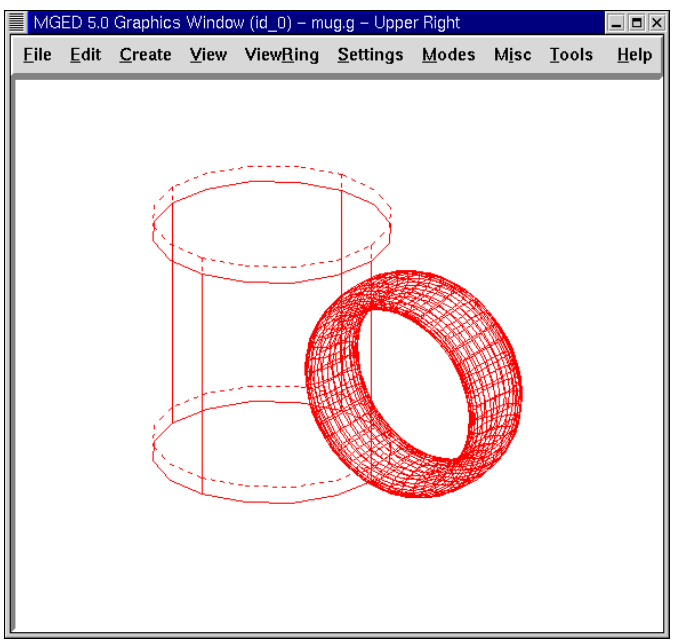
The last step, of this part of making the mug is to make a region out of your combinations. Type:

```
r mug.r u body.c u handle.c<ENTER>
```

r	mug.r	u	body.c	u	handle.c
Make a region of shapes of the same material and color	Call it mug.r	Create a union of	the body.c combination	and merge it with	the handle. c combination

If you did this correctly, the program should say something similar to:

```
Defaulting item number to 1002
Creating region id=1001, air=0, GIFTmaterial=1, los=100
```



Wireframe Mug

You should now have the region **mug.r** that is a combination of shapes containing the same material and color. You could assign color and material at this point, but you will want to do some more work on this design to make it more realistic. So, for now, review the lessons of this chapter. When you are ready to work again, you can continue refining your design in the next lesson.

Review

In this lesson you:

- Created an outside cylinder using the **in** command.
- Created an inside cylinder for hollowing out the larger shape cylinder.
- Created a handle for your mug.
- Created a combination to produce the body of your mug.
- Created a combination to join the handle to the body.
- Created a region of combined shapes with the same material and color.

Intentionally Left Blank

Lesson 11: Refining the Mug

In this lesson, you will:

- Assign material properties to the mug using the **mater** command.
- Refine the mug.
- Combine the shapes.

In this lesson, you will refine the mug you made in the previous lesson. If you stopped at the end of that lesson, open the database **mug.g** before continuing.

1. Assigning Material Properties to the Mug Using the mater **Command**

Assigning material properties to a region can be done with either the **mater** or **shader** command. The program will respond with a series of questions. These concern the various parameters of the shader you select to use in rendering the object.

The most commonly used shader is the **plastic** shader, which uses a Phong lighting model. Select the plastic shader and set the color to a medium shade of green. The dialog in the Command Window should appear as follows:

```
mged> mater mug.r<ENTER>
Shader =
Shader?  ('del' to delete, CR to skip) plastic<ENTER>
Color = (No color specified)
Color R G B (0..255)? ('del' to delete, CR to skip)
32 128 32<ENTER>
Inherit = 0:  lower nodes (towards leaves) override
Inheritance (0|1)? (CR to skip) 0<ENTER>
```

Enter the appropriate information that is shown in bold font. If you want to use the streamlined version, type:

```
mater mug.r plastic 32 128 32 0<ENTER>
```

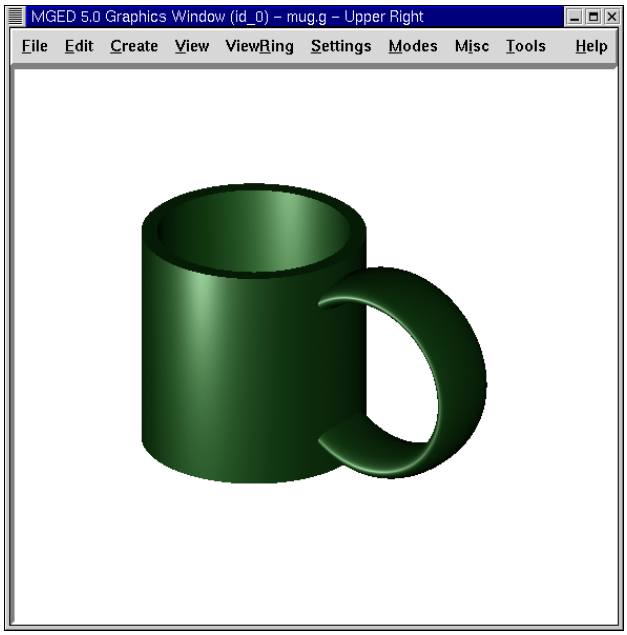
Diagrammed, this command says:

mater	mug.r	plastic	32 128 32	0
Assign material properties to a region	Called mug.r	Make the material plastic	Color the mug green	Turn inheritance off*

**Note: Inheritance is an advanced topic beyond the scope of the present discussion.*

2. Raytracing the Mug

Open the Raytrace Control Panel and select **Raytrace**. You should get an image of a green mug on a dark background (we use a white background here to save printing ink). If your mug is not green, you probably need to redraw your wireframe before raytracing.



Raytraced Mug Without Rim

3. Refining the Mug

Now let's improve the cup. Notice that the lip of the cup looks a little too squared off. To fix this, you will need to add a rounded top to the lip. You can do this by positioning a circular torus shape exactly at the top of the cup. Then you can add it to the combination **body.c**.

At the MGED prompt, type:

```
in rim.s tor 0 0 3.5 0 0 1 1.625 0.125<ENTER>
```

in	rim.s	tor	0 0 3.5	0 0 1	1.625	0.125
Make a shape	Call it rim.s	Make the shape a torus	With *x*, *y*, and *z* vertices of 0, 0, and 3.5	With *x*, *y*, and *z* of normal vector being 0, 0, and 1	Radius 1 is 1.625 inches and	Radius 2 is 0.125 inches

4. Combining the Shapes

To combine the torus with the cup, you will need to type at the prompt:

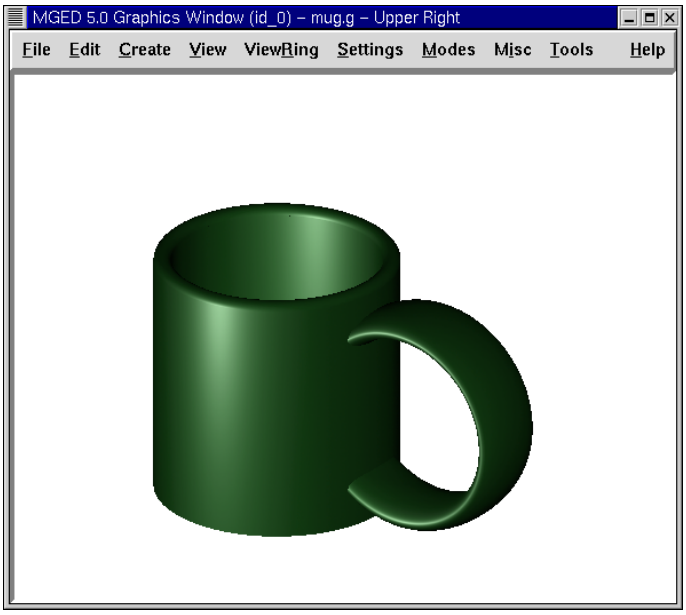
```
comb body.c u rim.s<ENTER>
```

Now you are faced with a unique situation. The shape **rim.s** was added to the list of objects being displayed when it was created. However, now it is also a part of **mug.r** (via **body.c**). If you raytrace the current view, you will have two copies of this shape. The raytracer will complain that they overlap.

One way to fix this is to clear the display, redisplay the new, complete object, and then raytrace. As discussed previously, the **fbclear** command in the Raytrace Control Panel clears the framebuffer display, and the **Z** command in the Command Window clears all wireframed objects. You can redisplay the objects you want to raytrace with the **draw** command. Type at the prompt:

```
Z<ENTER>
draw mug.r<ENTER>
```

Raytrace your mug. It should now look similar to the following.



The Mug Made Through the Command Line

Review

In this chapter, you:

- Assigned material properties to the mug using the **mater** command.
- Refined the mug by smoothing the lip.
- Combined the shapes.

Lesson 12: Creating the Mug Through the GUI

In this lesson, you will:

- Create the shapes of the mug through the GUI.
- Use new shapes to create the handle and rim of the mug.
- Combine the shapes.
- Make a region of the combinations.
- Check the data tree and make corrections.
- Assign material properties using the Combination Editor.

In the previous two lessons, you created a mug by entering commands at the Command Window prompt. Now, you will create the same type of mug using the GUI and different shapes.

Begin by opening a new database. Call it **mug2.g**.

1. Creating the Body of the Mug

Go to the **Create** menu and select **Make Solid**. Click on **rcc** (right circular cylinder). Enter the name for the rcc. Call it **outside.s**.

Go to the **Edit** menu, where you will be offered the following options:

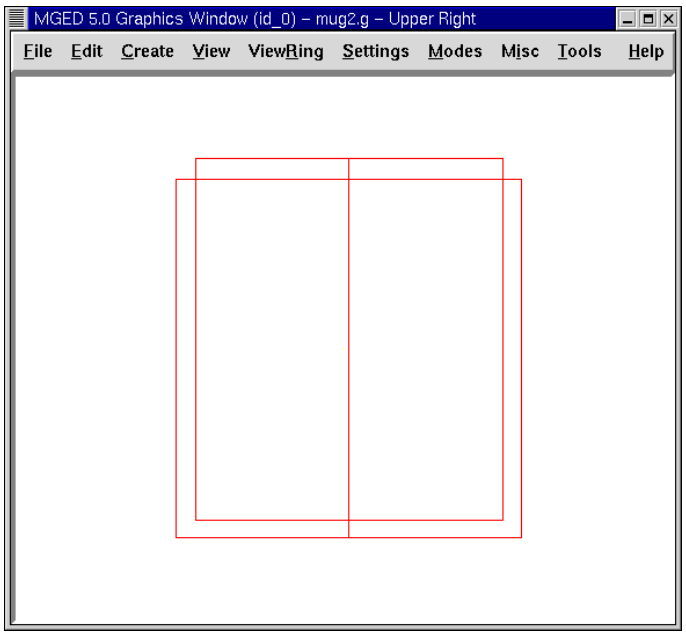
Set H **Set H (Move V)** **Set A** **Set B** **Set c** **Set d** **Set A,B** **Set C,D** **Set A,B,C,D** **Rotate H** **Rotate AxB** **Move End H(rt)** **Move End H**	**Rotate** **Translate** **Scale** **None of the Above**	**Reject** **Accept** **Apply** **Reset**	**Solid Editor** **Combination Editor**

Note: The first column includes shape-specific functions. The other columns are functions common to all shapes.

Select **Set H**. From a **Front** view, move your mouse cursor to the *lower* half of the screen and click the *middle* mouse button to reduce the scale of the cylinder's height. Next select **Set A,B,C,D**. Move your mouse cursor to the *upper* half of the screen and click the *middle* mouse button to increase the diameter of the cylinder. **Accept** your changes when your object appears similar to the one shown in the following figure.

	Note: If at any time when you are editing through the GUI you don't like your changes, you can click on **Reject** to refuse the changes or **Reset** to return the shape to its original form. However, if you select **Reject**, you will have to re-enter the Solid Edit state, as described in previous lessons.

Next, create an inside right circular cylinder and name it **inside.s**. Edit the cylinder the same way you edited the outside cylinder. Before you accept your changes, change **View** to **Top** and make sure your cylinders are in alignment. If the cylinders are out of alignment, use the **SHIFT** key and *left* mouse button to drag the inside cylinder into position. Return your **View** to **Front** and **Accept** your changes when the cylinders are lined up. Your cylinders should look like those in the following example:



Two Cylinders Shown from a Front View

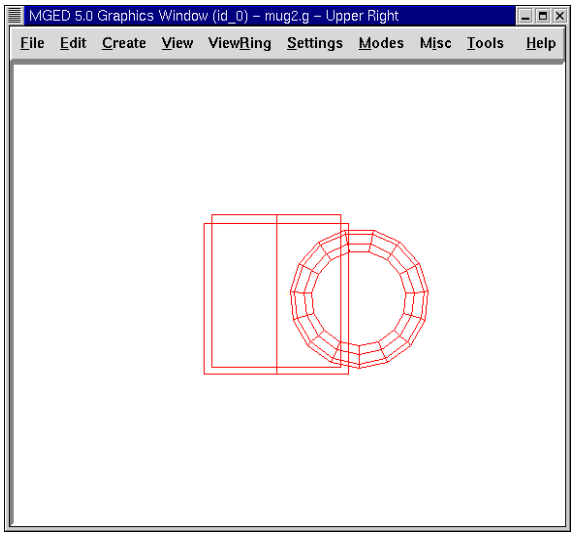
	Note: Remember that when you scale a shape, the position of the mouse pointer in the Graphics Window will determine how large or small the change will be. The closer the mouse pointer is to the center horizontal line of the window, the smaller the change will be, and vice versa.

2. Creating the Handle of the Mug

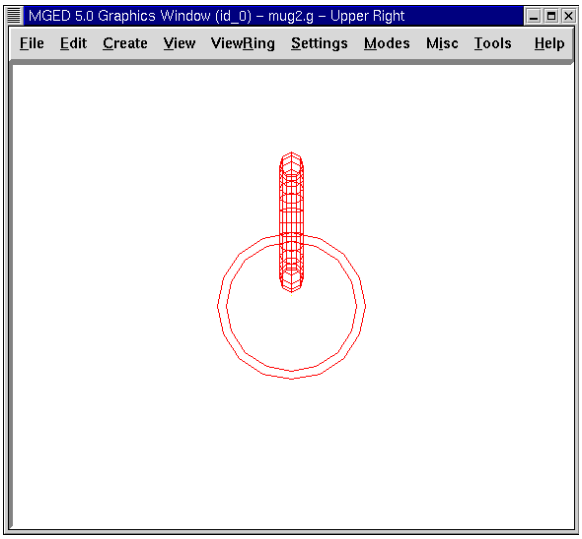
Previously, we made the handle of the mug using an elliptical torus. In this lesson, we make the handle by selecting a torus (which is a doughnut shape) from the menu of shapes. Name the torus **handle.s**. The **Edit** menu will now offer a different set of parameters than those of the right circular cylinders, as shown in the following list:

Set Radius 1 **Set Radius 2**	**Rotate** **Translate** **Scale** **None of the Above**	**Reject** **Accept** **Apply** **Reset**	**Solid Editor** **Combination Editor**

In this instance, **Set Radius 1** changes the distance from the center of the doughnut hole to the middle of the dough. **Set Radius 2** changes the radius of the dough ring. With the same technique used in editing the rcc's, edit the size of the torus until it looks similar to the following examples:



Mug and Handle from a Front View



Mug and Handle from a Top View

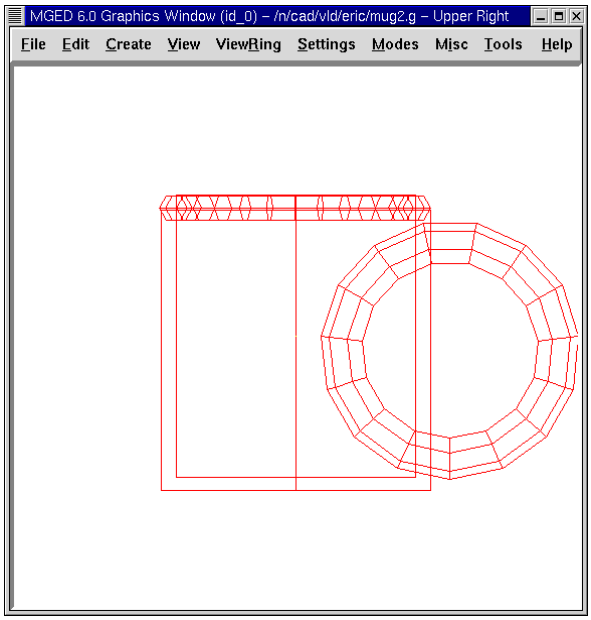
Check your mug from the top to make sure the handle is aligned. Accept your changes when you are finished.

3. Creating the Rim of the Mug

To make the rim of the mug, go to the **Create** and then **Make Solid** menus, select **tor**, and name it **rim.s**. Select the **Rotate** command and type on the Command Line:

```
p 0 90 0<ENTER>
```

to rotate the torus on its side (90˚ about the *y* axis). Then, **Scale** and edit the various parameters of the torus using the front and top views until the mug looks similar to the following example. Make sure you **Accept** your changes when you are finished.



Mug with Rim Shown from a Front View

4. Creating Combinations of the Various Shapes

To combine the various shapes of the mug, type the following commands at the Command Window prompt:

```
comb mug.c u outside.s - inside.s<ENTER>
comb handle.c u handle.s - outside.s<ENTER>
comb mug.c u rim.s<ENTER>
```

Note: Refer to the previous two lessons to recall how each of these commands works.

5. Making a Region of the Combinations

To make a region out of the combinations you just created, type at the Command Window prompt:

`r mug.r u mug.c u handle.c<ENTER>`

6. Checking the Data Tree

Before continuing, it would be wise to check your data tree and make sure it agrees with the following tree:

```
mug.r/R
   u mug.c/
      u outside.s
      - inside.s
      u rim.s
   u handle.c/
      u handle.s
      - outside.s
```

If your data tree doesn't look like this example, you will need to go back and figure out where you went wrong. If necessary, you can kill off a shape, combination, or region by typing at the Command Window prompt:

`kill [`*`name of shape, combination, or region`*`]<ENTER>`

For example, in this lesson you may have created an extra shape, named **rim2.s**, which you no longer want. To kill this shape, you would type:

`kill rim2.s<ENTER>`

7. Assigning Material Properties Using the Combination Editor

Go to the **Edit** menu and select **Combination Editor**. Type **mug.r** in the **Name** entry box. Press **ENTER**. Type **0 148 0** in the **Color** entry box. Select a **plastic** shader. Check the **Boolean Expression** box to make sure it says:

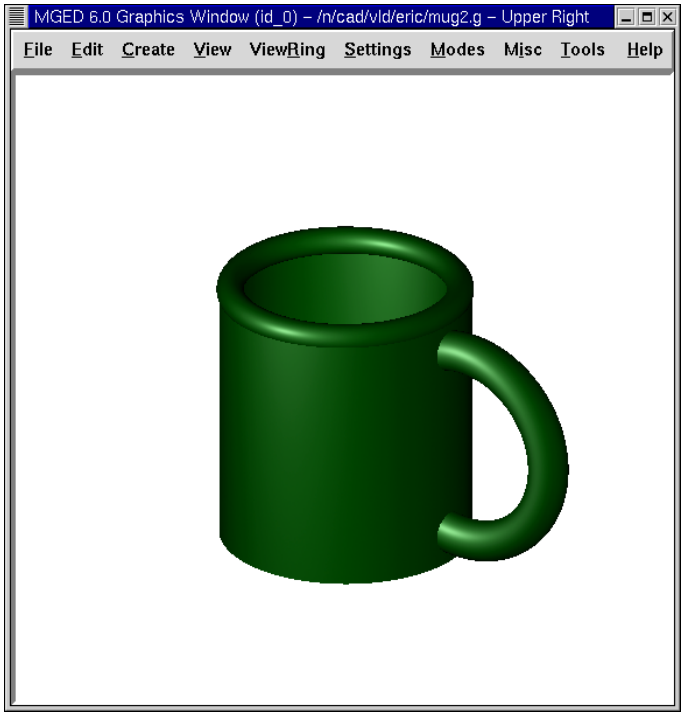
```
u mug.c
u handle.c
```

When you are finished, click on **Apply** and then **Dismiss**. In the Command Window then, type at the prompt:

`B mug.r<ENTER>`

8. Raytracing the Design

Go to the **View** option of the menu bar and select **az35, el25**. Go to **File** and then **Raytrace**. Select a **white** background color and **Raytrace** your design. Click on **Overlay**. When the raytracing is finished, it should look like the following example:



The Completed Raytraced Mug

Review

In this lesson, you:

- Created the shapes of the mug through the GUI.
- Used new shapes to create the handle and rim of the mug.
- Combined the shapes.
- Made a region of the combinations.
- Checked the data tree and made corrections.
- Assigned material properties using the Combination Editor.

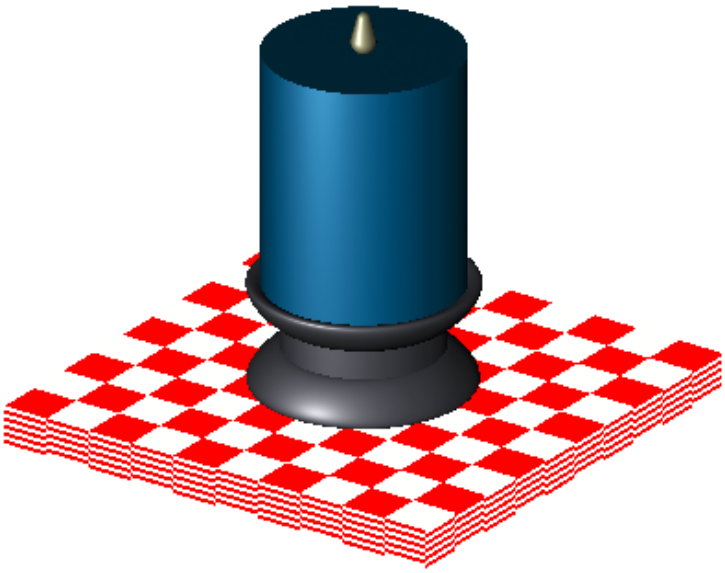
Lesson 13: Placing Shapes in 3-D Space

In this lesson, you will:

- Create, edit, and place shapes in 3-D space.
- Create custom colors using the Combination Editor.
- Identify the attributes of the checker shader.
- Identify how RGB colors are created.

In previous lessons, you created and edited shapes. You also placed objects in three-dimensional space. This lesson will provide more advanced practice on creating and editing shapes and placing them in 3-D space.

The design you will make in this lesson is a simple candle in a candle holder sitting on a table (as shown in the following figure). In the next lesson, you will add decorations and lighting to make the design more realistic.



The Candle Design

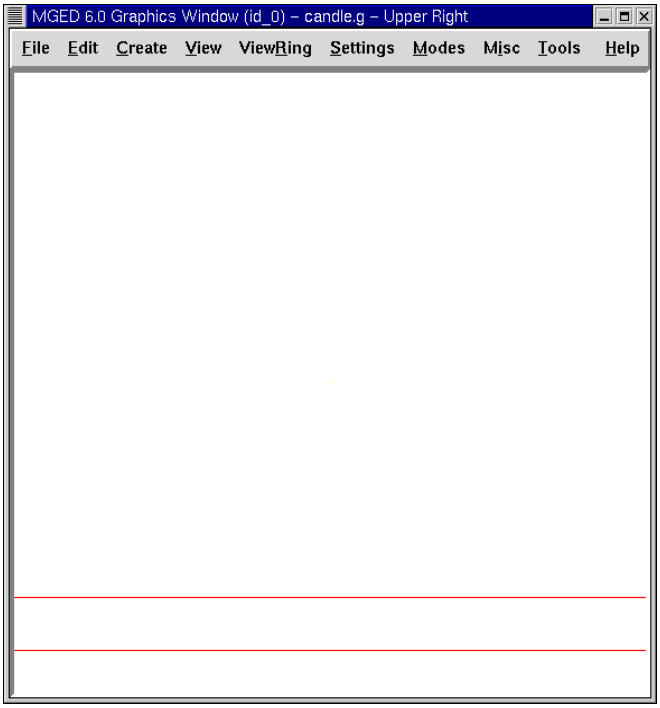
Begin by creating a new database called **candle.g**. Title your database **Candle Tutorial**.

1. Creating the Tabletop

Create an arbitrary convex polyhedron (**arb8**) from the GUI. Name the shape **arb8.s**. Go to **View** and select **Front**.

Go to the **Edit** option of the menu bar. The arb8 needs to be made larger, so, under the **Edit** menu, select **Scale**. Put the mouse pointer in the *upper* half of the screen to make the arb8 larger and click the *middle* mouse button until the sides of the arb8 touch each side of the screen. Use the **SHIFT** key and *left* mouse button to drag the arb into position, if necessary.

Under the **Edit** menu, select **Move Faces** and then **Move Face 4378**. Place the mouse pointer in the *lower* half of the screen and click the *middle* mouse button until the arb8 is about the thickness of a tabletop. Go back to **Edit** and **Accept** the changes, and then use the SHIFT and *any* mouse key to position the tabletop so that it appears similar to the following:



Wireframe Representation of Tabletop from Front View

Make a region of the tabletop by typing at the Command Window prompt:

```
r table1.r u arb8.s<ENTER>
```

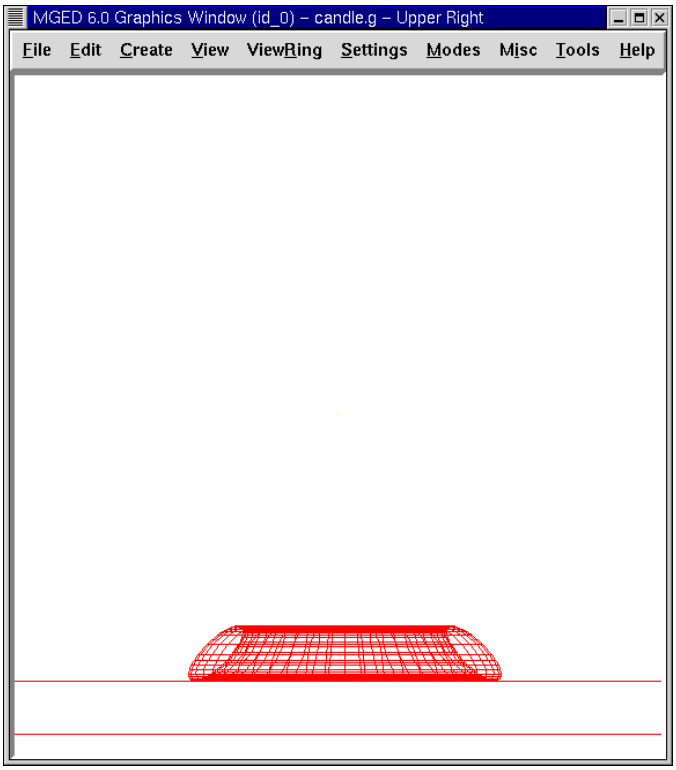
2. Creating the Candle Base

Create an elliptical torus (eto) and name it **eto1.s**. To create the bottom of the candle base, you will need to flip the eto 180˚. Type at the Command Window prompt:

```
rot 0 180 0<ENTER>
```

This tells MGED to rotate the shape 180˚ along the *y* axis. Next, select **Scale** and make the eto a little smaller than its default size. Place the eto on the tabletop by using the **SHIFT** key and *left* mouse button to drag the base into position.

View your design from different angles to make sure the eto sits flush on the center of the tabletop. Click on **Accept** when you are satisfied with its size and placement. Your base should be similar to the one shown as follows:



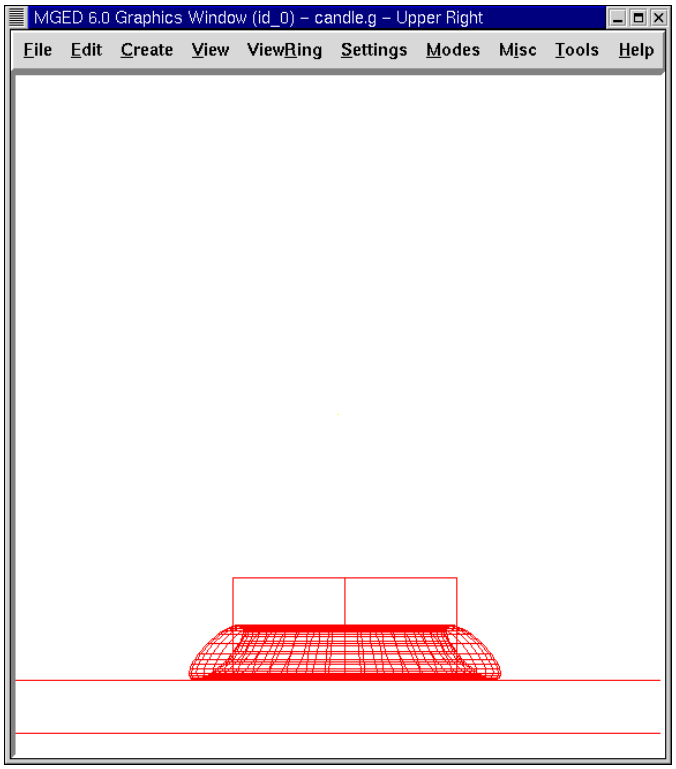
Wireframe Representation of Tabletop and First eto

The next step in creating a candle base is to make a right circular cylinder (rcc). Name the shape **rcc1.s**.

Go to **Edit**. In addition to the standard commands, you will be presented with a menu of thirteen shape-specific ways to edit this shape.

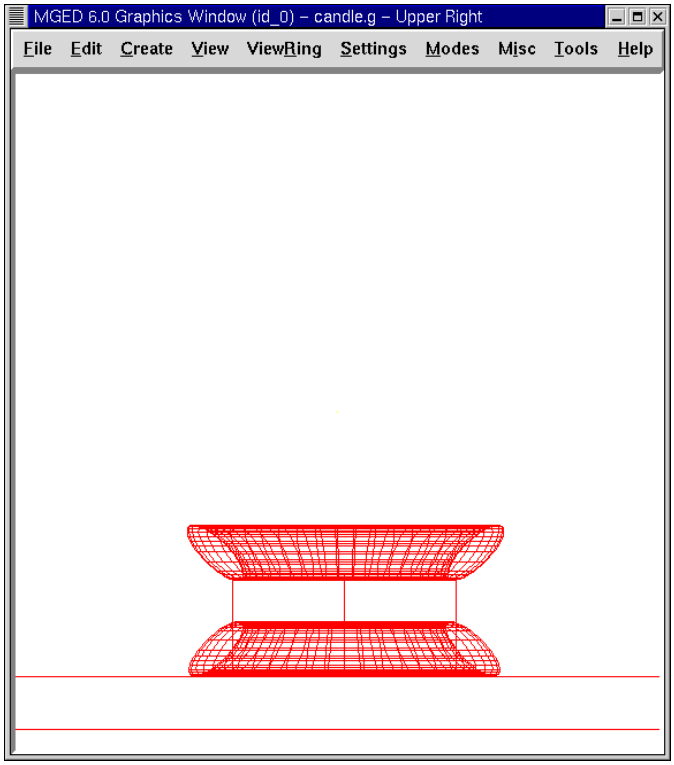
Set H **Set H (Move V)** **Set A** **Set B** **Set c** **Set d**	**Set A,B** **Set C,D** **Set A,B,C,D** **Rotate H** **Rotate AxB** **Move End H(rt)** **Move End H**

Scale the shape until it is slightly larger in diameter than the top of the **eto1.s** (you can check this by switching to a top view). Go back to **Edit** and select **Set H**. Reduce the height of the shape until the rcc is about two times the height of **eto1.s**. Position the cylinder on the candle-holder base. Check the placement of the rcc from the top, left, and front to ensure that it is centered in the eto. Make sure the bottom of the rcc is not quite touching the tabletop. **Accept** your changes. When done, your design should look like the following:



Wireframe Representation of Tabletop, First eto and First rcc

The last step in making the candle base is to create another eto. Name it **eto2.s**. Edit this shape as you did the previous eto and place it on top of the rcc, as shown in the following figure. **Accept** your changes when finished. Your candle base should now look like this:



Wireframe Representation of Tabletop and Candle Base

Make a region of the three shapes of the base. Name it **base1.r**. Your Boolean expression should read:

```
r base1.r u eto1.s u rcc1.s u eto2.s
```

Note that we could have written it

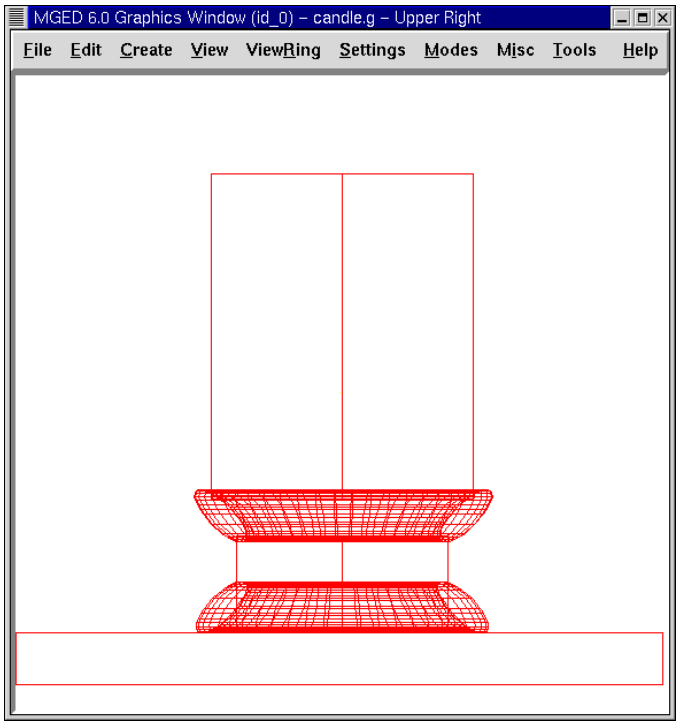
```
r base1.r u eto1.s u eto2.s u rcc1.s
```

but the first expression is preferred to be consistent with the order of a later example. In a moment, we will want to remove some material that **eto2.s** gives us. By placing **eto2.s** last in the list, we can perform this removal easily.

3. Creating the Candle

Create an rcc and name it **rcc2.s**. Edit the shape as you did the previous rcc. When you are done, it should look similar to the one in the following illustration. (Note: After you have accepted the changes, you can get all of your tabletop and candle in the Graphics

Window by using the **SHIFT** key and *left* mouse button to move your view of the design.)



Wireframe Representation of Tabletop, Candle Base, and Candle

Make a region of the candle. Your Boolean statement should read:

```
r candle1.r u rcc2.s
```

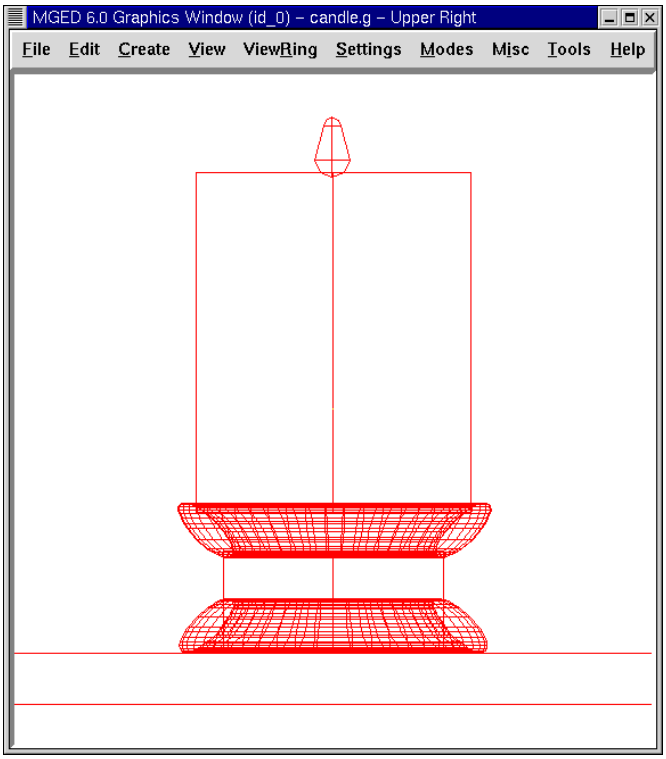
Now we create a cylindrical cutout in the base for the candle to sit in. To do this we can use the shape of the candle, as follows:

```
r base1.r - rcc2.s
```

Earlier we had mentioned that we would want to remove some material that we got from **eto2.s**. Now we have done it.

4. Creating the Candle Flame

Create a particle (**part**) and name it **part1.s**. Edit and position the shape until your design looks like the following one:



Wireframe Representation of Completed Candle Design

Make a region of the flame by typing at the Command Window prompt:

```
r flame1.r u part1.s<ENTER>
```

5. Making a Combination of the Base, Candle, and Flame

To make a combination of the parts of the candle, type at the Command Window prompt:

```
comb candle1.c u base1.r u candle1.r u flame1.r<ENTER>
```

6. Checking the Data Tree

Now that you have made a number of regions and a combination, it would be a good time to check your data tree and make sure it agrees with the following tree. If you find that you have made a mistake in any of the parts of the tree, you can change them in the **Boolean Expression** box of the Combination Editor (refer to Lesson 5). At the Command Line prompt, type:

```
tree candle1.c<ENTER>
```

Your Boolean expression should read:

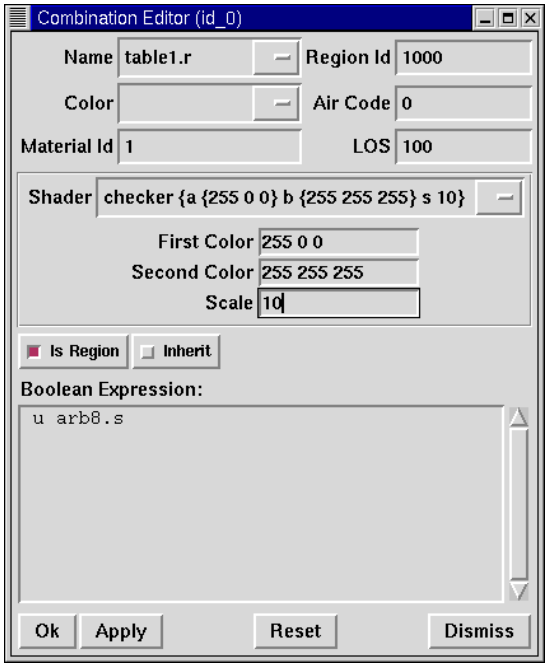
```
candle1.c/
   u base1.r/R
      u eto1.s
      u rcc1.s
           u eto2.s
      - rcc2.s
   u candle1.r/R
      u rcc2.s
   u flame1.r/R
      u part1.s
```

7. Assigning Material Properties to the Elements of the Design

To assign material properties to your design, go to the **Edit** menu and select the **Combination Editor**. Assign the following material properties to each of the elements:

Element	Shader	Color(s)	Other
Tabletop	Checker	Red (255 0 0); White (255 255 255)	Scale (10)
Candle Base	Plastic	Medium Gray (128 130 144)	
Candle	Plastic	Light Blue (0 166 255)	
Flame	Plastic	Light Yellow (255 255 190)	

Notice that the **checker** shader for the tabletop includes two color values and a scale value. Type the values for red, white, and a scale of 10 in the boxes, as follows:



Combination Editor with the Checker Shader Selected

For the rest of the elements of the design, use the **Color Tool** to make the colors shown or simply type them in the **Color** text box, remembering to leave a space between each set of numbers.

	As discussed previously, a color is made up of three numbers, ranging from 0 to 255. The first number represents the amount of red, the second represents the amount of green, and the third represents the amount of blue used to make the color. A color of 0 0 0 is black, and 255 255 255 is white. This method of creating colors is different from mixing pigment colors used in painting because you are dealing with light. While it may seem strange at first, most MGED users quickly become adept at creating RGB colors.

8. Raytracing Your Design

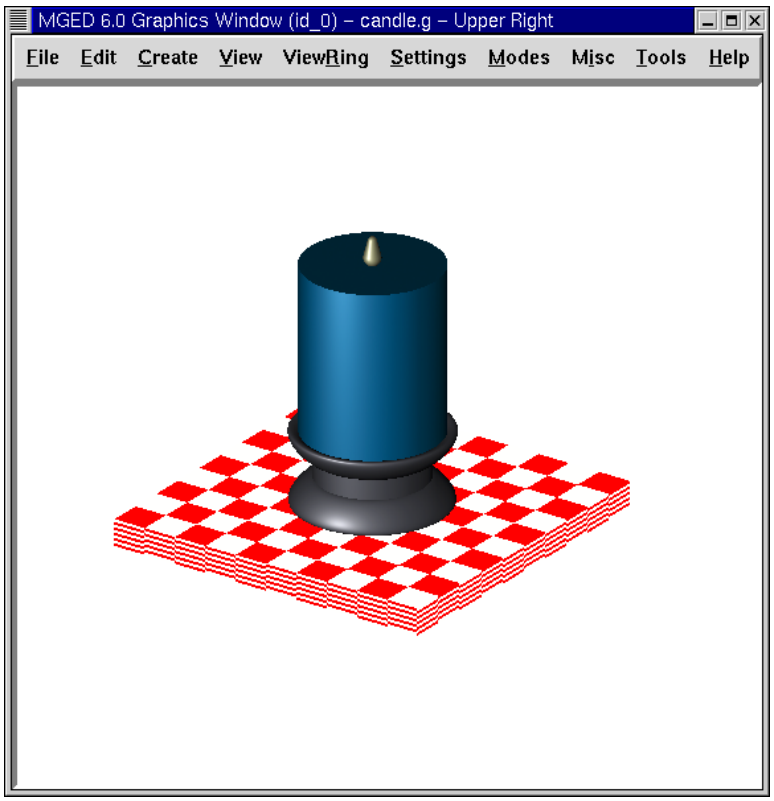
Before raytracing, change the **View** to **az35, el25** to give a better view of the completed design and then Blast the old design by typing at the Command Window prompt:

```
B table1.r candle1.c<ENTER>
```

This command tells the MGED program to:

B	table1.r	candle1.c
Clear the Graphics Window	Draw the region named table1.r	Draw the combination named candle1.c

To provide the most light on your design, use a white background color. Your raytraced candle should look similar to the following:



Raytraced Candle Design in Overlay Mode

Review

In this lesson, you:

- Created, edited, and placed shapes in 3-D space.
- Created custom colors using the Combination Editor.
- Identified the attributes of the checker shader.
- Identified how RGB colors are created.

Lesson 14: Gaining More Practice Placing Shapes in Space

In this lesson, you will:

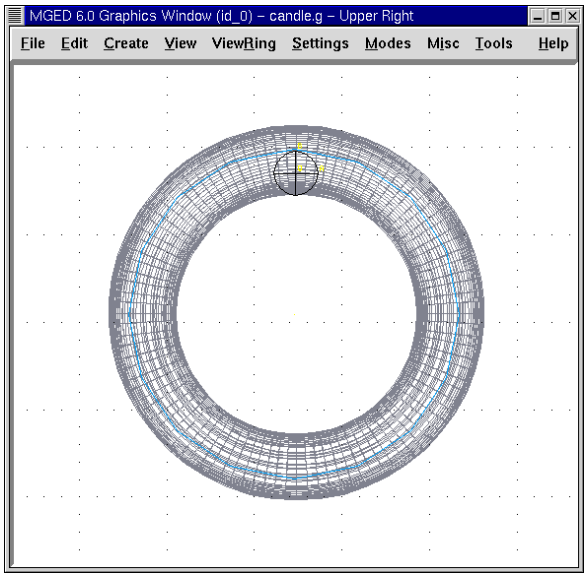
- Create copies of a shape using the Solid Editor.
- Draw a grid to help position objects.
- Check the data tree and make corrections (if needed).
- Assign material properties using the Combination Editor.

In previous lessons, we created and edited shapes and placed objects in 3-D space. This lesson provides more advanced practice in these areas using the candle design you created in the last lesson.

Open the **candle.g** database if it isn't already open and draw **candle1.c**.

1. Making the First Sphere

Using the GUI, create a sphere named **sph1.s**. Go to the **View** menu and select **Top** view. Go to the **Edit** menu, select **Scale**, and size the sphere until it is proportionally about the same size as the one in the following illustration:



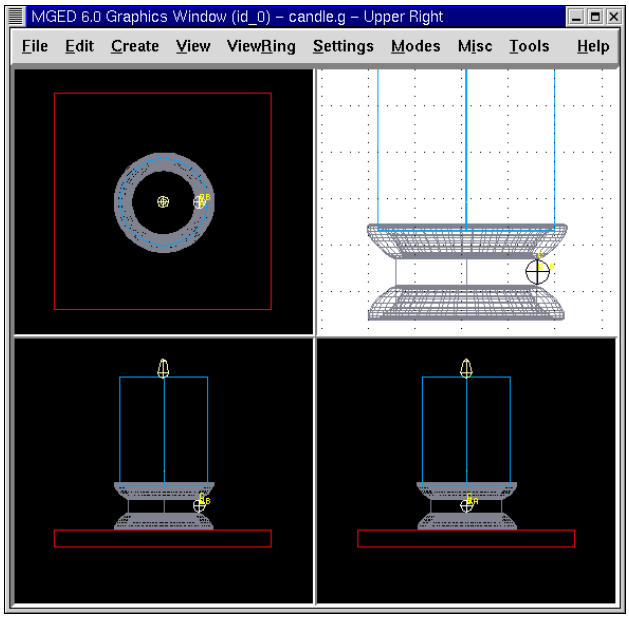
Sphere Placed on Candle Base from Top View

2. Using the Draw Grid Feature

Drag the sphere into position on the rcc, as shown in the previous illustration. To make this task a little easier, you can go to the **Modes** menu and click on **Draw Grid**. This will create a grid overlay in the Graphics Window, which can help you position your spheres on the candle base.

3. Using the Multipane Feature

As discussed earlier, another feature that is available to help you position each sphere is the **Multipane** option under the **Modes** menu. This will allow you to see multiple views of the design you are creating.



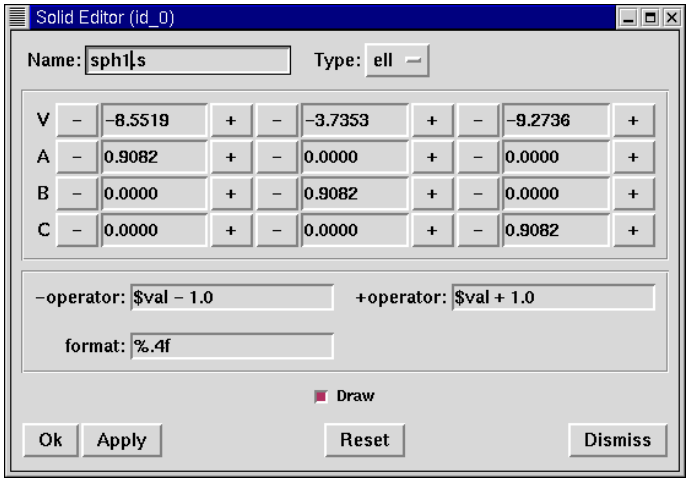
Multipane Feature

As you move a shape, the change in position will be reflected in each pane. The multipanes help you visualize where the shape is in 3-D space. In the default mode, the top left pane shows the top view, the top right pane shows the current view, the bottom left pane shows the front view, and the bottom right pane shows the left view. To turn off either the grid or the multipane functions, go back to **Modes** and click on the feature you want to disable.

4. Creating Copies of a Shape

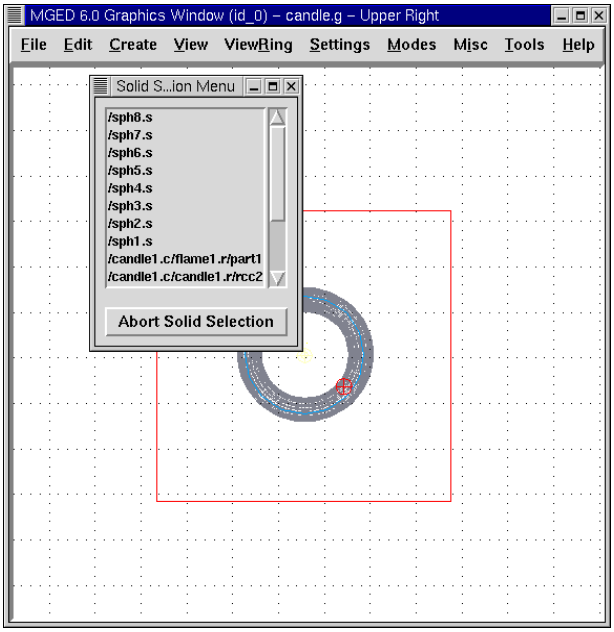
To make more jewels for the base, you could use the copy command on the Command Line **(cp sph1.s sph2.s),** but another way to do this is to go to the **Edit** menu and select **Solid Editor**. Type **sph1.s** in the text box to the right of **Name**. Click on **Reset** and then

change the name to **sph2.s** and click **Apply**. Continue doing this until you've made eight jewels. Because each of the new spheres is an exact copy of the first sphere, you won't be able to see them until you select and then move them.



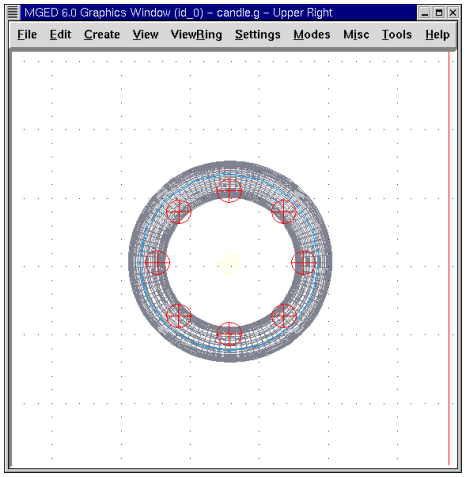
The Solid Editor Dialog Box

To position your new spheres, go to **Solid Selection**. A submenu of shapes you have created will drop down. Use the scrollbar to the right of the list of shapes to access the spheres you have created, as shown in the following illustration.

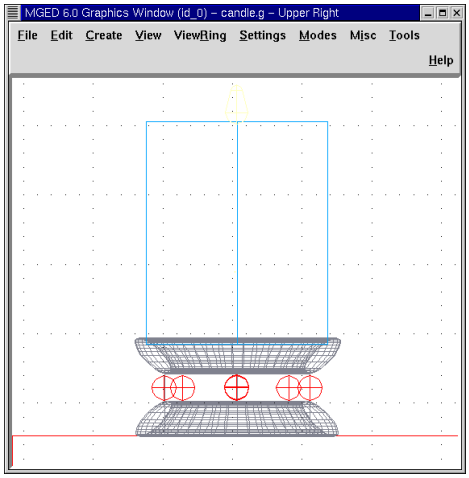


Drop-Down Menu of Solids Available Through Solid Selection

Click on **sph2.s** and drag it into position. Once you have positioned the eight spheres around the rcc, your design should look similar to the following ones when viewed from the top and front.

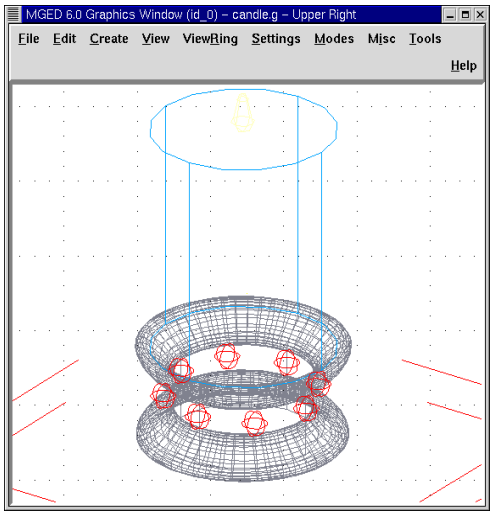


Candle from Top View



Candle from Front View

Notice from the front view that there appears to only be five spheres around the base of the candle, but there are eight spheres when you view the design from the top. That is because you are viewing 3-D space on a 2-D screen and the spheres in the back are behind the ones in the front. If you change the view to an az35, el25 view, all of the spheres will appear, as shown in the following figure. This is one reason why it is important to continually check your design from multiple views. A mistake in placement that doesn't appear from one view may be very noticeable from another view.



Candle Viewed from az35, el25

5. Making Regions of the Spheres

Now that all your spheres are made and in place, it is time to make a region of each sphere. To do this, type the following in the Command Window:

```
r sph1.r u sph1.s<ENTER>
r sph2.r u sph2.s<ENTER>
r sph3.r u sph3.s<ENTER>
r sph4.r u sph4.s<ENTER>
r sph5.r u sph5.s<ENTER>
r sph6.r u sph6.s<ENTER>
r sph7.r u sph7.s<ENTER>
r sph8.r u sph8.s<ENTER>
```

There are three easier ways to make all of the regions. The first involves typing the first command:

```
r sph1.r u sph1.s<ENTER>
```

and then using the up arrow to recall this command. Now use the left arrow to move backward in the Command Line to replace both occurrences of the number "1" with "2" and press **ENTER**. Repeat this for each of the numbers 3 through 8.

The second approach is based upon the fact that the Command Line interpreter of MGED uses the Tcl/Tk language. This gives us access to some convenient loop commands. The following will make all of the regions for us in a single command:

```
foreach i { 1 2 3 4 5 6 7 8 } { <ENTER>
     r sph$i.r u sph$i.s <ENTER>
}<ENTER>
```

This is much easier and faster than either of the previous methods. However, if there were many more spheres (say 1000 or more), then it might be easier to use a third approach, which employs a different loop type:

```
for {set i 1} {i <= 1000} {incr i} {<ENTER>
     r sph$i.r u sph$i.s <ENTER>
}
```

Next, go to **Edit** and then **Combination Editor**. Select **sph1.r** from the **Select From All** choice in the pull-down menu to the right of the **Name** entry box. Assign properties of **plastic** and the color **red** and then press **Apply**. We then can go back to the **Select From**

All menu listing and repeat this process for the other seven spheres. Alternatively, we could use **Apply** after selecting the appropriate material properties and then type in the next sphere's name; however, this method requires the user to remember to update the **Boolean Expression** box (e.g., change **u sph1.s** to **u sph2.s**) so that the Booleans for one shape are not applied to another shape.

Once again, we are performing the same operation multiple times. This is another good opportunity to use a loop.

```
foreach i { 1 2 3 4 5 6 7 8 } {<ENTER>
     mater sph$i.r "plastic" 255 0 0 0<ENTER>
}<ENTER>
```

In general, the graphical interface is good for doing one thing at a time or doing highly visual operations. Repetitive operations are best performed using a Command Line interface.

6. Combining the Spheres with the Candle Base

We are now faced with an important decision. At the moment, the jewels overlap a portion of the candle base (specifically, the **rcc1.s** portion). Because two objects cannot occupy the same space, we must decide how to resolve this situation. There are two choices:

1. We can have perfectly round jewels with dents in the side of the candle base where the jewels are mounted.

2. We can have a perfectly round base with a cylindrical bite taken out of the back of each jewel.

For this lesson, we will use the first choice.

Now we are faced with a second decision: how to achieve this result. The key is that the space the jewels occupy must be subtracted from the correct part (**rcc1.s**) of the base.

On the Command Line, create **rcc1.c** by typing:

```
comb rcc1.c u rcc1.s - sph1.r - sph2.r - sph3.r - sph4.r -
sph5.r - sph6.r - sph7.r - sph8.r<ENTER>
```

Next, bring up the Combination Editor and select **base1.r**. Change the union of **rcc1.s** in the Boolean Expression window to a union of **rcc1.c**, and click **OK**.
The tree of **base1.r** should now look like:

```
u base1.r/R
   u eto1.s
   u rcc1.c
      u rcc1.s
      - sph1.r/R
         u sph1.s
      - sph2.r/R
         u sph2.s
      - sph3.r/R
         u sph3.s
      - sph4.r/R
         u sph4.s
      - sph5.r/R
         u sph5.s
      - sph6.r/R
         u sph6.s
      - sph7.r/R
         u sph7.s
      - sph8.r/R
         u sph8.s
         u eto2.s
   - rcc2.s
```

Note that we could have achieved the same results on the Command Line by using the **rm** (remove) command to remove **rcc1.s** from **base1.r** and then adding **rcc1.c**:

```
rm base1.r rcc1.s<ENTER>
r base1.r u rcc1.c<ENTER>
```

This would have resulted in a very similar tree as above:

```
u base1.r/R
   u eto1.s
         u eto2.s
   - rcc2.s
   u rcc1.c
      u rcc1.s
      - sph1.r/R
         u sph1.s
      - sph2.r/R
         u sph2.s
      - sph3.r/R
         u sph3.s
      - sph4.r/R
         u sph4.s
      - sph5.r/R
         u sph5.s
      - sph6.r/R
```

```
            u sph6.s
         - sph7.r/R
            u sph7.s
         - sph8.r/R
            u sph8.s
```

Finally, we could have avoided making an intermediate object in the database by moving **rcc1.s** to the end of the Boolean equation for **base1.r** and then subtracting each of the jewels from **base1.r** (hence, removing material from **rcc1.s**). This would have resulted in:

```
u base1.r/R
   u eto1.s
         u eto2.s
   - rcc2.s
   u rcc1.s
   - sph1.r/R
      u sph1.s
   - sph2.r/R
      u sph2.s
   - sph3.r/R
      u sph3.s
   - sph4.r/R
      u sph4.s
   - sph5.r/R
      u sph5.s
   - sph6.r/R
      u sph6.s
   - sph7.r/R
      u sph7.s
   - sph8.r/R
      u sph8.s
```

It would be good practice to consider the relative merits of each of the approaches discussed.

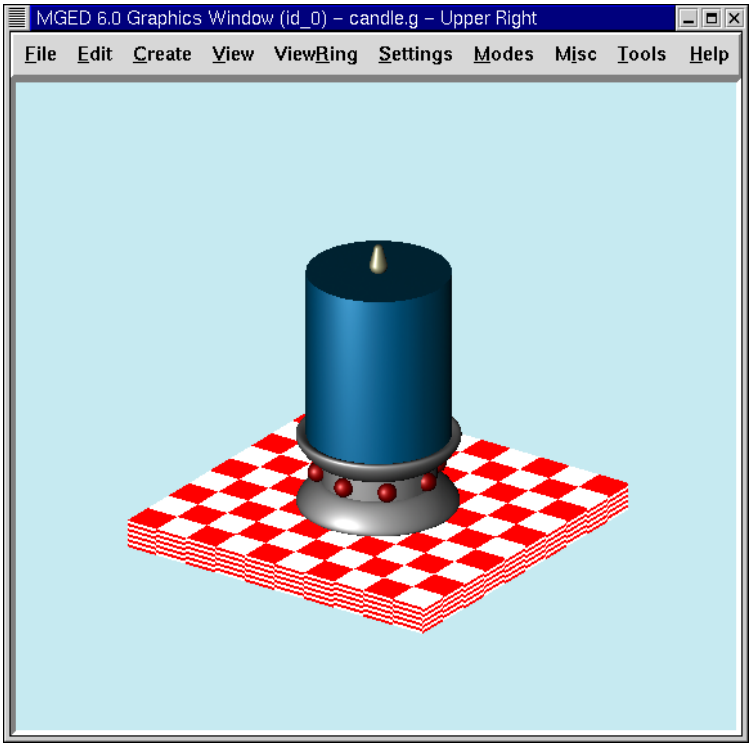
Now we need to add the jewels to the whole of **candle1.c**:

```
comb candle1.c u sph1.r u sph2.r u sph3.r u sph4.r u sph5.r
u sph6.r u sph7.r u sph8.r<ENTER>
```

There are just a couple of things left to do before you raytrace your design. If you have enabled **Multipanes** or **Draw Grid**, go back to the **Modes** menu and disable them. Then, clear your screen and draw your new design by typing in the Command Window:

```
B candle1.c table1.r
```

Your new design should appear in the Graphics Window. Open the Raytrace Control Panel and select a pale blue color (**200 236 242**) by typing the three values in the **Background Color** entry box. When you raytrace your design, it should look similar to the following one:



Raytraced Candle with Jeweled Base

Review

In this lesson, you:

- Created copies of a shape using the Solid Editor.
- Drew a grid to help position objects.
- Checked the data tree and made corrections (if needed).
- Assigned material properties using the Combination Editor.

Intentionally Left Blank

Lesson 15: Creating a Toy Truck

In this lesson, you will:

- Create a toy truck from three shapes.
- Make copies of shapes using the Solid Editor.
- Make combinations and regions of a more complex object.
- Check the data tree for accuracy.
- List contents of the database.
- Assign material properties using the Combination Editor.
- Identify the difference between **OK**, **Accept**, **Apply**, **Reset**, **Cancel**, and **Dismiss**.
- Identify the on-screen help option.
- Identify the stacker option.
- Experiment with the attributes of the camo shader.

In previous lessons, you created and edited shapes to produce simple objects. This lesson focuses on creating a slightly more complex object, a toy truck, from the Command Window. Your completed truck should look similar to the following truck:

Raytraced Toy Truck Design

Begin by creating a new database called **truck.g**.

1. Creating an rpp for the Cab of the Truck Using the In Command

To make the cab of the truck, you will create a right parallel piped using the **in** (insert) command. At the Command Window prompt, type:

```
in cab1.s rpp<ENTER>
```

MGED will ask you to enter values for XMIN, XMAX, YMIN, YMAX, ZMIN, and ZMAX. Type at the prompt:

```
0 1 0 1 0 1<ENTER>
```

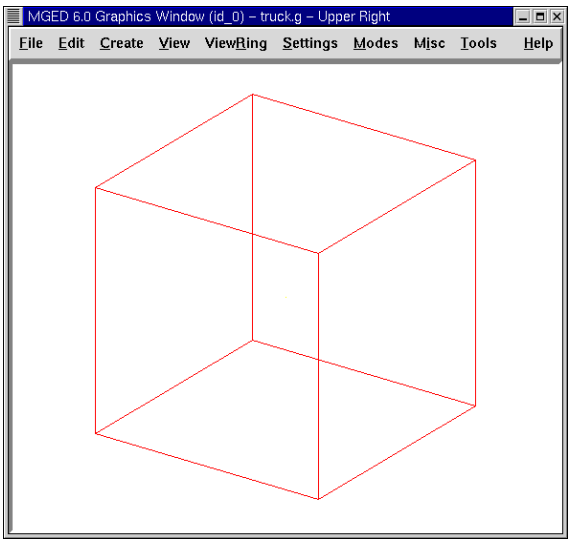
This will tell MGED to:

0	1	0	1	0	1
Make the value of the rpp's XMIN 0	Make the value of the rpp's XMAX 1	Make the value of the rpp's YMIN 0	Make the value of the rpp's YMAX 1	Make the value of the rpp's ZMIN 0	Make the value of the rpp's ZMAX 1

You could also have used the streamlined version of:

```
in cab1.s rpp 0 1 0 1 0 1<ENTER>
```

A cube shape should appear in the Graphics Window, as follows:



Wireframe Representation of Shape cab1.s

2. Using the Inside Command to Create an rpp for the Hood of the Cab

To make the hood of the cab, you will need to make another rpp shape, this time using the **inside** command. This special command was originally created to hollow out objects such as gas tanks and boxes; however, it can be used to create any new shape that has some relationship to a pre-existing shape. In this lesson, it is used to cut away material above the hood and in front of the cab.

If you are using BRL-CAD version 6.0 or later, at the Command Window prompt, type:

```
inside cab1.s caboff1.s   .5 -.1 .7 -.1 -.1 -.1<ENTER>
```

The **inside** command tells MGED to:

inside	**cab1.s**	**caboff1.s**	**.5**	**-.1**	**.7**	**-.1**	**-.1**	**-.1**
Inside the shape	named cab1.s,	create rpp called caboff1.s	Make face 1234 (bottom) .5 units thick	Make face 5678 (top) -.1 units thick	Make face 1485 (rear) .7 units thick	Make face 2376 (front) -.1 units thick	Make face 1265 (right) -.1 units thick	Make face 3478 (left) -.1 units thick

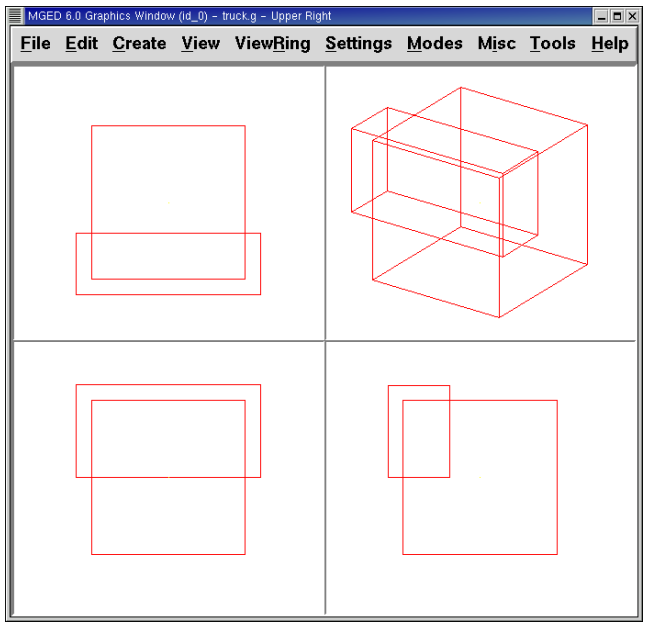
*Note: In this example, each negative thickness number indicates that **caboff1.s** will protrude through the corresponding face of **cab1.s**.*

Note that in BRL-CAD versions 6.0 and later, the faces of an rpp are numbered in a different order than in previous versions. So, if you are using a pre-6.0 release of the package, the following parameter order must be used for the aforementioned **inside** command:

```
inside cab1.s caboff1.s -.1 .7 -.1 -.1 .5 -.1<ENTER>
```

(The order of faces in this case is *front*, *rear*, *right*, *left*, *bottom*, *top* instead of that specified for versions 6.0 and later.)

When in Multipane mode, the design should resemble the following illustration.



Truck Cab with Cutoff

3. Using an rcc to Create a Wheel Well in the Cab

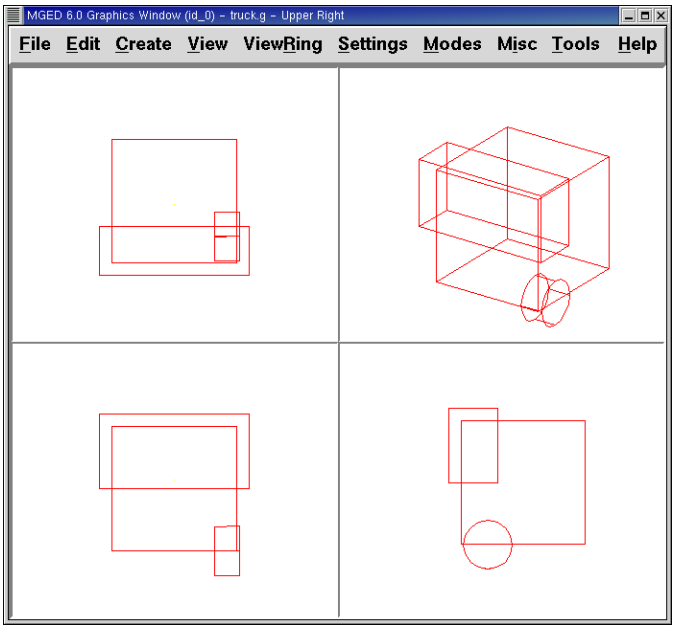
Make a cylinder (rcc) to use for cutting away a space for the first wheel of the cab. At the Command Window prompt, type:

```
make well1.s rcc<ENTER>
```

Go to the **Edit** menu and click on **Solid Selection**, then **well1.s**. After selecting a **Left** view, go back to **Edit** and select **Scale**. Reduce the size of the rcc until its diameter is about the right size for a wheel well. Go back to **Edit** and select **Rotate**. As discussed previously, one way to easily rotate the rcc is by using the **CTRL** key and the *left* mouse button to drag the top lip of the rcc down (in a straight line) until the A and C edit labels overlap. However, because we know we want to flip the object exactly 90˚ along the *x* axis, a better choice is to use the Command Line and type:

```
p 90 0 0<ENTER>
```

After doing so, you may still have to use the **Set H** and **Scale** editing options to resize and the **SHIFT** key and left mouse button to position your wheel well. When satisfied, select **Accept**. Your design should look similar to the following in Multipane mode:



Multipane View of Truck with First Wheel in Place

4. The Difference Between OK, Accept, Apply, Reset, Cancel, and Dismiss

The GUI environment of MGED offers users several options for applying, accepting, or rejecting changes made through buttons at the bottom of dialog pop-up windows. To use any of these options, just place the mouse cursor over the desired button and click the *left* mouse button.

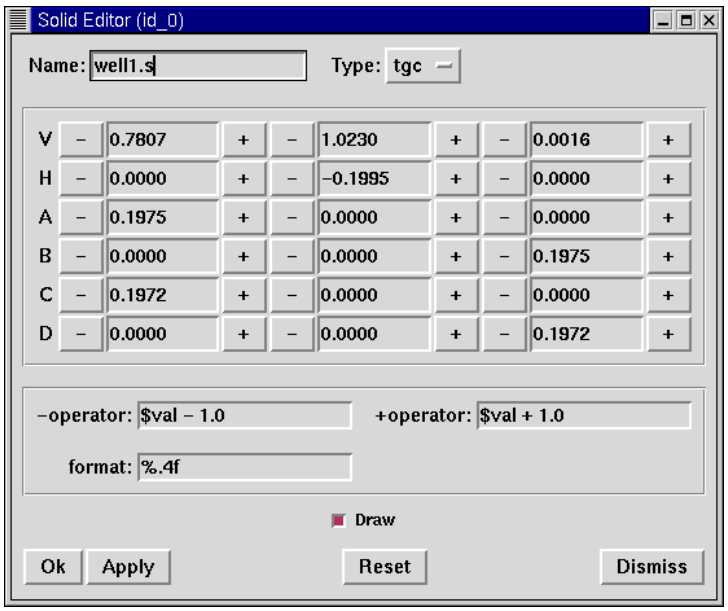
Selecting the **Accept** or **OK** button will tell MGED to record the changes you have made to a shape, region, or combination. MGED uses the **Accept** and **OK** buttons interchangeably. When you click on **Accept** or **OK**, the window you are using will automatically close.

The **Apply** button tells MGED to apply a change you have made and wait for further instructions. The window does not automatically close. This enables you to make changes to several things without having to reopen the window for each change.

The **Reset** button tells MGED to reset values you have changed in a dialog box to the last values you applied or accepted. The **Reset** button does not close the dialog box. The **Cancel** or **Dismiss** buttons discard any changes made in the dialog box and leave values unchanged from their last stored settings. These two buttons close the dialog box.

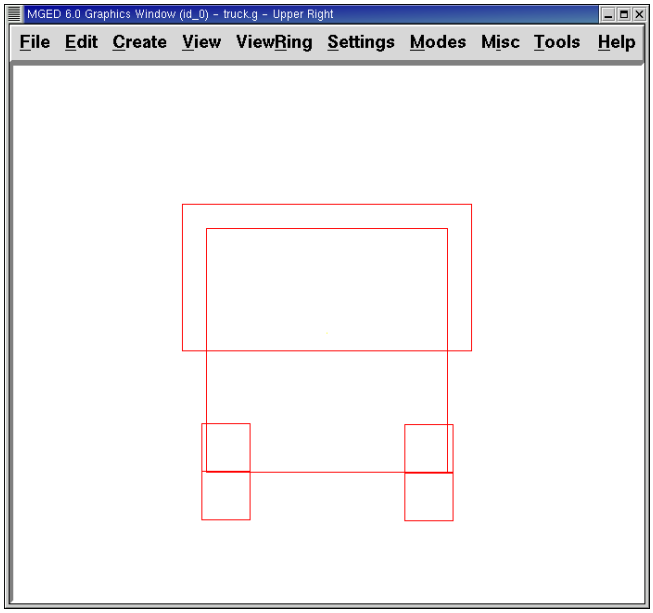
5. Using the Solid Editor to Make a Copy of the Wheel Well

To make a copy of your wheel well, go to the **Edit** menu and select **Solid Editor**, which will contain information about the last shape edited through the dialog box (or **mySolid** if nothing has been edited). Erase the old shape's name in the **Name** entry box. Type in **well1.s** and press **Reset** (or press **Enter** while the cursor is still in the **Name** entry box). The parameter values of the old shape will be replaced by those of the new shape. The Solid Editor will change and look something like the following example:



The Solid Editor

Go back to the **Name** text box and change the **1** to a **2** and click on **OK**. Change to **Front** under the **View** menu. Go to **Edit/Solid Selection** and click on **well2.s**. Use the **SHIFT** and *left* mouse button to drag the new wheel well into position, as shown in the following illustration. Check your alignment in Multipane mode and then click on **Accept** when you are finished.



Placement of the Second Wheel

6. Making a Combination of the Cab Shapes

It is now time to make a combination of the various cab shapes.

```
comb cab1.c u cab1.s - caboff1.s - well1.s - well2.s<ENTER>
```

This command tells MGED to:

comb	cab1.c	u	cab1.s	-	caboff1.s	-	well1.s	-	well2.s
Make a combination	Name it cab1.c	Make a union	of the shape cab1.s	minus	the shape caboff1.s	minus	the shape well1.s	minus	the shape well2.s

Before you go any further, you should check your data tree by typing **tree cab1.c**. The data tree should say:

```
cab1.c/
   u cab1.s
   - caboff1.s
   - well1.s
   - well2.s
```

If you type **ls** (list) at the Command Window prompt, you should find that your database is composed of the combination **cab1.c** and the shapes **cab1.s**, **caboff1.s**, **well1.s**, and **well2.s**. You will find as you make more complex objects that you will periodically refer to the list of the database to ensure it is composed of the elements you want.

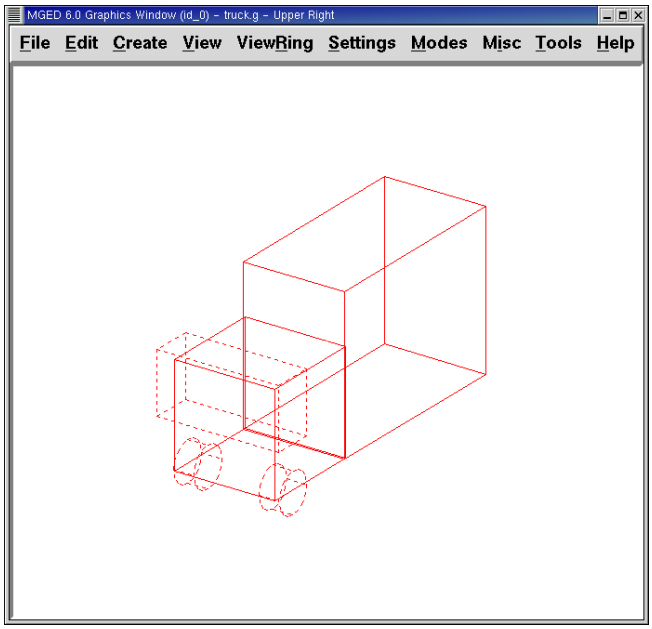
7. Creating an rpp for the Body of the Truck Using the In Command

To make the body of the truck, type at the Command Window prompt:

```
in body1.s rpp  0 2 0 1 0 1.5<ENTER>
```

By now, you should know what this command tells MGED to do. If you have forgotten, refer back to making the cab of the truck.

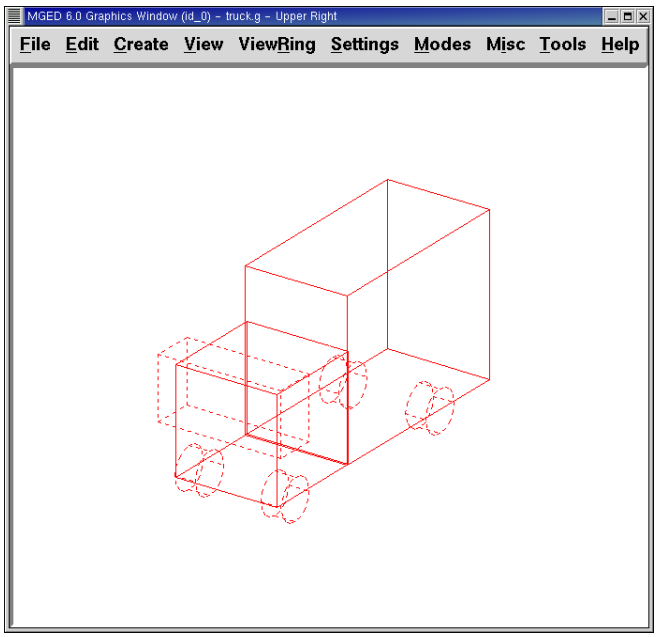
Edit the body of the truck so that its front face slightly overlaps the cab's back face. Check different views to make sure the body lines up correctly with the cab. **Accept** your changes when you are done, and then **Blast** your design. Your truck should now look like the following:



Truck Cab and Body

8. Using the Solid Editor to Make Two More Wheel Wells

To make two wheel wells for the body of the truck, repeat the steps used in making the second wheel well. Name your new shapes **well3.s** and **well4.s**. Using multiple views, move the new shapes into position so that your truck now looks similar to the following:



Wireframe Representation of Truck with Wheel Wells

9. Making a Combination of the Truck Body and Wheel Wells

Make a combination of the truck body and the two new wheel wells. Name it **body1.c**.

The tree for **body1.c** should say:

```
body1.c/
   u body1.s
   - well3.s
   - well4.s
```

10. Making a Region of the Cab and Body

Before adding wheels to the truck, you need to make a region of the cab and body. At the Command Window prompt, type:

```
r truck1.r u cab1.c u body1.c<ENTER>
```

11. Making Wheels for the Truck

Perhaps the best shape for making wheels is the torus. You can create a shape through the Command Window that has the correct size and placement for your design without further editing. However, this lesson is designed to give you practice rotating and translating shapes.

To make the first wheel, type at the Command Window prompt:

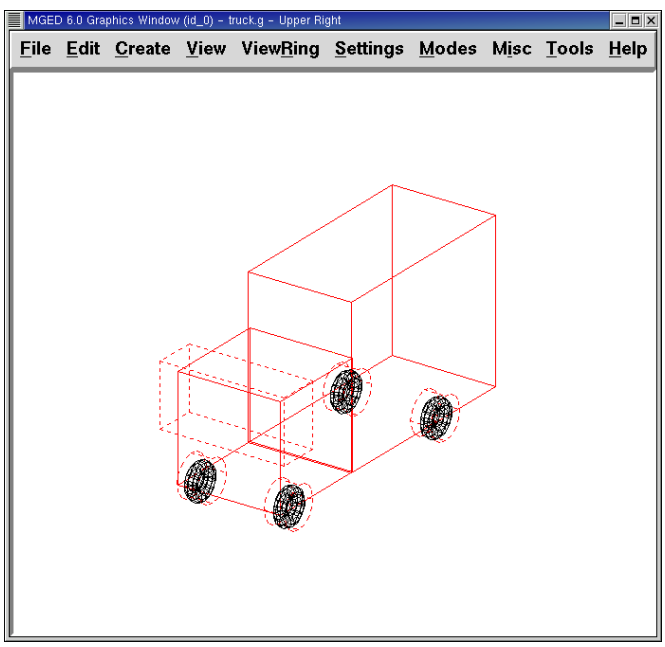
```
in wheel1.s tor 0 0 0 .5774 .5774 .5774 .18 .08<ENTER>
```

This command tells MGED to:

in	**wheel1.s**	**tor**	**0 0 0**	**.5774 .5774 .5774**	**.18**	**.08**
Create a shape	Name it wheel1.s	Make the shape a torus	Make the values of the vertex 0 0 0	Make the values for the *x*, *y*, and *z* of the normal vector .5774 .5774 .5774	Make the value of the outer radius .18	Make the value of radius 2 (the tire thickness) .08

Change **View** to **Left** and then **Edit** the position of the wheel. To correctly align the wheel with the truck, you will have to **Rotate** the tire using the **CTRL** key and *any* mouse button. **Scale** and **Translate** the wheel into position as appropriate and check your alignment from several different views. **Accept** your changes when finished.

Using the Solid Editor, make the second, third, and fourth wheels. Move each of these wheels into position until your truck looks like the following:



Wireframe Truck and Wheels

12. Making a Region of the Wheels

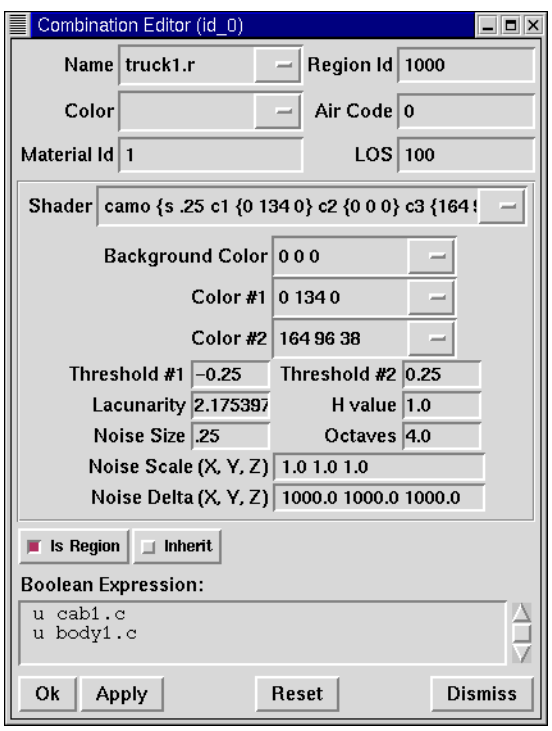
Make a region of the four wheels. When you are finished, your data tree for **wheel1.r** should say:

```
wheel1.r/R
   u wheel1.s
   u wheel2.s
   u wheel3.s
   u wheel4.s
```

13. Assigning Material Properties to the Truck Regions

Your truck is composed of two regions: **truck1.r** and **wheel1.r**. Use the Combination Editor and select **truck1.r**.

In the Combination Editor, the **camouflage** (**camo**) shader creates a pseudo-random tricolor camouflage pattern on the object using a fractal noise pattern. The shader offers lots of attributes from which to choose. For now, select a **Background Color** of black (**0 0 0**) and make **Color #1** green (**0 134 0**), and **Color #2** rust brown (**164 96 38**). To make the pattern design proportional to the truck, select a **Noise Size** of **.25** and then **Apply** the selections. The Combination Editor window for the camo shader should look like this:



Camo Shader

Apply a **black** color and **plastic** shader to the wheels (**wheel1.r**) and click on **OK**. Then raytrace your design.

14. Using the On-Screen Help Option

You have probably noticed that many of the MGED menus offer a wide variety of options from which to choose. With so many choices available, it is easy to forget what a particular selection does. To help users quickly access basic information about the various MGED options, the program offers a context-sensitive, on-screen help feature.

The on-screen help can be accessed from any menu or pop-up window by placing the mouse cursor over the name of any option in the menu or window and clicking the right mouse button. The only place this feature doesn't work is in the geometry portion of the Graphics Window, where the design is drawn.

15. The Stacker Option

In previous lessons, you applied color and a shader to an object to make it appear realistic. Sometimes, however, you will need to apply two or three shaders to an object to get the design you want.

MGED offers three categories of shaders: *paint*, *plastic*, *and light*. Any combination of these three types of shaders can be applied to the same object using the stacker option of the shader menu.

There are three plastic shaders: *glass*, *mirror*, and *plastic*. A plastic shader is used to give the perception of space. It does this by making the object's surface shiny so that it reflects light. A plastic shader is normally applied last in the stacker process.

The paint shaders are used to apply pigment and texture to the surface of an object. Color is pigment, and texture is the three-dimensional quality of the surface material (such as stucco paint).

Pigment shaders include *camo*, *texture (color)*, *texture (black/white)*, *fake star*, *cloud*, *checker*, *test map*, and *projection*. Texture shaders include *bump map*, *fbm bump*, and *tur bump*. Paint shaders are normally applied first in the stacking process and are used in combination with the plastic shader.

The light shader is used to produce illumination in the scene. This helps produce realism in the final image. The light shader is technically complex and is not discussed in this tutorial.

The camo shader involves applying pigments, in a random pattern, to the surface of an object. The camo shader doesn't indicate the three-dimensional nature of an object. If

you want your design to show depth, you will need to stack the camo shader and the plastic shader.

16. Using the Stacker Option

To use the stacker option, open the Combination Editor and select **truck1.r**. Click on the button to the right of the **Shader** entry box and then select **stack** from the drop-down menu. A button with the words **Add Shader** will appear under the text box. Click on the button and then select **camouflage**. Set the **Background Color** to black (**0 0 0**), **Color #1** to green (**0 134 0**), and **Color #2** to rust brown (**164 96 38**). Make the **Noise Size** .**25**. Click on **Add Shader** once again and select **plastic**.

At this point, your Combination Editor window may have gone off the bottom of the screen. If this happens, reduce the size of the window as much as you can and then drag it up to the top of the screen. The buttons at the bottom of the box should now appear, and you can **Apply** your selections.

	Caution: When using the stacker option, you need to keep track of the number of characters and spaces in the shader text box. MGED versions prior to release 6.0 will only recognize 64 characters/spaces, so be careful stacking shaders with complex attributes.

17. Making a Combination of the Truck Regions

To make a combination of the two truck regions, type at the Command Window prompt:

```
comb truck1.c u truck1.r u wheel1.r<ENTER>
B truck1.c<ENTER>
```

Your data tree for **truck1.c** should read:

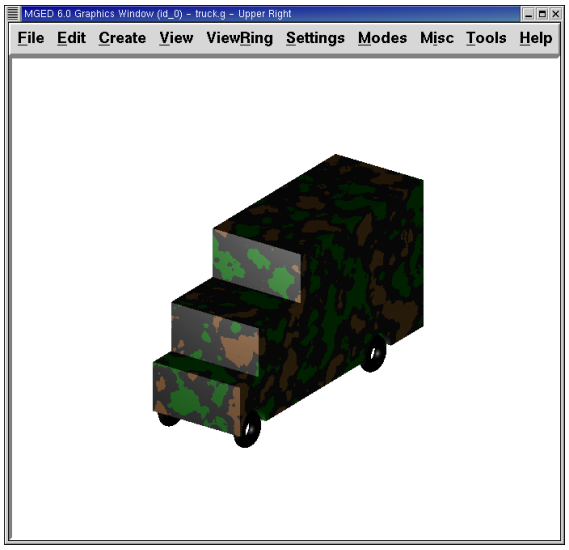
```
truck1.c/
   u truck1.r/R
      u cab1.c/
         u cab1.s
         - caboff1.s
         - well1.s
         - well2.s
      u body1.c
         u body1.s
         - well3.s
         - well4.s
   u wheel1.r/R
      u wheel1.s
      u wheel2.s
```

```
u wheel3.s
u wheel4.s
```

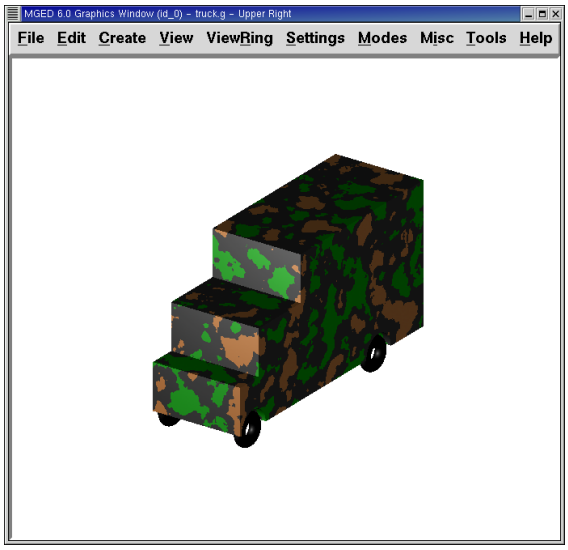
18. Raytracing the Truck

The last step in creating your truck is to raytrace your design. When the raytracer has finished, notice that the top and one side of the truck are very dark. This is because there is very little light falling on them. Because we have not specified any light sources for our scene, MGED provides us with a set of default lights. These defaults consist of a dim light at the location of the viewer and a brighter one located to the left and below the viewer. Since the primary light is not really shining on one side of the truck, it is dark.

There is a special adjustment we can make to improve the overall brightness of the scene. We can adjust the amount of ambient light, which is light that does not come from a particular light source but is a measure of the light generally present in the scene. To adjust the amount of ambient light, click on the **Advanced Settings** button in the Raytrace Control Panel. Next to **Other Options**, type **-A .9** and click **Dismiss**. Now when you raytrace, you will get a much lighter image.



Truck with Default Lighting



Truck with Added Ambient Light

Review

In this lesson, you:

- Created a toy truck from three shapes.
- Made copies of shapes using the Solid Editor.
- Made combinations and regions of a more complex object.
- Checked the data tree for accuracy.
- Listed contents of the database.
- Assigned material properties using the Combination Editor.
- Identified the difference between **OK**, **Accept**, **Apply**, **Reset**, **Cancel**, and **Dismiss**.
- Identified the on-screen help option.
- Identified the stacker option.
- Experimented with the attributes of the camo shader.

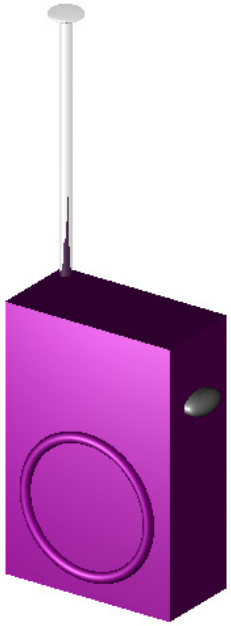
Intentionally Left Blank

Lesson 16: Learning Modeling Techniques and Structures

In this lesson, you will be:

- Making the shapes of the walkie-talkie radio into regions.
- Gathering the regions into an assembly combination.
- Assigning material properties to the regions.
- Adding internal components to the radio.
- Creating specialty models of the radio.
- Redefining the structure of the radio.

In Lesson 2, we made the basic shapes of a walkie-talkie radio to gain experience in viewing objects. Now that we have successfully modeled a few simple objects, let's return to the radio to make it more realistic and, in so doing, discuss logical techniques and structures in modeling. When finished, our radio should look as follows:



Walkie-Talkie Radio

1. Making the Shapes into Regions

Open the database **radio.g** that you created in Lesson 2. In the Command Window, use the **ls** command to list all the contents of your radio. It should read as follows:

```
ant.s     btn.s     knob.s
body.s    btn2.s    spkr.s
```

Now what does this list really contain? Parts to a model radio? Not really. What the list actually comprises is just a collection of *shapes* (which we have hinted at by using a **.s** suffix) that (1) do not have material properties, and therefore (2) do not occupy space.

	Remember, in MGED no shape truly becomes an object until it is included in a *region*, which, by definition, is an object or collection of objects that has a common material type.

So our first order of business is to identify the major parts of the radio so we can properly define the regions. So far, our choices are fairly simple. The radio basically consists of (1) a body, which houses the speaker and all of the internal parts; (2) an antenna; (3) a volume control knob, and (4) a talk button. These should be our four regions.

Most of these shapes were fairly straightforward to create, with each item consisting of just one or two primitive shapes. However, if we think of the radio as a real-world object, the body of the radio is actually more complex than just a solid box with a few shapes glued to its surface. (Remember that all objects are solid unless constructed to be otherwise.) Therefore, let's start with the main component of the radio—the body.

The Body of the Radio

If we think about it, the body of a radio is actually a hollow case. So, the first thing we need to do is hollow out the case's interior to make room for internal components. To do this, we can use the **inside** command to create a shape, which we'll call **cavity.s**:

```
inside body.s cavity.s 1 1  1 1  1 1<ENTER>
```

Now, we'll make a region called **case.r** and define it as what's left of **body.s** after **cavity.s** has been subtracted out. The command should look like this:

```
r case.r u body.s - cavity.s<ENTER>
```

	Remember that the **inside** command was originally created to hollow out objects such as gas tanks and boxes; however, it can also be used to create any new cutaway shape that has some relationship to a pre-existing shape.

With our case now made, we can proceed to cut several holes through this structure to accommodate the antenna, the volume control knob, and the talk button. To do this, we must subtract the three shapes from the case as follows:

```
r case.r - ant.s - knob.s - btn.s<ENTER>
```

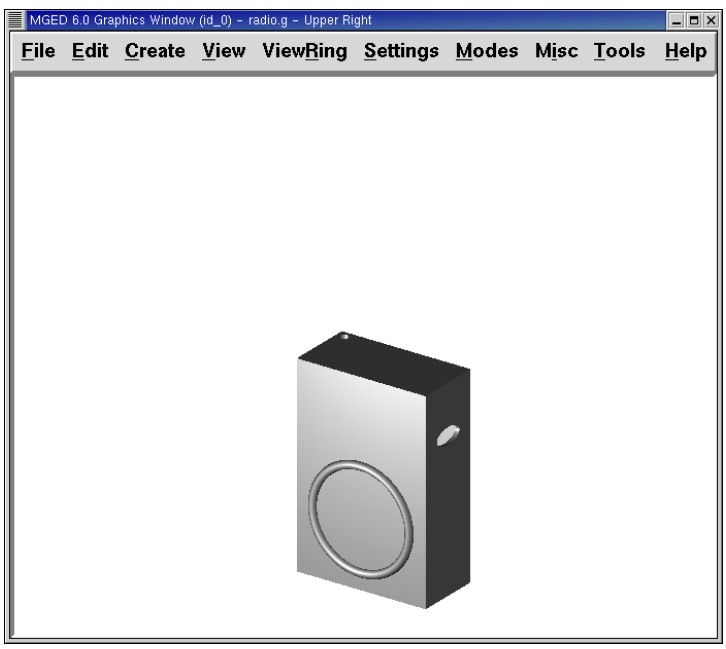
Finally, we need to "glue" the lip around the speaker to the case's front face by typing

```
r case.r u spkr.s<ENTER>
```

Our body is now finished. Note that an experienced modeler would probably have combined the preceding three Boolean functions into a single command as follows:

```
r case.r u body.s - cavity.s - ant.s - knob.s - btn.s u
spkr.s<ENTER>
```

If we were to raytrace **case.r** at this point, we would see the following:



Radio with Component Cutouts and Default Material Properties

Note the hole for the antenna in the top of the case and the hole for the talk button on the side of the case. We will now fill these holes with their respective components.

Precedence Review

The order in which these primitives were unioned and subtracted is important. We unioned in **spkr. s** last so that all the subtractions would apply to **body.s**. The rules of precedence for Boolean operators indicate that subtraction and intersection have a higher priority than union (meaning that they are performed first).

Although the following operation is not in proper MGED syntax, it does illustrate the implied parentheses that precede and follow the union operators in our last command:

```
r case.r u (body.s – cavity.s – ant.s – knob.s –
btn.s) u (spkr.s) <ENTER>
```

Optionally, we could've unioned in **spkr.s** before **body.s** as follows:

```
r case.r u spkr.s u body.s – cavity.s – ant.s –
knob.s – btn.s<ENTER>
```

Let's consider, however, what would have happened if we had done the following:

```
r case.r u body.s u spkr.s – cavity.s – ant.s –
knob.s – btn.s<ENTER>
```

In this last case, operator precedence would have caused the program to subtract **cavity.s**, **ant.s**, **knob.s**, and **btn.s** from **spkr.s**. Nothing would have been subtracted from **body.s**. Therefore, the holes in the case would not have been created.

Subtracting **cavity.s**, **ant.s**, **knobs**, and **btn.s** from **spkr.s** would have produced no apparent effect because they do not overlap the volume of **spkr.s.**

The Other Regions

Making the talk button is simpler than making the case. The button consists of the union of two primitive shapes. To make them into a region, type

```
r button.r u btn.s u btn2.s<ENTER>
```

The volume knob and antenna are even simpler. They are single primitive shapes that can be made into regions by typing

```
r knob.r u knob.s<ENTER>
r ant.r u ant.s<ENTER>
```

2. Gathering the Regions into an Assembly Combination

Now let's take all of the regions we have made so far and gather them into an assembly (or group) combination called **radio.c** so that we can keep all of these parts together. There are several ways to do this. One way would be to use a similar method to the one we used to make the regions:

```
comb radio.c u case.r u button.r u knob.r u ant.r<ENTER>
```

A shortcut, however, would be to use the **g** (group) command as follows:

```
g radio.c case.r button.r knob.r ant.r<ENTER>
```

Unlike the **comb** command, the **g** command assumes that all of the items specified will be unioned together, and so no Boolean operators need to be specified.

A final improvement would include using the database name wildcard ***.r** to quickly and easily specify **all** of the regions in the database:

```
g radio.c *.r<ENTER>
```

If we now tree **radio.c**, we should get the following output in the Command Window.

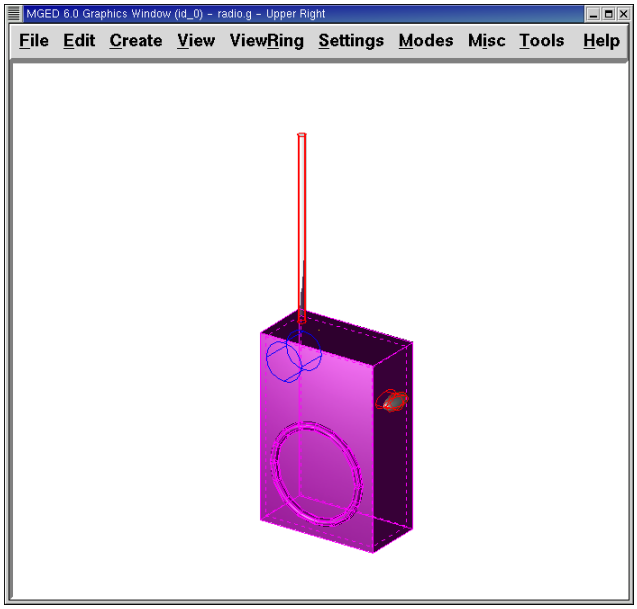
```
radio.c/
   u case.r/R
      u body.s
      - cavity.s
      - ant.s
      - knob.s
      - btn.s
      u spkr.s
   u button.r/R
      u btn.s
      u btn2.s
   u knob.r/R
      u knob.s
   u ant.r/R
      u ant.s
```

3. Assigning Material Properties to the Regions

Thus far, the objects we have created have no material properties other than the gray plastic that MGED assigns by default to any object without assigned material values. Now let's improve our design by assigning other material properties to the components.

We'll give the antenna a realistic look by opening the Combination Editor, choosing **ant.r** from the drop-down Name menu, selecting **mirror** from the drop-down Shader menu, and clicking on **Apply**.

We'll let the other components remain with the default plastic, but we'll assign them different colors. With the Combination Editor still open, select **case.r** from the drop-down Name menu, select **magenta** from the drop-down Color menu, and then click **Apply.** Use the same method to assign the volume control knob (**knob.r**) a **blue** color. For the talk button (**button.r**), let's keep it gray by leaving the default values in place. The design should appear similar to the following when raytraced in **Underlay** mode:



Radio with Material Properties Assigned

As we look at our radio now, we can see that the antenna looks a little bit like a straw. In reality, it should have a small cap on the end so that we can raise and lower the antenna. We can approximate this shape by creating an ellipsoid (which we'll call **ant2.s**) and unioning it in with the rest of the antenna as follows:

```
in ant2.s ell1  2 2 94   0 0 1  3<ENTER>
r ant.r u ant2.s<ENTER>
```

4. Adding Internal Components

Our radio is looking more and more realistic; however, it is still just a hollow shell. Let's further improve it by making a circuit board to go inside the case. To do this, type:

```
in board.s rpp 3 4 1 31 1 47<ENTER>
r board.r u board.s<ENTER>
```

Let's give the board a green semi-shiny color. The easiest way to do this is via the Combination Editor, but this time let's use the Command Line approach. Type:

```
mater board.r "plastic sh=4" 0 198 0 1<ENTER>
```

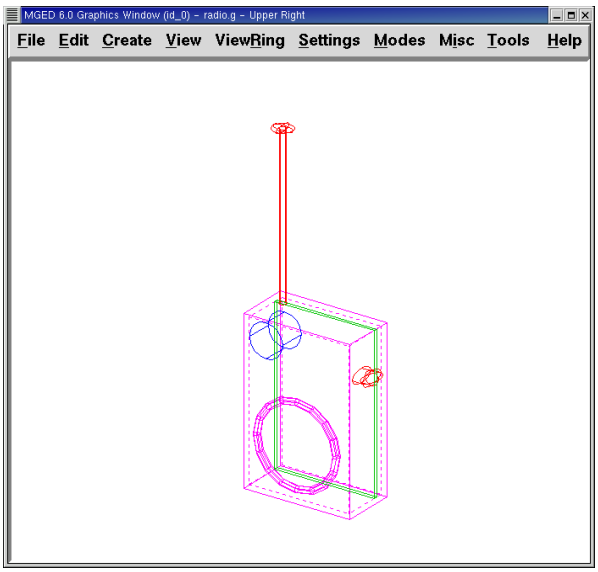
Diagrammed, this command says to:

mater	board.r	"plastic sh=4"	0 198 0	1
Assign material properties to	the region called board.r.	Apply the plastic shader with a shininess of 4.	Give it a green color	Inherit color material type

Finally, we'll include the board with the rest of the components in **radio.c** as follows:

```
g radio.c board.r<ENTER>
```

Our radio should now look like the following:



Wireframe Radio with Circuit Board Added

In addition, the tree for **radio.c** should now look as follows:

```
radio.c/
   u case.r/R
      u body.s
      - cavity.s
      - ant.s
      - knob.s
      - btn.s
      u spkr.s
   u button.r/R
      u btn.s
      u btn2.s
   u knob.r/R
      u knob.s
   u ant.r/R
      u ant.s
      u ant2.s
   u board.r/R
      u board.s
```

5. Making Specialty Models of the Radio

Now, what would happen to the circuit board if we were to raytrace the radio at this point? It would disappear because it lies within the case. So how can we make the circuit board visible outside of the case?

There are two common ways to do this: a *transparent* view and a *cutaway* view. Each method has its advantages and disadvantages. With the transparent view, the Boolean operations remain unchanged, but some of the material properties of the "outside shell" are altered to better view interior parts of the model. With the cutaway view, the material properties remain unchanged, but some of the Boolean operations are altered to remove parts of the model that are obstructing our view of other parts. We will try both ways to view the inside of our radio.

Different Approaches to Creating Specialty Models

An important point to note here is that the transparent and cutaway views are specialty models. They are similar in nature to items a manufacturer might make for special purposes. For example, an automobile manufacturer makes cars for everyday use, but also makes modified versions for display at certain events. The body panels might be replaced with a transparent material or be partially cut away to reveal interior components.

Good modeling practice follows the same pattern. The actual model of an item should not have to be changed in order to create a specialty view of it. Instead, a modified version of the item should be created. Thus, the modeler will not have to worry about remembering to return the model to the original condition after its special-purpose use, and the modeler can also retain the "display model" for future use.

There are two common approaches to making these specialty models: First, the modeler can copy the original and replace components with modified versions. Second, the modeler can create new, unique parts from scratch and construct the modified item. The method chosen is a matter of personal choice and is usually determined by the extent of the modifications being done and the complexity of the original object.

Transparent View

Making a specialty radio with a transparent case would probably be the easiest way to view the circuit board inside. All we have to do is make a copy of our present radio case and modify its material properties. We'll call the specialty case **case_clear.r**. Type

```
cp case.r case_clear.r<ENTER>
```

We can now use the Combination Editor to set the material properties on this case without affecting the "master" design of the radio. When this has been done, we can combine this modified case with the other unchanged radio components and group them as a new specialty radio named **radio_clear.c**.

To set the material properties of **case_clear.r**, choose **plastic** from the drop-down menu to the right of the Shader entry box in the Combination Editor. (Although this is the shader that is used by default, we want to explicitly select it in order to change one of its values.) Now change the **Transparency** of the case to a value of **.8. Apply** the change and close the Combination Editor.

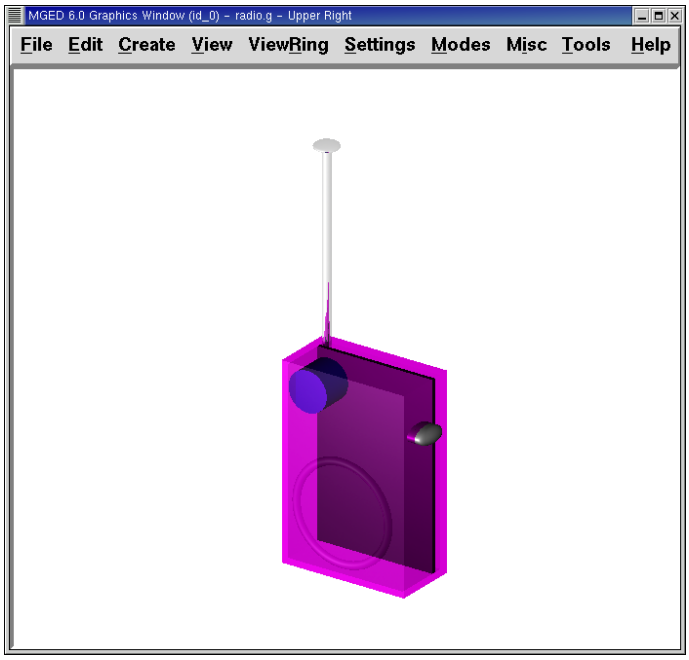
Finally, create the specialty radio combination by typing:

```
g radio_clear.c case_clear.r button.r knob.r ant.r
board.r<ENTER>
```

and then Blast the display with

```
B radio_clear.c<ENTER>
```

Now raytrace your design to view the resulting effect. The new transparent case should appear similar to the following:



Transparent View of the Radio

As shown in the following tree diagram, the structure of this specialty **radio_clear.c** is not much different than that of the regular **radio.c**. The only difference is that **case.c** has been replaced with **case_clear.c.**

```
radio_clear.c/
   u case_clear.r/R
      u body.s
      - cavity.s
      - ant.s
      - knob.s
      - btn.s
      u spkr.s
   u button.r/R
      u btn.s
      u btn2.s
   u knob.r/R
      u knob.s
   u ant.r/R
      u ant.s
      u ant2.s
   u board.r/R
      u board.s
```

	Notice in the preceding figure that the color chosen for the transparent case does influence the appearance of the internal objects. Although we made the circuit board green, the filter effect of the transparent magenta case—which allows no green light to enter or exit the case—causes the board to appear to be dark purple. This is okay in our situation. However, if accuracy in color is important in a model, the modeler should remember to select a neutral color (such as white or light gray) for the transparent object.

Cutaway View

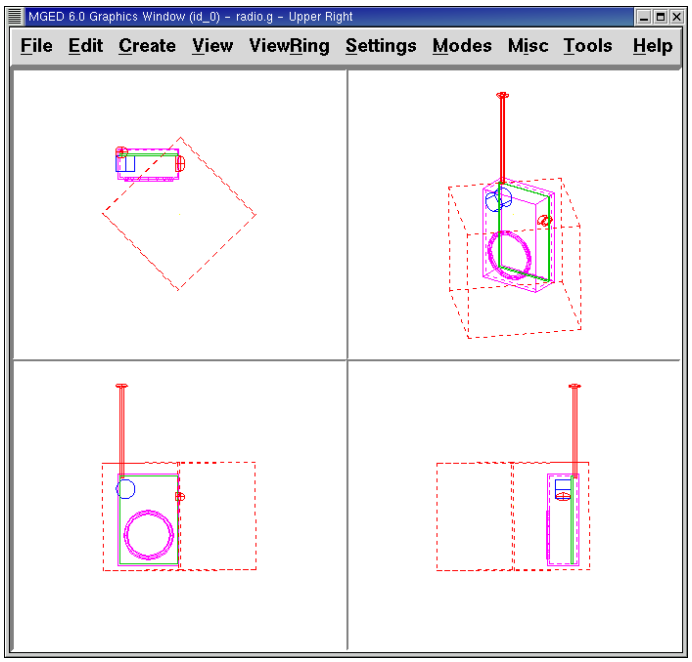
Another way we can make the interior components of the radio visible is to create a cutaway view. Although it is a little more complex to make than the transparent view was, the cutaway view offers a particularly interesting way to view geometry.

There are several ways to make the cutaway view. Probably the easiest way is to use the "chainsaw" method to cut off part of the radio and reveal what is inside.

To do this, create an arb8 called **cutaway.s**, which will be used to cut off the front corner of the radio. Because this is a *cutting shape* (i.e., it is simply used to erase a portion of another shape and will not actually be viewed), the dimensions of the arb8 are not critical. The only concern is that **cutaway.s** be as tall as the case so that it completely removes a corner from it.

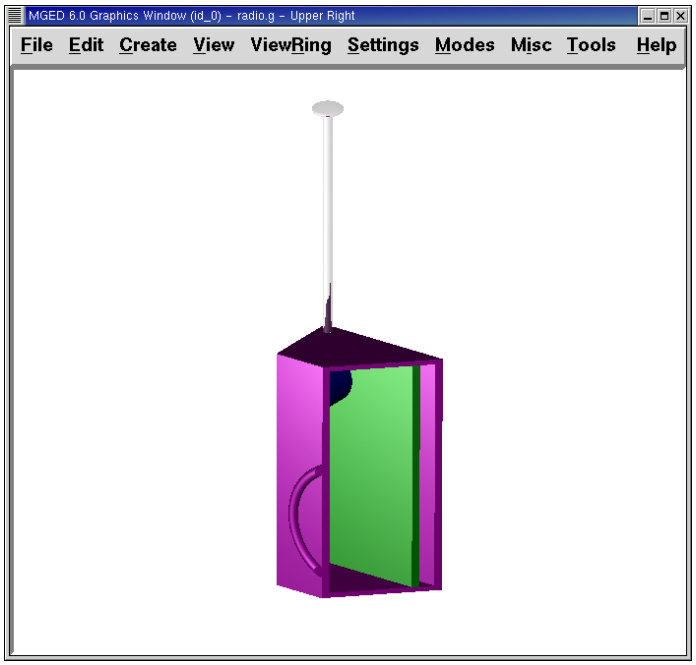
Use the Shift Grips and multiple views (especially the **Top** view) to align **cutaway.s** so that it angles diagonally across the top of the radio (as shown in the following wireframe representation). When you've aligned the shape the way you want it, create the following **radio_cutaway.c** combination that unions in **radio.c** and subtracts out the shape (**cutaway.s**) that is covering what you want to see (**board.r**):

```
comb radio_cutaway.c u radio.c - cutaway.s<ENTER>
```



Multipane View of Cutting Solid

Blast the **radio_cutaway.c** combination onto the display and raytrace. Depending on how your arb8 intersected the radio, the cutaway should look similar to the following:



Cutaway View of Radio with Circuit Board Cut Off

Notice in the preceding figures that **cutaway.s** removes everything it overlaps (including part of the circuit board). This is okay if we just want to see inside the case. However, if we want to see all of the circuit board and any other component overlapped by **cutaway.s** (e.g., **button.r**), we would have to adjust our Boolean operations a little so that the cutaway is subtracted only from our case.

To do this, we basically have two options: (1) we could move **cutaway.s** in the structure so that it is subtracted from *only* **case.r**, or (2) we could move **cutaway.s** in the structure so that it is subtracted from *both* **body.s** and **spkr.s**, the two components that make up **case.r**. While both of these options would produce the same effect, the first method requires just one subtraction, whereas the second method potentially provides more control by having the user select the individual components that will subtract out the cutting shape.

Take a minute and compare the following trees for the cutaways we have discussed so far. Especially note the position of **cutaway.s** in the different structures. Also, note that when **cutaway.s** was subtracted from a particular region or combination, the name of that region or combination was changed. The reasoning behind this goes back to our original discussion of specialty models. Remember that our purpose is to create a new special-purpose model, not change the existing model. So, we must change the name of any region or combination that contains any modified components or structures. If we don't, the master model will also be changed.

```
radio_cutaway.c/
   u radio.c/
         u case.r/R
            u body.s
            - cavity.s
            - ant.s
            - knob.s
            - btn.s
            u spkr.s
      u button.r/R
         u btn.s
         u btn2.s
      u knob.r/R
         u knob.s
         u ant.r/R
            u ant.s
            u ant2.s
      u board.r/R
         u board.s
      -  cutaway.s
```

Cutaway Subtracted from Entire Radio

```
radio_cutaway.c/
   u radio_casecut.c/
         u case.r/R
            u body.s
            - cavity.s
            - ant.s
            - knob.s
            - btn.s
            u spkr.s
         - cutaway.s
      u button.r/R
         u btn.s
         u btn2.s
      u knob.r/R
         u knob.s
         u ant.r/R
            u ant.s
            u ant2.s
      u board.r/R
         u board.s
```

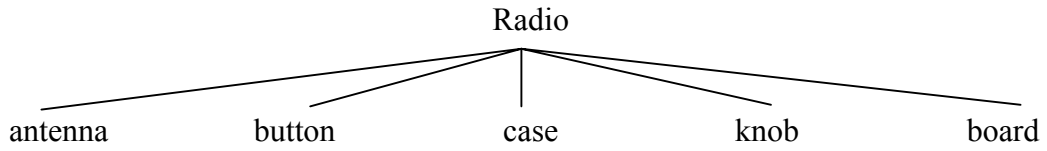
Cutaway Subtracted from case.r

```
radio_cutaway.c/
   u radio_compcut.c/
         u case_compcut.r/R
            u body.s
                  - cutaway.s
            - cavity.s
            - ant.s
            - knob.s
            - btn.s
            u spkr.s
                  - cutaway.s
      u button.r/R
         u btn.s
         u btn2.s
      u knob.r/R
         u knob.s
         u ant.r/R
            u ant.s
            u ant2.s
      u board.r/R
         u board.s
```

Cutaway Subtracted from body.s **and** spkr.s

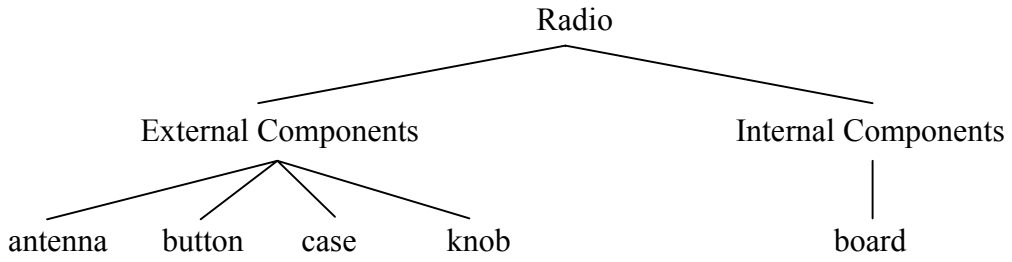
6. Redefining the Structure of the Radio

As shapes are added in a design, the modeler often finds that the structure or association of components needs to change. Thus, we should pause at this point and consider how our radio is structured. While there are many ways to structure a model, two common modeling categories are *location* and *functionality.* For our radio, we have so far grouped everything together under the general category of **Radio**, as shown in the following:



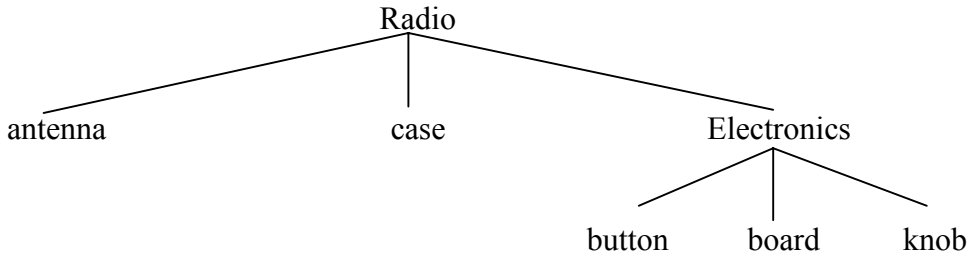
Current Radio Structure

If we wanted to categorize our components according to location, however, we might structure the model as follows:



Location-Based Structure of Radio

If we wanted to define our components according to functionality, we might structure the model another way. For instance, to repair an actual radio, we would open the case, take out the circuit board, fix it, and put it back in. When taking out the board, however, the knob and button would probably be attached to it in some way, and so they too would need to come out. Accordingly, our structure should be changed as shown in the following diagram to associate the knob and button with the circuit board.



Function-Based Structure of Radio

To accomplish this restructuring according to functionality, create an assembly called **electronics.c** to hold these components together. Type:

```
g electronics.c board.r knob.r button.r<ENTER>
```

Of course, we now need to remove **board.r**, **knob.r,** and **button.r** from the **radio.c** assembly so that when **electronics.c** is added to the **radio.c** assembly, we won't have the knob and button included twice in the model. To do this, use the **rm** (remove) command:

```
rm radio.c board.r knob.r button.r<ENTER>
```

and then union in the electronics assembly:

```
g radio.c electronics.c<ENTER>
```

Now the tree for **radio.c** should appear as follows:

```
radio.c/
   u case.r/R
      u body.s
      - cavity.s
      - ant.s
      - knob.s
      - btn.s
      u spkr.s
   u ant.r/R
      u ant.s
      u ant2.s
   u electronics.c/
      u board.r/R
         u board.s
      u knob.r/R
         u knob.s
      u button.r/R
         u btn.s
         u btn2.s
```

Now let's remake our cutaway view. This time, let's do what we discussed earlier and make the cutaway remove material from only the case, showing all the other components.

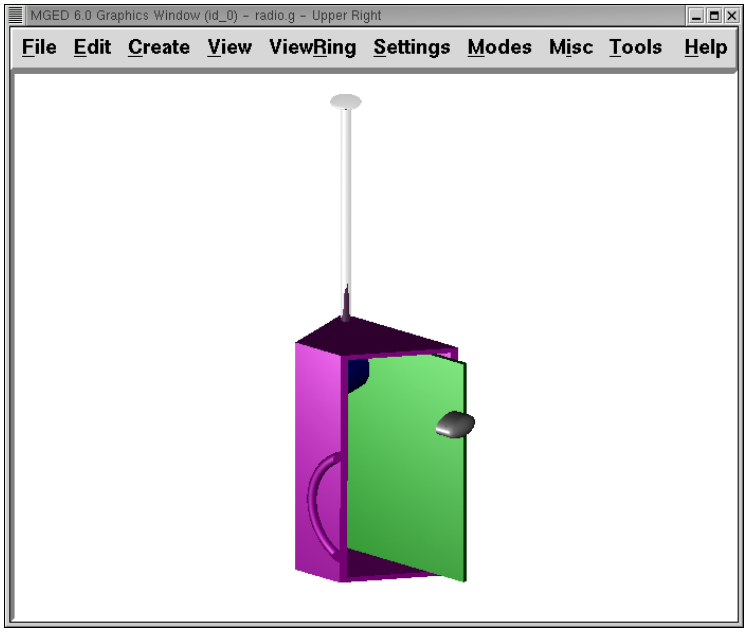
First, we need to get rid of the old **radio_cutaway.c**, which was based on our previous structure. To do this, type

```
kill radio_cutaway.c<ENTER>
```

and then remake the combination by typing

```
comb radio_cutaway.c u case.r - cutaway.s u electronics.c u
ant.r<ENTER>
```

Now when we **Blast** the display and raytrace **radio_cutaway.c**, we should see the following:



View of Radio with Just the Case Cut Away

Review

In this lesson, you:

- Made the shapes of the walkie-talkie radio into regions.
- Gathered the regions into an assembly combination.
- Assigned material properties to the regions**.**
- Added internal components to the radio.
- Created specialty models of the radio.
- Redefined the structure of the radio.

Intentionally Left Blank

Appendix A:

MGED Commands

Intentionally Left Blank

MGED User Commands

%	3ptarb	?	?devel	?lib
B	E	M	Z	adc
ae	analyze	animmate	apropos	aproposdevel
aproposlib	arb	arced	area	arot
attach	bev	c	cat	center
color	comb	comb_color	copyeval	copymat
cp	cpi	d	dall	db
db_glob	dbconcat	debugbu	debugdir	debuglib
debugmem	debugnmg	decompose	delay	dm
draw	dup	e	eac	echo
edcodes	edcolor	edcomb	edgedir	edmater
eqn	erase	erase_all	ev	exit
expand	extrude	eye_pt	e_muves	facedef
facetize	find	fracture	g	garbage_collect
gui	help	helpdevel	helplib	history
i	idents	ill	in	inside
item	joint	journal	keep	keypoint
kill	killall	killtree	knob	l
labelvert	listeval	loadtk	lookat	ls
l_muves	make	make_bb	mater	matpick
memprint	mirface	mirror	mrot	mv
mvall	nirt	nmg_collapse	nmg_simplify	oed
opendb	orientation	orot	oscale	overlay
p	pathlist	paths	permute	pl
plot	polybinout	pov	prcolor	prefix
press	preview	prj_add	ps	push
putmat	q	qorot	qray	query_ray
quit	qvrot	r	rcodes	read_muves
red	redraw_vlist	refresh	regdebug	regdef
regions	release	rfarb	rm	rmater
rmats	rot	rotobj	rrt	rt
rtcheck	savekey	saveview	sca	sed
setview	shader	shells	showmats	size
solids	status	summary	sv	sync
t	ted	title	tol	tops
tra	track	translate	tree	t_muves
units	vars	vdraw	view	viewsize
vnirt	vquery_ray	vrmgr	vrot	wcodes
whatid	which_shader	whichair	whichid	who
wmater	x	xpush	zoom	

%

Start a "/bin/sh" shell process for the user. The *mged* prompt will be replaced by a system prompt for the shell, and the user may perform any legal shell commands. The *mged* process waits for the shell process to finish, which occurs when the user exits the shell. This only works in a command window associated with a tty (i.e., the window used to start *mged* in classic mode).

Examples:

`mged` **%**

-- Start a new shell process.

`$` **ls -al**
-- Issue any shell commands.

`$` **exit**
-- Exit the shell.

`mged`
-- Continue editing in *mged*.

3ptarb [*arb_name x1 y1 z1 x2 y2 z2 x3 y3 z3 x|y|z coord1 coord2 thickness*]

Build an ARB8 shape by extruding a quadrilateral through a given *thickness*. The arguments may be provided on the command line; if they are not, they will be prompted for. The *x1, y1,* and *z1* are the coordinates of one corner of the quadrilateral. *x2, y2, z2,* and *x3, y3, z3* are the coordinates of two other corners. Only two coordinates of the fourth point are specified, and the code calculates the third coordinate to ensure all four points are coplanar. The *x|y|z* parameter indicates which coordinate of the fourth point will be calculated by the code. The *coord1* and *coord2* parameters supply the other two coordinates. The direction of extrusion for the quadrilateral is determined from the order of the four resulting points by the right-hand rule; the quadrilateral is extruded toward a viewer for whom the points appear in counter-clockwise order.

Examples:

`mged` **3ptarb**

-- Start the *3ptarb* command.

`Enter name for this arb:` **thing**
-- The new ARB8 will be named *thing*.

`Enter X, Y, Z for point 1:` **0 0 0**
-- Point one is at the origin.

`Enter X, Y, Z for point 2:` **1 0 0**
-- Point two is at (1, 0, 0).

`Enter X, Y, Z for point 3:` **1 1 0**
-- Point three is at (1, 1, 0).

`Enter coordinate to solve for (x, y, or z):` **z**
-- The code will calculate the *z* coordinate of the fourth point.

`Enter the X, Y coordinate values:` **0 1**
-- The *x* and *y* coordinates of the fourth point are 0 and 1.

`Enter thickness for this arb:` **3**
-- The new ARB8 will be 3 units thick.

`mged` **3ptarb thing 0 0 0 1 0 0 1 1 0 z 0 1 3**
-- Same as above example, but with all arguments supplied on the command line.

?

Provide a list of available *mged* commands. The ?devel, ?lib, help, helpdevel, and helplib commands provide additional information on available commands.

Examples:

`mged` **?**

-- Get a list of available commands.

?devel

Provide a list of available *mged developer* commands. The ?, ?lib, help, helpdevel, and helplib commands provide additional information on available commands.

Examples:

`mged` **?devel**

-- Get a list of available *developer* commands.

?lib

Provide a list of available *BRL-CAD* library interface commands. The ?, ?devel, help, helpdevel, and helplib commands provide additional information on available commands.

Examples:

`mged` **?lib**

-- Get a list of available *BRL-CAD* library interface commands.

B [-*C#/#/#*] <*objects*>

Clear the *mged* display of any currently displayed objects, then display the list of objects provided in the parameter list. Equivalent to the Z command followed by the command draw <*objects*>. The -*C* option provides the user a way to specify a color that overrides all other color specifications including combination colors and region id based colors.

Examples:

`mged` **B some_object**

-- Clear the display, then display the object named *some_object*.

E [-s] <*objects*>

Display *objects* in an evaluated form. All the Boolean operations indicated in each object in *objects* will be performed, and a resulting faceted approximation of the actual objects will be displayed. Note that this is usually much slower than using the usual *draw* command. The -*s* option provides a more accurate, but slower, approximation.

Examples:

`mged` **E some_object**

-- Display a faceted approximation of *some_object*.

M *1|0 xpos ypos*

Send an *mged* mouse (i.e., defaults to a middle mouse button) event. The first argument indicates whether the event should be a button press (*1*) or release (*0*). The *xpos* and *ypos* arguments specify the mouse position in *mged* screen coordinates between -2047 and +2047. With the default bindings, an *mged* mouse event while in the viewing mode moves the view so that the point currently at screen position (*xpos*, *ypos*) is repositioned to the center of the *mged* display (compare to the center

command). The *M* command may also be used in other editing modes to simulate an *mged* mouse event.

Examples:

`mged` **M 1 100 100**

-- Translate the point at screen coordinates (100, 100) to the center of the *mged* display.

Z

Zap (i.e., clear) the *mged* display.

Examples:

`mged` **Z**

-- Clear the *mged* display.

---

adc [*-i*] [*subcommand*]

This command controls the angle/distance cursor. The *adc* command with no arguments toggles the display of the angle/distance cursor (ADC). The *-i* option, if specified, causes the given value(s) to be treated as increments. Note that the *-i* option is ignored when getting values or when used with subcommands where this option makes no sense. You can also control the position, angles, and radius of the ADC using a knob or the knob command. This command accepts the following subcommands:

vars

Returns a list of all ADC variables and their values (i.e., var = val).

draw [*0|1*]

Set or get the draw parameter.

a1 [*angle*]

Set or get angle1 in degrees.

a2 [*angle*]

Set or get angle2 in degrees.

dst [*distance*]

Set or get radius (distance) of tick in local units.

odst [*distance*]

Set or get radius (distance) of tick (+-2047).

hv [*position*]

Set or get position (grid coordinates and local units).

xyz [*position*]

Set or get position (model coordinates and local units).

x [*xpos*]

Set or get horizontal position (+-2047).

y [*ypos*]

Set or get vertical position (+-2047).

dh *distance*

Add to horizontal position (grid coordinates and local units).

dv *distance*

Add to vertical position (grid coordinates and local units).

dx *distance*

Add to *x* position (model coordinates and local units).

dy *distance*

Add to *y* position (model coordinates and local units).

dz *distance*

Add to *z* position (model coordinates and local units).

anchor_pos [*0*|*1*]

Anchor ADC to current position in model coordinates.

anchor_a1 [*0*|*1*]

Anchor angle1 to go through anchorpoint_a1.

anchor_a2 [*0*|*1*]

Anchor angle2 to go through anchorpoint_a2.

anchor_dst [*0*|*1*]

Anchor tick distance to go through anchorpoint_dst.

anchorpoint_a1 [*x y z*]

Set or get anchor point for angle1 (model coordinates and local units).

anchorpoint_a2 [*x y z*]

Set or get anchor point for angle2 (model coordinates and local units).

anchorpoint_dst [*x y z*]

Set or get anchor point for tick distance (model coordinates and local units).

reset

Reset all values to their defaults.

help

Print the help message.

Examples:

`mged` **adc**
-- Toggle display of the angle/distance cursor.

`mged` **adc a1 37.5**
-- Set angle1 to 37.5˚.

`mged` **adc a1**
37.5
-- Get angle1.

`mged` **adc xyz 100 0 0**
-- Move ADC position to (100, 0, 0), model coordinates and local units.

---

ae [*-i*] *azimuth elevation* [*twist*]

Set the view orientation for the *mged* display by rotating the eye position about the center of the viewing cube. The eye position is determined by the supplied azimuth and elevation angles (degrees). The *azimuth* angle is measured in the *xy* plane with the positive *x* direction corresponding to an azimuth of 0˚. Positive azimuth angles are measured counter-clockwise about the positive *z* axis. Elevation angles are

measured from the *xy* plane with +90˚ corresponding to the positive *z* direction and -90 corresponding to the negative *z* direction. If an optional *twist* angle is included, the view will be rotated about the viewing direction by the specified *twist* angle. The *-i* option results in the angles supplied being interpreted as increments.

Examples:

`mged` **ae -90 90**

-- View from top direction.

`mged` **ae 270 0**
-- View from right hand side.

`mged` **ae 35 25 10**
-- View from azimuth 35, elevation 25, with view rotated by 10˚.

`mged` **ae -i 0 0 5**
-- Rotate the current view through 5˚ about the viewing direction.

---

analyze [*arb_name*]

The "analyze" command displays the rotation and fallback angles, surface area, and plane equation for each face of the ARB specified on the command line. The total surface area and volume and the length of each edge are also displayed. If executed while editing an *ARB,* the *arb_name* may be omitted, and the *ARB* being edited will be analyzed.

Examples:

`mged` **analyze arb_name**

-- Analyze the *ARB* named *arb_name.*

---

animmate

The "animmate" command starts the Tcl/Tk-based animation tool. The capabilities and correct use of this command are too extensive to be described here, but there is a tutorial available.

---

apropos *keyword*

The "apropos" command searches through the one-line usage messages for each *mged* command and displays the name of each command where a match is found.

Examples:

`mged` **apropos region**

-- List all commands that contain the word "region" in their one-line usage messages.

---

aproposdevel *keyword*

The "aproposdevel" command searches through the one-line usage messages for each *mged developer* command and displays the name of each command where a match is found.

Examples:

`mged` **aproposdevel region**

-- List all *developer* commands that contain the word "region" in their one-line usage messages.

---

aproposlib *keyword*

The "aproposlib" command searches through the one-line usage messages for each *BRL-CAD* library interface command and displays the name of each command where a match is found.

Examples:

`mged` **aproposlib mat**

-- List all commands that contain the word "mat" in their one-line usage messages.

---

arb *arb_name rotation fallback*

The "arb" command creates a new ARB shape with the specified *arb_name*. The new *ARB* will be 20 inches by 20 inches and 2 inches thick. The square faces will be perpendicular to the direction defined by the rotation and fallback angles. This direction can be determined by interpreting the rotation angle as an azimuth and the fallback angle as an elevation as in the ae command.

Examples:

`mged` **arb new_arb 35 25**

-- Create *new_arb* with a rotation angle of 35˚ and a fallback angle of 25˚.

`mged` **ae 35 25**
-- Rotate view to look straight on at square face of *new_arb*.

---

arced *comb/memb anim_command*

The objects in a *BRL-CAD* model are stored as Boolean combinations of primitive shapes and/or other combinations. These combinations are stored as Boolean trees, with each leaf of the tree including a corresponding transformation matrix. The "arced" command provides a means for directly editing these matrices. The first argument to the "arced" command must identify the combination and which member's matrix is to be edited. The *comb/memb* argument indicates that member *memb* of combination *comb* has the matrix to be edited. The remainder of the "arced" command line consists of an *animation* command to be applied to that matrix. The available animation commands are:

- *matrix rarc <xlate|rot> matrix elements*
  -- Replace the members matrix with the given matrix.
- *matrix lmul <xlate|ro> matrix elements* .
  -- Left multiply the members matrix with the given matrix.
- *matrix rmul <xlate|rot> matrix elements.*
  -- Right multiply the members matrix with the given matrix.

Examples:

`mged` **arced body/head matrix rot 0 0 45**

-- Rotate member *head* (in combination *body*) about the *z* axis through a 45˚ angle. By default, the *matrix* commands expect a list of 16 matrix elements to define a matrix. The *xlate* option may be used along with three translation distances in the *x*, *y*, and *z* directions (in mm) as a shorthand notation for a matrix that is pure translation. Similarly, the *rot* option along with rotation angles (degrees) about the *x*, *y*, and *z* axes may be used as shorthand for a matrix that is pure rotation.

---

area [*tolerance*]

The "area" command calculates an approximate presented area of one region in the *mged* display. For this command to work properly, a single *BRL-CAD* region must be displayed using the E command. The *tolerance* is the distance required between two vertices in order for them to be recognized as distinct vertices. This calculation considers only the minimum bounding polygon of the region and ignores holes.

Examples:

`mged` **Z**

-- Clear the *mged* display(s).

`mged` **E region_1**
-- *E* a single region.

`mged` **area**
-- Calculate the presented area of the enclosing polygon of the region.

---

arot *x y z angle*

The "arot" command performs a rotation about the specified axis (*x y z*) using screen units (-2048 to +2048). The amount of rotation is determined by *angle,* which is in degrees. Exactly what is rotated and how it is rotated are dependent on MGED's state as well as the state of the display manager. For example, in normal viewing mode, this command simply rotates the view. However, in solid edit mode, it rotates the shape being edited.

Examples:

`mged` **arot 0 0 1 10**

-- Rotate 10˚ about z axis.

---

attach [-*d display_string*] [-*i init_script*] [-*n name*] [-*t is_toplevel*] [-*W width*] [-*N height*] [-*S square_size*] win_type

The "attach" command is used to open a display window. The set of supported window types includes X and ogl. It should be noted that *attach* no longer releases previously attached display windows (i.e., multiple attaches are supported). To destroy a display window, use the release command.

Examples:

`mged` **attach ogl**

-- Open an ogl display window named .dm_ogl0 (assuming this is the first ogl display window opened using the default naming scheme).

`mged` **attach ogl**
-- Open a ogl display window named .dm_ogl1.

`mged` **attach -n myOgl -W 720 -N 486 ogl**
-- Open a 720x486 OpenGL display window named myOgl.

`mged` **attach -n myX -d remote_host:0 -i myInit X**
-- Open an X display window named myX on remote_host that is initialized by myInit.
-- myInit might contain user specified bindings like those found in the default bindings.

`mged` **toplevel .t**
-- Create a toplevel window named .t.

`mged` **attach -t 0 -S 800 -n .t.ogl ogl**
-- Open a 800x800 OpenGl display window named .t.ogl that is not a top-level window.

`mged` **button .t.dismiss -text Dismiss -command "release .t.ogl; destroy .t"**
-- Create a button to dismiss the display manager etc.

`mged` **pack .t.ogl -expand 1 -fill both**
-- Pack the display manager inside .t.

`mged` **pack .t.dismiss**
-- Pack the Dismiss button inside .t.

`mged` **attach**
-- List the help message that includes the valid display types.

---

bev [-*t*] [-*P#*] *new_obj Boolean_formula*

The "bev" command performs the operations indicated in the *Boolean_formula* and stores the result in *new_obj*. The *new_obj* will be stored as an NMG shape (it may be converted to a polysolid by using the nmg_simplify command). If the *-t* option is specified, then the resulting object will consist entirely of triangular facets. The default is to allow facets of any complexity, including holes. The *-P* option specifies the number of CPU's to use for the calculation (however, this is currently ignored). Only simple *Boolean_formulas* are allowed. No parentheses are allowed and the operations are performed from left to right with no precedence. More complex expressions must be expressed as *BRL-CAD* objects using the r, g, or c commands and evaluated using the facetize or ev commands.

Examples:

`mged` **bev -t triangulated_lens sphere1 + sphere2**

-- Create a triangulated object by intersecting objects *sphere1* and *sphere2*.

---

c [*-g*|*r*] *combination_name* [*Boolean_expression*]

The "c" command creates a *BRL-CAD* combination with the name *combination_name*. The *-g* option is the default and indicates that the combination should be flagged as a *BRL-CAD* group, and the *-r* option indicates that the combination is a *BRL-CAD* region. The *Boolean_expression* allows parentheses. Where no order is specified, intersections are performed before subtractions or unions; then subtractions and unions are performed, left to right. Where there is no *Boolean_expression* and *combination_name* does not already exist, a new empty combination will be created. If no *Boolean_expression* is provided, and *combination_name* does already exist and one of *-g* or *-r* is specified, then *combination_name* is flagged to agree with the indicated option. If a new *region* is created with this command, its region specific attributes will be set according to the current defaults (see regdef). The comb and r commands may also be used to create combinations.

Examples:

`mged` **c -g abc (a u b) - (a + d)**

-- Create a combination named *abc* according to the formula *(a u b) - (a + d)*.

---

cat <*objects*>

The "cat" command displays a brief description of each item in the list of *objects*. If the item is a primitive shape, the type of shape and its vertex are displayed. If the item is a combination, the Boolean formula for that combination—including operands, operators, and parentheses—is displayed. If the combination is flagged as a region, then that fact is also displayed along with the region's ident code, air code, los, and GIFT material code.

Examples:

`mged` **cat region_1 region_2**

-- Display the Boolean formulas for some regions.

---

center *[x y z]*

The "center" command positions the center of the *mged* viewing cube at the specified model coordinates. This is accomplished by moving the eye position while not changing the viewing direction. (The lookat command performs a related function by changing the viewing direction, but not moving the eye location.) The coordinates are expected in the current editing units. In case the coordinates are the result of evaluating a formula, they are echoed back. If no coordinates are provided, the current center coordinates (in current editing units, not mm) are printed and can be used in subsequent calculations.

It is often convenient to use the center of the view when visually selecting key locations in the model for construction or animation because of (1) the visible centering dot on the screen, (2) the fact that zoom and rotation are performed with respect to the view center, (3) the default center-mouse behavior is to move the indicated point to the view center, and (4) the angle/distance cursors are centered by default. This command provides the means to set and retrieve those values numerically.

Examples:

`mged` **center**

-- Print out the coordinates of the center of the *mged* display.

`mged` **center 12.5 5.6 8.7**

-- Move the center of the *mged* display to the point (12.5, 5.6, 8.7).

`mged` **set oldcent [center]**
-- Set the Tcl variable $oldcent to the display center coordinates.

`mged` **set glob_compat_mode 0**

`mged` **units mm**

`mged` **eval center [vadd2 [center] {2 0 0}]**
-- Move the center point two mm in the model + *x* direction.

`mged` **units mm**

`mged` **db adjust sphere.s V [center]**

`mged` **redraw_vlist sphere.s**
-- Update the "V" vertex of shape sphere.s in the database to be located at the current center of the view, and recreate the vector display lists of only those object(s) that were affected by the change.

---

color *low high r g b str*

The "color" command creates an entry in the database that functions as part of a color lookup table for displayed regions. The ident number for the region is used to find the appropriate color from the lookup table. The *low* and *high* values are the limits of region ident numbers to have the indicated *rgb* color (0-255) applied. The *str* parameter is intended to be an identifying character string, but is currently ignored. The current list of color table entries may be displayed with the prcolor command,

and the entire color table may be edited using the edcolor command. If a color lookup table exists, its entries will override any color assigned using the mater command.

Examples:

`mged` **color 1100 1200 255 0 0 fake_string**

-- Make an entry in the color lookup table for regions with idents from 1100 to 1200 using the color red.

---

comb *combination_name <operation object>*

The "comb" command creates a new combination or extends an existing one. If *combination_name* does not already exist, then it will be created using the indicated list of *operations* and *objects*. If it does exist, the list of *operations* and *objects* will be appended to the end of the existing combination. The *<operation object>* list is expected to be in the same form as used in the r command. The c command may also be used to create a *combination*.

Examples:

`mged` **comb abc u a - b + c**

-- Create combination abc as ((a - b) + c).

---

comb_color *combination_name r g b*

The "comb_color" command assigns the color *rgb* (0-255) to the existing combination named *combination_name*.

Examples:

`mged` **comb_color region1 0 255 0**

-- Assign the color green to *region1*.

---

copyeval *new_primitive path_to_old_ primitive*

Objects in a *BRL-CAD* model are stored as Boolean trees (combinations), with the members being primitive shapes or other Boolean trees. Each member has a transformation matrix associated with it. This arrangement allows a primitive to be a member of a combination, and that combination may be a member of another combination, and so on. When a combination is displayed, the transformation matrices are applied to its members and passed down through the combinations to the leaf (primitive shape) level. The accumulated transformation matrix is then applied

to the primitive before it is drawn on the screen. The "copyeval" command creates a new primitive object called *new_primitive* by applying the transformation matrices accumulated along the *path_to_old_primitive* to the leaf primitive shape object at the end of the path and saving the result under the name *new_primitive*. The *path_to_old_primitive* must be a legitimate path ending with a primitive shape.

Examples:

`mged` **copyeval shapeb comb1/comb2/comb3/shapea**

-- Create *shapeb* from *shapea* by applying the accumulated transformation matrices from the path comb1/comb2/comb3.

---

copymat *comb1/memb1 comb2/memb2*

The "copymat" command copies the transformation matrix from a member of one combination to the member of another.

Examples:

`mged` **copymat comb1/memb1 comb2/memb2**

-- Set the matrix for member *memb2* in combination *comb2* equal to the matrix for member *memb1* in combination *comb1*.

---

cp *from_object to_object*

The "cp" command makes a duplicate of an object (shape or combination). If *from_object* is a shape, then it is simply copied to a new shape named *to_object*. If *from_object* is a combination, then a new combination is created that contains exactly the same members, transformation matrices, etc, and it is named *to_object*.

Examples:

`mged` **cp comb1 comb2**

-- Make a duplicate of combination *comb1* and call it *comb2*.

---

cpi *old_tgc new_tgc*

The "cpi" command copies *old_tgc* (an existing TGC shape) to a new TGC shape (*new_tgc*), positions the new TGC such that its base vertex is coincident with the center of the top of *old_tgc*, and puts *mged* into the solid edit state with *new_tgc* selected for editing. This command was typically used in creating models of wiring or piping runs; however, a new pipe solid has since been added to *BRL-CAD* to handle such requirements.

Examples:

`mged` **cpi tgc_a tgc_b**

-- Copy *tgc_a* to *tgc_b* and translate *tgc_b* to the end of *tgc_a*.

---

d <*objects*>

The "d" command deletes the specified list of objects from the *mged* display. This is a synonym for the erase command. Only objects that have been explicitly displayed may be deleted with the "d" command (use the who command to see a list of explicitly displayed objects). Objects that are displayed as members of explicitly displayed combinations cannot be deleted from the display with this command (see dall or erase_all). Note that this has no effect on the *BRL-CAD* database itself. To actually remove objects from the database, use the kill command.

Examples:

`mged` **d region1 shapea**

-- Delete *region1* and *shapea* from the *mged* display.

---

dall <*objects*>

The "dall" command deletes the specified list of objects from the *mged* display. This is a synonym for the erase_all command. This command will allow the user to delete objects that have not been explicitly displayed (unlike the d command). Note that this has no effect on the *BRL-CAD* database itself. To actually remove objects from the database, use the kill command.

Examples:

`mged` **dall region1 shapea**

-- Delete *region1* and *shapea* from the *mged* display.

---

db *command* [*args...*]

The "db" command provides an interface to a number of database manipulation routines. Note that this command always operates in units of millimeters. The *command* must be one of the following with appropriate arguments:

- *match* <*regular_exp*>
  -- Return a list of all objects in that database that match the list of regular expressions.
- *get shape_or_path [attribute]*
  -- Return information about the primitive shape at the end of the *shape_or_path*.

If a path is specified, the transformation matrices encountered along that path will be accumulated and applied to the leaf shape before displaying the information. If no *attribute* is specified, all the details about the shape are returned. If a specific *attribute* is listed, then only that information is returned.

- *put shape_name shape_type attributes*
  -- Create shape named *shape_name* of type *shape_type* with attributes as listed in *attributes*. The arguments to the *put* command are the same as those returned by the *get* command.
- *adjust shape_name attribute new_value1 [new_value2 new_value3...]*
  -- Modify the shape named *shape_name* by adjusting the value of its *attribute* to the *new_values*.
- *form object_type*
  -- Display the format used to display objects of type *object_type*.
- *tops*
  -- Return all top-level objects.
- *close*
  -- Close the previously opened database and delete the associated command.

Examples:

`mged` **db match *.s**

-- Get a list of all objects in the database that end with ".s".

`mged` **db get cone.s**
-- Get a list of all the attributes and their values for shape *cone.s*.

`mged` **db get cone.s V**
-- Get the value of the *V* (vertex) attribute of shape *cone.s*.

`mged` **db put new_cone.s tgc V {0 0 0} H {0 0 1} A {1 0 0} B {0 1 0} C {5 0 0} D {0 5 0}**
-- Create a new TGC shape named *new_cone.s* with the specified attributes.

`mged` **db adjust new_cone.s V {0 0 10}**
-- Adjust the *V* (vertex) attribute of *new_cone.s* to the value {0 0 10}.

mged **db form tgc**
-- Display the format used by the *get* and *put* commands for the TGC shape type.

---

db_glob *cmd_string*

Globs *cmd_string* against the MGED database resulting in an expanded command string.

Examples:

`mged` **db_glob "l r23\[0-9\]"**

l r230 r231 r232 r233 r234 r235 r236 r237 r238 r239
-- Returns a command string to list objects r230 through r239.

---

dbconcat *database_file* [*prefix*]

The "dbconcat" command concatenates an existing *BRL-CAD* database to the database currently being edited. If a *prefix* is supplied, then all objects from the *database_file* will have *prefix* added to the beginning of their names. Note that each *BRL-CAD* object must have a unique name, so care must be taken not to "dbconcat" a database that has objects with names the same as objects in the current database. The dup command may be used to check for duplicate names. If the *dup* command finds duplicate names, use the *prefix* option to both the *dup* and *dbconcat* commands to find a *prefix* that produces no duplicates. If duplicate names are encountered during the "dbconcat" process, computer-generated prefixes will be added to the object names coming from the *database_file* (but member names appearing in combinations will not be modified, so this is a dangerous practice and should be avoided).

Examples:

`mged` **dbconcat model_two.g two_**

-- Copy all the objects in *model_two.g* to the current database, but prefix the name of every object copied with the string *two_*.

---

debugbu [*hex_code*]

The "debugbu" command allows the user to set or check the debug flags used by *libbu*. With no arguments, the *debugbu* command displays all the possible settings for the *bu_debug* flag and the current value. When a *hex_code* is supplied, that value is used as the new value for *bu_debug*. Similar debug commands for other *BRL-CAD* libraries are debuglib for *librt* and debugnmg for the NMG portion of *librt*. Other debugging commands include debugmem and debugdir.

Examples:

`mged` **debugbu**

-- Get a list of available *bu_debug* values and the current value.

`mged` **debugbu 2**
-- Set *bu_debug* to <MEM_CHECK>.

---

debugdir

The "debugdir" command displays a dump of the in-memory directory for the current database file. The information listed for each directory entry includes:

- memory address of the directory structure.
- name of the object.
- "d_addr" for objects on disk, or "ptr" for objects in memory.
- "SOL," "REG," or "COM" if the object is a shape, region, or combination, respectively.
- file offset (for objects on disk) or memory pointer (for objects in memory).
- number of instances referencing this object (not normally filled in).
- number of database granules used by this object.
- number of times this object is used as a member in combinations (not normally filled in).

Examples:

`mged` **debugdir**

-- Get a dump of the in-memory directory.

---

debuglib [*hex_code*]

The "debuglib" command allows the user to set or check the debug flags used by *librt*. With no arguments, the *debuglib* command displays all the possible settings for the *librt* debug flag and the current value. When a *hex_code* is supplied, that value is used as the new value for the flag. Similar debug commands for other *BRL-CAD* libraries are debugbu for *libbu* and debugnmg for the NMG portion of *librt*. Other debugging commands include debugmem and debugdir.

Examples:

`mged` **debuglib**

-- Get a list of available debug values for *librt* and the current value.

`mged` **debuglib 1**
-- Set the *librt* debug flag to <DEBUG_ALLRAYS> (print info about rays).

---

debugmem

The "debugmem" command prints a list of all the memory blocks that have been allocated and recorded in the *memdebug* table. Memory allocation is not normally recorded in the *memdebug* table, but executing the *debugbu 2* command will turn on the <MEM_CHECK> flag, and as long as that flag is set, all memory allocation will be recorded.

Examples:

`mged` **debugmem**

-- Get a list of allocated memory blocks.

---

debugnmg [*hex_code*]

The "debugnmg" command with no options displays a list of all possible debug flags available for NMG processing. If the command is invoked with a hex number argument, that value is used as the new value for the *NMG* debug flag. Similar debug commands for other *BRL-CAD* libraries are debuglib for *librt* and debugbu for *libbu*. Other debugging commands include debugmem and debugdir.

Examples:

`mged` **debugnmg 100**

-- Set the *NMG* debug flag to get details on the classification process.

---

decompose *NMG_shape* [*prefix*]

The "decompose" command processes an NMG shape and produces a series of new *NMG* shapes consisting of each maximally connected shell in the original *NMG* shape. If an optional prefix is supplied, the resulting *NMG* shapes will be named by using the prefix and adding an underscore character and a number to make the name unique. If no prefix is supplied, the default prefix "sh" will be used.

Examples:

`mged` **decompose shape.nmg part**

-- Decompose the *NMG* shape named *shape.nmg* into maximally connected shells and put each resulting shell into a separate *NMG* shape named *part_1*, *part_2*, ...

---

delay *seconds microseconds*

The "delay" command provides a delay of the specified time before the next command will be processed.

Examples:

`mged` **delay 5 0**

-- Delay for 5 seconds.

---

dm *subcommand* [*args*]

The "dm" command provides a means to interact with the display manager at a lower level. The *dm* command accepts the following subcommands:

set [*var* [*val*]]

The "set" subcommand provides a means to set or query display manager-specific variables. Invoked without any arguments, the *set* subcommand will return a list of all available internal display manager variables. If only the *var* argument is specified, the value of that variable is returned. If both *var* and *val* are given, then *var* will be set to *val*.

size [*width height*]

The "size" subcommand provides a means to set or query the window size. If no arguments are given, the display manager's window size is returned. If *width* and *height* are specified, the display manager makes a request to have its window resized. Note that a size request is just that, a request, so it may be ignored, especially if the user has resized the window using the mouse.

m *button x y*

The "m" subcommand is used to simulate an M command. The *button* argument determines which mouse button is being used to trigger a call to this command. This value is used in the event handler to effect dragging the faceplate scrollbars. The *x* and *y* arguments are in X screen coordinates, which are converted to MGED screen coordinates before being passed to the M command.

am *<r | t | s> x y*

The "am" subcommand effects *mged*'s alternate mouse mode. The alternate mouse mode gives the user a different way of manipulating the view or an object. For example, the user can drag an object or perhaps rotate the view while using the mouse. The first argument indicates the type of operation to perform (i.e., *r* for rotation, *t* for translation, and *s* for scale). The *x* and *y* arguments are in X screen coordinates and are transformed appropriately before being passed to the knob command.

adc *<1 | 2 | t | d> x y*

The "adc" subcommand provides a way of manipulating the angle distance cursor while using the mouse. The first argument indicates the type of operation to perform (i.e., *1* for angle 1, *2* for angle 2, *t* for translate, and *d* for tick distance). The *x* and *y* arguments are in X screen coordinates and are transformed appropriately before being passed to the adc command (i.e., not "dm adc").

con *<r | t | s <x | y | z> xpos ypos*

This form of the "con" subcommand provides a way to effect constrained manipulation of the view or an object while using the mouse. This simulates the behavior of sliders without taking up screen real estate. The first argument indicates the type of operation to perform (i.e., *r* for rotation, *t* for translation, and *s* for scale). The <*x* | *y* | *z*> argument is the axis of rotation, translation, or scale. The *xpos* and *ypos* arguments are in X screen coordinates and are transformed appropriately before being passed to the knob command.

con *a* <*x* | *y* | *1* | *2* | *d*> *xpos ypos*

This form of the "con" subcommand provides a way to effect constrained manipulation of the angle distance cursor while using the mouse. This simulates the behavior of sliders without taking up screen real estate. The first argument indicates that this is to be applied to the angle distance cursor. The next argument indicates the type of operation to perform (i.e., *x* for translate in the *x* direction, *y* for translate in the *y* direction, *1* for angle 1, *2* for angle 2, and *d* for tick distance). The *xpos* and *ypos* arguments are in *x* screen coordinates and are transformed appropriately before being passed to the knob command.

Examples:

`mged` **dm set**

-- Get a list of the available display manager internal variables.

`mged` **dm set perspective 1**
-- Turn on perspective projection in the display.

`mged` **dm size**
-- Return the size to the display manager.

`mged` **dm size 900 900**
-- Request that the display manager window be resized to 900x900.

`mged` **dm m 2 100 200**
-- Simulate a button2 press at (100, 200) in X screen coordinates.

`mged` **dm am r 400 100**
-- Start an alternate mouse mode rotation.

`mged` **dm adc d 300 200**
-- Start a tick distance manipulation.

`mged` **dm con t z 200 200**
-- Start a constrained translation down the Z axis.

`mged` **dm con a d 200 100**
-- Start a constrained tick distance manipulation.

`mged` **dm idle**
-- End drag.

---

draw [-*C#/#/#*] [-r] [-s] <*objects*>

Add <*objects*> to the display list so that they will appear on the *MGED* display. The e command is a synonym for *draw*.

- The -*C* option provides the user a way to specify a color that overrides all other color specifications including combination colors and region-id-based colors.
- The -*r* option specifies that all objects are to be drawn in red, which can be useful when examining objects colored black.
- The -*s* option specifies that subtracted and intersected objects should be drawn with shape lines rather than dot-dash lines.

Examples:

`mged` **draw object1 object2**

-- Draw *object1* and *object2* in the *MGED* display.

`mged` **draw –C 255/255/255 object2**
-- Draw *object2* in white.

---

dup *file* [*prefix*]

The "dup" command checks the specified *file* (which is expected to contain a *BRL-CAD* model) for names that are the same as those in the current model. If a *prefix* is included on the command line, all names in the specified *file* will have that *prefix* added to their names before comparison with the current model. This command is often used prior to invoking the dbconcat command to ensure that there are no name clashes.

Examples:

`mged` **dup other_model.g**

-- Check *other_model.g* for names duplicating those in the current model.

`mged` **dup other_model.g abc**
-- Do the same check as above, but prefix all the names in *other_model.g* with *abc* before comparing with the names in the current model.

---

e [-*C#/#/#*] <*objects*>

The "e" command adds the objects in the argument list to the display list so that they will appear on the *MGED* display. This is a synonym for the draw command; see that entry for a full list of options. The -*C* option provides the user a way to specify a color that overrides all other color specifications including combination colors and region-id-based colors.

Examples:

`mged` **e object1 object2**

-- Draw *object1* and *object2* in the *MGED* display.

---

eac <*aircodes*>

The "eac" command adds all the regions in the current model that have one of the aircodes in the argument list to the display list so that they will appear on the *MGED* display. Regions that have nonzero ident numbers will not be listed by this command. The whichair command will perform the same search, but just lists the results.

Examples:

`mged` **eac 1 2 3**

-- Draw all regions with *aircodes* 1, 2, or 3 in the *MGED* display.

---

echo *text*

The "echo" command merely echos whatever text is provided as an argument on the command line. This is intended for use in *MGED* scripts.

Examples:

`mged` **echo some text goes here**

-- Display the text, "some text goes here."

---

edcodes <*objects*>

The "edcodes" command puts the user into an editor to edit a file that has been filled with the ident, air code, material code, LOS, and name of all the regions in the specified objects. The user may then modify the entries (except for the names). The editor used is whatever the user has set in the environment variable *EDITOR*. If *EDITOR* is not set, then */bin/ed* is used.

Examples:

`mged` **edcodes object1 object2**

-- Edit the region codes for all regions below object1 and object2.

---

edcolor

The "edcolor" command puts the user into an editor to edit a file that has been filled with the ident based color lookup table. The entire table may be seen with the prcolor command, and entries may be added using the color command. The editor used is whatever the user has set in the environment variable *EDITOR*. If *EDITOR* is not set, then */bin/ed* is used.

Examples:

`mged` **edcolor**

-- Edit the color table.

---

edcomb *combname R|G regionid air_code los* [*material_code*]

The "edcomb" command allows the user to modify the attributes of a combination. The *combname* is the name of the combination to be modified. An *R* flag indicates that the region flag should be set; otherwise, the region flag is unset. If the region flag is not being set, then the remainder of the attributes are ignored. If the region flag is being set, then the region_id, aircode, los, and material_code are set according to the arguments supplied.

Examples:

`mged` **edcomb comb1 R 1001 0 50 8**

-- Make *comb1* a *region* and set its *ident* to 1001, its *air code* to 0, its *LOS* to 50, and its *material code* to 8.

`mged` **edcomb comb1 0 0 0 0**
-- Unset the *region* flag for combination *comb1*.

---

edgedir [*x y z*]|[*rot fb*]

The "edgedir" command allows the user to set the direction of an edge by specifying a direction vector in the form of *x, y,* and *z* components or via rotation and fallback angles. This can only be done while moving an edge of an ARB.

Examples:

`mged` **edgedir 0 1 0**

-- Rotate the edge being edited to be parallel to the *y* axis.

---

edmater <*combinations*>

The "edmater" command places the user in an editor ready to edit a file filled with shader arguments for the combinations listed on the command line. The arguments placed in the file for editing are the *shader* name and its own arguments, RGB color, *RGB_valid* flag, and the inheritance flag. The editor used is whatever the user has set in the environment variable *EDITOR*. If *EDITOR* is not set, then */bin/ed* is used.

Examples:

`mged` **edmater comb1 comb2**

-- Edit the *shader* parameters for combinations named *comb1* and *comb2*.

---

eqn *A B C*

The "eqn" command allows the user to rotate the face of an ARB shape by providing the coefficients of an equation of the desired plane for the face. The coefficients *A, B*, and *C* are from the plane equation:

```
Ax + By + Cz = D
```

The user must be editing an *ARB* shape and be rotating a face of the *ARB* for this command to have any effect. When entering such a state, the user will be asked which of the face vertices should be held constant, and from this information the *D* coefficient of the equation is determined.

Examples:

`mged` **eqn 0 0 1**

-- Rotate the face of the *ARB* being edited to be parallel to the *xy* plane.

---

erase <*objects*>

The "erase" command deletes the specified list of objects from the MGED display. This is a synonym for the d command. Only objects that have been explicitly displayed may be deleted with the "erase" command (use the who command to see a list of explicitly displayed objects). Objects that are displayed as members of explicitly displayed combinations cannot be deleted from the display with this command (see dall or erase_all). Note that this has no effect on the *BRL-CAD* database itself. To actually remove objects from the database, use the kill command.

Examples:

`mged` **erase region1 shapea**

-- Delete *region1* and *shapea* from the MGED display.

---

erase_all <*objects*>

The "erase_all" command deletes the specified list of objects from the MGED display. This is a synonym for the dall command. This command will allow the user to delete objects that have not been explicitly displayed (unlike the d or erase commands). Note that this has no effect on the *BRL-CAD* database itself. To actually remove objects from the database, use the kill command.

Examples:

`mged` **erase_all region1 shapea**

-- Delete *region1* and *shapea* from the MGED display.

---

ev [-*dfnrstuvwST*] [-*P#*] [-C#/#/#] <*objects*>

The "ev" command evaluates the *objects* specified by tessellating all primitive shapes in the objects and then performing any Boolean operations specified in the *objects*. The result is then displayed in the MGED display according to the specified options:

- *d* -- Do not perform Boolean operations or any checking; simply convert shapes to polygons and draw them. Useful for visualizing BoT and polysolid primitives.
- *f* -- Fast path for quickly visualizing polysolid primitives.
- *w* -- Draw wireframes (rather than polygons).
- *n* -- Draw surface normals as little "hairs."
- *s* -- Draw shape lines only (no dot-dash for subtract and intersect).
- *t* -- Perform CSG-to-tNURBS conversion (still under development).
- *v* -- Shade using per-vertex normals, when present.
- *u* -- Draw NMG edgeuses (for debugging).
- *S* -- Draw tNURBs with trimming curves only, no surfaces.
- *T* -- Do not triangulate after evaluating the Boolean (may produce unexpected results if not used with the *w* option).
- *P#* -- Use # processors in parallel. Default=1.
- *r* -- Draw all objects in red. Useful for examining objects colored black.
- *C#/#/#* -- Draw all objects in in the specified rgb color.

Examples:

`mged` **ev region1 shapea**

-- Display evaluated *region1* and *shapea* as shaded polygons.

`mged` **ev -wT region1**
-- Display evaluated *region1* as wireframe without triangulating.

---

exit

The "exit" command ends the MGED process. This is a synonym for the quit command.

Examples:

`mged` **exit**

-- Stop MGED.

---

expand *regular_expression*

The "expand" command performs matching of the *regular_expression* with the names of all the objects in the database. It returns all those that successfully match.

Examples:

`mged` **expand *.r**

-- Display a list of all database object names that end in ".r".

---

extrude #### *distance*

The "extrude" command modifies an ARB shape by extruding the specified face through the specified *distance* to determine the position of the opposing face. The face to be extruded is identified by listing its vertex numbers as they are labeled in the MGED display when the *ARB* is edited. Note that the face identified is not moved, but the opposite face is adjusted so that it is the specified *distance* from the specified face. The order that the vertex numbers are listed determines the direction of the extrusion using the right-hand rule.

Examples:

`mged` **extrude 1234 5**

-- Move face 5678 so that it is 5 units from face 1234.

---

eye_pt *x y z*

The "eye_pt" command positions the *eye point* to the given *x, y*, and *z* coordinates (specified in mm).

Examples:

`mged` **eye_pt 100 0 0**

-- Position the eye at 100 mm along the *x* axis.

---

e_muves *MUVES_component1 MUVES_component2 ...*

The "e_muves" command displays the *BRL-CAD* regions that are part of the indicated MUVES components. The internal list of MUVES components must have been created earlier by the read_muves command. The MUVES components listed on the command line must not use any wildcards (the expansion will result in *BRL-CAD* objects, not MUVES components).

Examples:

`mged` **e_muves fuel transmission**

-- Display the *BRL-CAD* regions that make up the MUVES components named "fuel" and "transmission."

---

facedef #### [*a*|*b*|*c*|*d parameters*]

The "facedef" command allows the user to redefine any face of an ARB8 shape. The user must be in Solid Edit Mode with an *ARB* selected for editing. The optional parameters may be omitted, and MGED will prompt for the missing values. The options are:

- *a*
  -- Specify the new location of this face by providing coefficients for its plane equation:
  `Ax + By + Cz = D.`
- *b*
  -- Specify the new location of this face using three points.
- *c*
  -- Specify the new location of this face using rotation and fallback angles.
- *d*
  -- Specify the new location of this face by changing the *D* value in the plane equation.
- *q*
  -- Return to MGED prompt.

Examples:

`mged` **facedef 1234 a 1 0 0 20**

-- Move face 1234 such that it is in the *yz* plane at *x*=20.

`mged` **facedef 5678 b 0 0 10 10 0 10 10 10 10**
-- Move face 5678 such that it is in the plane formed by the three points (0 0 10), (10 0 10), and (10 10 10).

---

facetize [-*tT*] [-*P#*] *new_object old_object*

The "facetize" command creates *new_object* as an NMG shape by tessellating all the primitive shapes in *old_object* and then performing any Boolean operations specified in *old_object*. The -*T* option indicates that all faces in the *new_object* should be triangulated. The -*t* option is to create TNURB faces rather than planar approximations (this option is still under development). The -*P* option is intended to allow the user to specifiy the number of CPU's to use for this command, but it is currently ignored.

Examples:

`mged` **facetize region1.nmg region1.r**

-- Create a facetized *NMG* version of existing object *region1.r*.

---

find <*objects*>

The "find" command displays all combinations that have any of the *objects* specified as a member.

Examples:

`mged` **find shapea**

-- List all *combinations* that refer to *shapea*.

---

fracture *NMG_shape* [*prefix*]

The "fracture" command creates a new NMG shape for every "face" in the specified *NMG_shape*. The new shapes will be named by adding an underscore and a number to the *prefix*. If no *prefix* is specified, then the *NMG_shape* name provided is used in place of the *prefix*.

Examples:

`mged` **fracture shape1.nmg f**

-- Create a series of *NMG* shapes named "f_#", one for each face in *shape1.nmg*.

---

g *groupname* <*objects*>

The "g" command creates a special type of combination often referred to as a group. This builds a *combination* by unioning together all the listed *objects*. If *groupname* already exists, then the list of *objects* will be unioned to the end of it. (Note that an existing *groupname* is not restricted to being a *group*; any *combination* is legal.) Other commands to build *combinations* are c, r, or comb.

Examples:

`mged` **g shape1.nmg f**

-- Create or extend shape1.nmg by unioning in *f*.

---

garbage_collect

The "garbage_collect" command eliminates unused space in a BRL-CAD database file.

Examples:

`mged` **garbage_collect**

-- Clean out unused space in the database.

---

gui [*-config b|c|g*] [*-d display_string*] [*-gd graphics_display_string*] [*-dt graphics_type*] [*-id name*] [*-c -h -j -s*]

This command is used to create an instance of MGED's default Tcl/Tk graphical user interface (GUI). The following options are allowed:

-config b|c|g	Configure the GUI to display the command window, the graphics window, or both. This option is useful only when the GUI is combining the text and graphics windows. See the -c option.
-d display_string	Display/draw the GUI on the screen indicated by the display_string. Note that this string format is the same as the X DISPLAY environment variable.
-gd display_string	Display/draw the graphics window on the screen indicated by the display_string. Note that this string format is the same as the X DISPLAY environment variable.
-dt graphics_type	Indicates the type of graphics windows to use. The possible choices are X and ogl (for machines that support OpenGL). Defaults to ogl, if supported; otherwise X.
-id name	Specify the id to use when referring to this instance of the GUI.
-c	Combine text window and display manager windows.
-s	Use separate text window and display manager windows. This is the default behavior.

-j	Join the collaborative session.
-h	Print the help message.

---

help [*command*]

The "help" command returns a list of available MGED commands along with a one-line usage message for each. If a command is supplied as an argument, the one-line usage message for that command is returned. The helpdevel, helplib, ?, ?devel, and ?lib commands provide additional information on available commands.

Examples:

`mged` **help ae**

-- Display a one-line usage message for the *ae* command.

---

helpdevel [*command*]

The "helpdevel" command returns a list of available *developer* commands along with a one-line usage message for each. If a command is supplied as an argument, the one-line usage message for that command is returned. The help, helplib, ?, ?devel,and ?lib commands provide additional information on available commands.

Examples:

`mged` **helpdevel winset**

-- Display a one-line usage message for the *winset* command.

---

helplib [*command*]

The "helplib" command returns a list of available *library* commands along with a one-line usage message for each. If a command is supplied as an argument, the one-line usage message for that command is returned. The help, helpdevel, ?, ?devel, and ?lib commands provide additional information on available commands.

Examples:

`mged` **helplib mat_trn**

-- Display a one-line usage message for the *mat_trn* command.

---

history [-*delays*]

The "history" command displays the list of commands executed during the current MGED session. The one exception is the hist_add command, which can add a

command to the history list without executing it. If the *-delays* option is used, then the delays between commands will also be displayed.

Examples:

`mged` **history**

-- Display the command history list.

---

i *obj_name comb_name* [*operation*]

The "i" command adds *obj_name* to the end of the combination named *comb_name*. The *operation* may be "+," "-," or "u." If no *operation* is specified, "u" is assumed. If *comb_name* does not exist, it is created.

Examples:

`mged` **i region3 group5**

-- Add *region3* to the combination *group5*.

---

idents *file_name <objects>*

The "idents" command places a summary of the regions in the list of *objects* specified in the file specified. If any regions include other regions, then only the first encountered region in that tree will be listed. The resulting file will contain two lists of regions, one in the order encountered in the list of *objects*, and the other ordered by ident number. The data written for each region includes (in this order) a sequential region count, the *ident* number, the air code, the material code, the LOS, and the path to the region.

Examples:

`mged` **idents regions_file group1 group2 region3**

-- Create a file named *regions_file* and list all the regions in *group1, group2,* and *region3* in the file.

---

ill *obj_name*

The "ill" command performs the function of selecting an object after entering *solid illuminate* or *object illuminate* mode. In *solid illuminate* mode, this command selects the specific shape for editing. In *object illuminate* mode, this command selects the leaf object for the object path, then the user may use the mouse to select where along the object path the editing should be applied. In both modes, the *ill* command will only succeed if the specified *obj_name* is only referenced once in the displayed objects; otherwise a *multiply referenced* message will be displayed. If the *ill*

command fails, the user must resort to either using the mouse to make the selection, or using aip and M 1 0 0.

Examples:

`mged` **ill shapea**

-- Select *shapea* for editing.

---

in [*-f*] [*-s*] *new_shape_name shape_type <parameters>*

The "in" command allows the user to type in the arguments needed to create a shape with the name *new_shape_name* of the type *shape_type*. The command may be invoked with no arguments, and it will prompt the user for all needed information. The *-s* option will invoke the solid edit mode on the new shape immediately after creation. The *-f* option does not draw the new shape, and therefore the *-s* option may not be used in conjunction with *-f*. The possible values for *shape_type* are:

- arb8 -- ARB (eight vertices).
- arb7 -- ARB (seven vertices).
- arb6 -- ARB (six vertices).
- arb5 -- ARB (five vertices).
- arb4 -- ARB (four vertices).
- arbn – Arbitrary polyhedron with arbitrary number of vertices (plane equations).
- bot – Bag of Triangles.
- dsp – Displacement Map.
- pipe – Pipe (run of connected pipe or wire).
- ebm --Extruded Bit Map.
- vol --Voxels.
- hf -- Height Field deprecated, see dsp.
- ars -- Arbitrary Faceted Solid.
- half -- Half Space.
- sph -- Ellipsoid (center and radius).
- ell -- Ellipsoid (center and three semi-axes).
- ellg -- Ellipsoid (foci and chord length).
- ell1 -- Ellipsoid (center, one semi-axis, and a radius of revolution).
- tor -- Torus.
- tgc -- Truncated General Cone (most general TGC).
- tec -- Truncated General Cone (top radii are scaled from base radii).
- rec -- Truncated General Cone (right elliptical cylinder).
- trc -- Truncated General Cone (truncated right circular cone).
- rcc -- Truncated General Cone (right circular cylinder).
- box -- ARB (vertex and three vectors).
- raw -- ARB (right angle wedge).
- rpp -- ARB (axis aligned rectangular parallelepiped).

- rpc -- Right Parabolic Cylinder.
- rhc -- Right Hyperbolic Cylinder.
- epa -- Elliptical Paraboloid.
- ehy -- Elliptical Hyperboloid.
- eto -- Elliptical Torus.
- part -- Particle.

Examples:

`mged` **in new1 raw 0 0 0 0 0 1 1 0 0 0 1 0**

-- Create an *ARB* named *new1* in the form of a right angle wedge.

---

inside [*outside_shape_name new_inside_shape_name <parameters>*]

The "inside" command creates a new shape that is *inside* an existing shape. This command is typically used to create an *inside* shape that can be subtracted from the original shape to produce a hollow shell. The command is typically used with no arguments, and it prompts the user for all needed information; however, all the parameters may be supplied on the command line. If MGED is in *solid edit mode* when the "inside" command is issued, then the shape currently being edited will be used as the "outside_shape." Similarly, if MGED is in *matrix edit mode* when the "inside" command is executed, then the current key shape will be used as the outside shape.

Examples:

`mged` **inside out_arb in_arb 1 1 1 1 1 1**

-- Create a shape named *in_arb* such that each face is 1 unit from the corresponding face in *out_arb.*

`mged` **inside in_arb 1 1 1 1 1 1**
-- Create a shape named *in_arb* such that each face is 1 unit from the corresponding face in the current key shape or the shape currently being edited.

---

item *region_name ident_number* [*air_code* [*material_code* [*LOS*]]]

The "item" command sets the values of ident_number, aircode, material_code, and LOS for the specified region.

Examples:

`mged` **item region_1 1137 0 8 100**

-- Set *ident number* to 1137, *air code* to 0, *material code* to 8, and *los* to 100 for *region_1*.

---

joint *command* [*options*]

articulation/animation commands (experimental)

?

This command returns a list of available joint commands.

accept [*-m*] [*joint_names*]

debug [*hex code*]

help [*commands*]

This command returns a usage message for each joint command.

holds [*names*]

list [*names*]

load *file_name*

mesh

move *joint_name p1* [*p2...p6*]

reject [*joint_names*]

save file_name

solve *constraint*

test file_name

unload

---

journal [*-d*] [*journal_file_name*]

The "journal" command starts or stops the journaling of MGED commands to a file. If executed with no arguments, the command stops journaling. If *journal_file_name* is provided, that file will become the recipient of the journaling. If a *-d* option is also provided, the journaling will include the delays between commands. Journaling is off by default.

Examples:

`mged` **journal journal_file**

-- Start journaling to *journal_file.*

---

keep *keep_file* <*objects*>

The "keep" command copies the *objects* specified to the *keep_file*. If *keep_file* does not exist, it is created. If *keep_file* does exist, the *objects* are appended to it. The *keep_file* is a *BRL-CAD* database file. The *objects* in the list must exist in the current database.

Examples:

`mged` **keep sample.g sample1 sample2**

-- Create *sample.g* file with objects *sample1* and *sample2* in it.

---

keypoint [*x y z* | *reset*]

The "keypoint" command without any options displays the current keypoint setting. If a point is specified, then that point becomes the *keypoint*. If *reset* is specified, then the default *keypoint* is restored. The *keypoint* is used as the center of rotation and scaling in solid edit or matrix edit (formerly known as object edit) modes. This command has no effect when used in nonediting modes.

Examples:

`mged` **keypoint 10 20 30**

-- Set the *keypoint* to the point (10 20 30) in model units.

---

kill [*-f*] <*objects*>

The "kill" command deletes the specified *objects* from the current database. This command affects only the *objects* actually listed on the command line. If a combination is killed, its members are not affected. If the *-f* option is specified, then kill will not complain if some, or all, of the *objects* specified do not actually exist in the database. Note that the *objects* are killed immediately. There is no need for a "write file" command in MGED, and there is no "undo" command. Use this command with caution. Other commands that remove objects from the database are killall and killtree.

Examples:

`mged` **kill group1 region2 shapeb**

-- Destroy *group1*, *region2*, and *shapeb.*

---

killall <*objects*>

The "killall" command deletes the specified *objects* from the current database and removes all references to them from all combinations in the database. Note that the *objects* are killed immediately. There is no need for a "write file" command in MGED, and there is no "undo" command. **Use this command with caution.** Other commands that remove objects from the database are kill and killtree.

Examples:

`mged` **killall group1 region2 shapeb**

-- Destroy *group1*, *region2*, and *shapeb* and remove all references to these objects from the database.

---

killtree <*objects*>

The "killtree" command deletes the specified *objects* from the current database and recursively deletes all objects referenced by any of those objects. If one of the *objects* listed is a combination, then that *combination* as well as any objects that are members of that *combination* will be deleted. If a member of that *combination* is itself a *combination*, then all of its members will also be destroyed. This continues recursively until the primitive shapes are reached and destroyed. Note that the *objects* are killed immediately. There is no need for a "write file" command in MGED, and there is no "undo" command. Use this command with extreme caution. Other commands that remove objects from the database are kill and killall.

Examples:

`mged` **killtree group1 region2 shapeb**

-- Destroy *group1*, *region2*, and *shapeb* and remove all references to these objects from the database.

---

knob [*-e -i -m -v*] [*-o v|m|e|k*] [*zap|zero|(id* [*val*])]

The "knob" command is used internally by MGED in the processing of knob input devices and is not recommended for users. The *knob* command provides a method for simulating knob input. With no options, it will display the current values for the knobs. With the *zap* or *zero* command provided, all the knob values will be reset to zero. If an *id* and *value* are provided, the specified knob setting will be simulated. If the *-i* option is specified, then the value provided will be used as an increment to be applied to the indicated knob. The knobs have different functions depending on the current mode. When in solid or matrix edit mode and a rotation or translation function is selected, the knob effects are applied to the edited object by default. However, the *-v* (view coordinates) and *-m* (model coordinates) options may be used with the *knob* command to adjust the view without modifying any shape or

object parameters. The *-e* option allows the knob effects to be applied to the edited object when they would normally be applied to the view. The *-o* option allows the origin of rotation to be specified. The units for *value* are degrees for rotation and local units for translation. The available *knob ids* are:

- **x** -- rate-based rotation about horizontal axis.
- **y** -- rate-based rotation about vertical axis.
- **z** -- rate-based rotation about axis normal to screen.
- **X** -- rate-based translation in horizontal direction.
- **Y** -- rate-based translation in vertical direction.
- **Z** -- rate-based translation in direction normal to screen.
- **S** -- rate-based Scale or Zoom.
- **ax** -- absolute rotation about horizontal axis.
- **ay** -- absolute rotation about vertical axis.
- **az** -- absolute rotation about axis normal to screen.
- **aX** -- absolute translation in horizontal direction.
- **aY** -- absolute translation in vertical direction.
- **aZ** -- absolute translation in direction normal to screen.
- **aS** -- absolute Scale or Zoom.
- **xadc** -- absolute translation of adc in horizontal direction (screen coordinates -2048 to +2048).
- **yadc** -- absolute translation of adc in vertical direction (screen coordinates -2048 to +2048).
- **ang1** -- absolute rotation of adc angle1 (degrees).
- **ang2** -- absolute rotation of adc angle2 (degrees).
- **distadc** -- distance setting of the adc (screen coordinates -2048 to +2048).

Examples:

`mged` **knob y 1**

-- Start the view rotating about the vertical axis.

---

l [*-r*] <*objects*>

The "l" command displays a verbose description about the specified list of objects. If a specified *object* is a path, then any transformation matrices along that *path* are applied. If the final *path* component is a combination, the command will list the Boolean formula for the *combination* and will indicate any accumulated transformations (including any in that *combination*). If a shader and/or color has been assigned to the *combination*, the details will be listed. For a region, its ident, air code, material code, and LOS will also be listed. For primitive shapes, detailed *shape* parameters will be displayed with the accumulated transformation applied. If the *-r* (recursive) option is used, then each *object* on the command line will be treated as a *path*. If the *path* does not end at a primitive *shape*, then all possible *paths* from that

point down to individual *shapes* will be considered. The *shape* at the end of each possible *path* will be listed with its parameters adjusted by the accumulated transformation.

Examples:

`mged` **l region1**

-- Display details about region1.

`mged` **l group1/group2/region1/shape3**
-- Display shape parameters for *shape3* with matrices applied from the *path.*

`mged` **l -r a/b**
-- Display all possible paths that start with *a/b* and end in a primitive *shape*
-- The shape parameters with the accumulated transformation applied will be displayed.

---

labelvert <*objects*>

The "labelvert" command labels the vertices of the indicated *objects* with their coordinate values in the MGED display window. The *objects* must have already been displayed using e, E, ev, B, or any other command that results in the display of an object.

Examples:

`mged` **labelvert shapeb**

-- Place coordinate values in display near the vertices of *shapeb.*

---

listeval [*path*]

Combinations may include transformation matrices to be applied to their members. A *path* through a series of *combinations* and ending with a primitive shape represents that *primitive shape* with the transformations accumulated through the path applied to it. The "listeval" command displays primitive shape parameters after applying the accumulated transformations from the indicated *path*. If the specified *path* does not end at a *primitive shape*, then all possible paths from the indicated *path* to any *primitive shape* will be evaluated and displayed.

Examples:

`mged` **listeval group1/region1/shapeb**

-- Display the parameters for *shapeb* after applying the transformation matrix from *group1* for *region1* and the transformation matrix from *region1* for *shapeb.*

---

loadtk

The "loadtk" command loads the initialization for the Tk window library. This is normally done automatically when the user attaches any display manager for MGED. If no display manager is attached, then the user must execute *loadtk* prior to using any Tk facilities.

Examples:

`mged` **loadtk**

-- Initialize the Tk window library.

---

lookat *x y z*

The "lookat" command adjusts the current view in MGED such that the eye is looking in the direction of the given coordinates, but does not move the eye point nor change the viewsize. This is similar to just rotating the viewers head to look at the specified point, putting that point in the center of the MGED display. The center command performs a similar function, but moves the eye_pt without changing the viewing direction.

Examples:

`mged` **lookat 10 20 30**

-- Rotate the view to place the point (10 20 30) (model coordinates) in the center of the display.

---

ls [*-a -c -r -s*] [*objects*]

The "ls" command with no *object* argument lists the name of every object in the database (in alphabetical order). If the *object* argument is supplied, only those *objects* are listed. The *object* argument may include regular expressions for matching. The following options are also allowed:

- a - List all objects in the database.
- c - List all combinations in the database.
- r - List all regions in the database.
- s - List all shapes in the database.

The *ls* command is a synonym for the t command. Note that when any of the above options are used, the output is not formatted.

Examples:

`mged` **ls shape***

-- list all objects with names beginning with "shape" (output is formatted).

`mged` **ls -a shape***
-- list all objects with names beginning with "shape."

`mged` **ls -s wheel***
-- list all shapes with names beginning with "wheel."

`mged` **ls -r wheel***
-- list all regions with names beginning with "wheel."

`mged` **ls -c suspension***
-- list all combinations with names beginning with "suspension."

---

l_muves *MUVES_component1 MUVES_component2 ...*

The "l_muves" command lists the *BRL-CAD* regions that are part of the indicated MUVES components. The internal list of MUVES components must have been created earlier by the read_muves command. The MUVES components listed on the command line must not use any wildcards (the expansion will result in *BRL-CAD* objects, not MUVES components).

Examples:

`mged` **l_muves fuel transmission**

-- List the *BRL-CAD* regions that make up the MUVES components named "fuel" and "transmission."

---

make *-t | new_shape type*

The "make" command creates a *new_shape* of the indicated type. The *new_shape* is sized according to the current viewsize and is dependent on the *type*. The possible values for *type* are:

- arb8 -- ARB (eight vertices).
- arb7 -- ARB (seven vertices).
- arb6 -- ARB (six vertices).
- arb5 -- ARB (five vertices).
- arb4 -- ARB (four vertices).
- sph -- Ellipsoid (sphere).
- ell -- Ellipsoid (ellipsoid of revolution).

- ellg -- Ellipsoid (general ellipsoid).
- tor -- Torus.
- tgc -- Truncated General Cone (most general TGC).
- tec -- Truncated General Cone (truncated elliptical cone).
- rec -- Truncated General Cone (right elliptical cylinder).
- trc -- Truncated General Cone (truncated right circular cone).
- rcc -- Truncated General Cone (right circular cylinder).
- half -- Half Space.
- rpc -- Right Parabolic Cylinder.
- rhc -- Right Hyperbolic Cylinder.
- epa -- Elliptical Paraboloid.
- ehy -- Elliptical Hyperboloid.
- eto -- Elliptical Torus.
- part -- Particle.
- nmg -- Non-Manifold Geometry (an NMG consisting of a single vertex is built).
- pipe -- Pipe.
- bot -- Bag of Triangles.
- grip -- support for joints.
- extrude -- experimental.
- sketch -- experimental.

Examples:

`mged` **make shapea sph**

-- Create a sphere named *shapea.*

`mged` **make -t**
-- Return a list of shape types handled by *make.*

---

make_bb *new_shape_name <objects|paths>*

The "make_bb" (make bounding box) command builds a single axis-aligned rectangular parallelepiped (RPP) that will contain the *objects* and *paths* specified. The *paths* will include any transformation matrices that accumulate from combinations along the path. The listeval command may be used to see the effect of transformation matrices along a path.

Examples:

`mged` **make_bb new_bb shapea group1/region3**

-- Create an RPP named *new_bb* that is large enough to contain *shapea* and *group1/region3.*

---

mater *combination* [*shader_parameters*[*RGB* [*inheritance*]]]

The "mater" command assigns shader parameters, RGB color, and inheritance to an existing combination. The information may be included on the command line; otherwise the user will be prompted for it. Some available shaders are:

- bump -- bump maps.
- bwtexture -- black and white texture maps.
- camo -- camouflage.
- checker -- checkerboard design.
- cloud -- 2D Geoffrey Gardner style cloud texture map.
- envmap -- environment mapping.
- fakestar -- a fake star pattern.
- fbmbump -- fbm noise applied to surface normal.
- fbmcolor -- fbm noise applied to color.
- fire -- flames simulated with turbulence noise.
- glass -- Phong shader with values set to simulate glass.
- gravel -- turbulence noise applied to color and surface normal.
- light -- light source.
- marble -- marble texture.
- mirror -- Phong shader with values set to simulate mirror.
- plastic -- Phong shader with values set to simulate plastic.
- rtrans -- random transparency.
- scloud -- 3D cloud shader.
- spm -- spherical texture maps.
- stack -- allows stacking of shaders.
- stxt -- shape texture mapping.
- texture -- full color texture mapping.
- turbump -- turbulence noise applied to surface normals.
- turcolor -- turbulence noise applied to color.
- wood -- wood texture.

Examples:

`mged` **mater region1 "plastic {tr 0.5 re 0.2}" 210 100 100 0**

-- Set *region1* to use the plastic shader with 50% transparency, 20% reflectivity, a base color of (210 100 100), and inheritance set to 0.

---

matpick #|*combination/member*

The "matpick" command selects which matrix in the illuminated path should be edited. A number may be specified with 0 being the topmost selection. A *combination/member* may be specified to indicate that the matrix in *combination* that corresponds to *member* is to be edited. This command is only useful in matrix edit

mode at the point where the user is selecting which matrix in the illuminated path should be edited. It is used internally by MGED; the user should generally use the mouse to make this selection.

Examples:

`mged` **matpick group1/region3**

-- Select the matrix for *region3* in *group1* for editing.

---

memprint

The "memprint" command displays memory maps for debugging purposes.

Examples:

`mged` **memprint**

-- List memory maps.

---

mirface #### *x|y|z*

The "mirface" command modifies an ARB shape by mirroring the indicated face along the selected *x*, *y*, or *z* axis. An *ARB* shape must be selected for editing. Not all faces of all *ARB* types may be edited using this command.

Examples:

`mged` **mirface 1234 x**

-- Modify currently edited *ARB* by moving the face opposite face 1234 such that it is the mirror image of face 1234 across the *yz* plane.

---

mirror *old_object new_object x|y|z*

The "mirror" command creates *new_object* by duplicating *old_object* and reflecting it along the indicated axis. If *old_object* is a primitive shape, then a new shape is created, with parameters adjusted to accomplish the *mirror* operation. If *old_object* is a combination, then *new_object* will simply be a copy of *old_object* with all of its members' matrices set to perform the appropriate reflection.

Examples:

`mged` **mirror shape1 shape1_mirror x**

-- Make a copy of *shape1*, name it *shape1_mirror*, and adjust its parameters so that it is a mirror image of *shape1* across the *yz* plane.

---

mrot *x y z*

Rotate the view using model *x y z*.

Examples:

`mged` **mrot 0 0 10**

-- Rotate the view about the model *z* axis by 10˚.

---

mv *old_name new_name*

The "mv" command changes the name of *old_name* to *new_name*. Note that this does not change any references to *old_name* that may appear in other combinations in the database. The mvall command will change an object's name everywhere.

Examples:

`mged` **mv shapea shapeb**

-- Change the name of *shapea* to *shapeb.*

---

mvall *old_name new_name*

The "mvall" command changes the name of *old_name* to *new_name*. This will also change any references to *old_name* that may appear in other combinations in the database. The mv command will change an object's name without changing references to it. The prefix command will also change the names and references of objects.

Examples:

`mged` **mvall shapea shapeb**

-- Change the name of *shapea* to *shapeb* everywhere it occurs in the database.

---

nirt [*nirt_args*]

The "nirt" command runs the *nirt* program that is distributed with *BRL-CAD* to intersect a single ray with the displayed objects. By default, *nirt* is run using the current database and the currently displayed objects, and it uses the current eye point as the ray start point and the current viewing direction as the ray direction. This effectively fires a ray at the center of the MGED display. The resulting collection of intersections between the ray and the objects is listed. Additional arguments may be supplied on the *nirt* command line. See the *man* page of *nirt* for more details.

Examples:

`mged` **nirt**

-- Fire a single ray through the center of the MGED display.

---

nmg_collapse *old_nmg_shape new_nmg_shape maximum_error_dist [minimum_angle]*

The "nmg_collapse" command simplifies an existing nmg_shape by a process of edge decimation. Each edge in the *old_nmg_shape* is considered; if it can be deleted without creating an error greater than the specified *maximum_error_dist*, then that edge is deleted. If a *minimum_angle* is specified (degrees), then the edge will not be deleted if it would create a triangle with an angle less than *minimum_angle*. The resulting shape is saved in *new_nmg_shape*. The *old_nmg_shape* must have been triangulated previous to using the *nmg_collpase* command. The resulting shape consists of all triangular faces.

Examples:

`mged` **nmg_collapse nmg_old nmg_new 1.0 10.0**

-- Decimate edges in *nmg_old* to produce an NMG with an error no greater than 1.0 units. The process will not create any triangles with an angle less than 10˚. The new NMG shape will be named *nmg_new*.

---

nmg_simplify [*arb*|*tgc*|*poly*] *new_shape nmg_shape*

The "nmg_simplify" command attempts to convert an existing nmg_shape to a simpler primitive shape type. The user may specify which type to attempt by including *arb, tgc,* or *poly* on the command line. If no shape type is specified, all will be attempted in the above order. If *tgc* is specified, the code will attempt to determine if the *nmg_shape* is an approximation of a TGC shape.

Examples:

`mged` **nmg_simplify poly shapea.poly shapea.nmg**

-- Convert the *NMG* shape named *shapea.nmg* to a polysolid named *shapea.poly.*

---

oed *path_lhs path_rhs*

The "oed" command places MGED directly into the matrix edit mode. The *path_rhs* must be a path to a primitive shape, and *path_lhs* must be a path to a combination that includes the first component of *path_rhs* as one of its members. The edited matrix will be the matrix in the final component of *path_lhs* that corresponds to the first component of *path_rhs*. The last component in *path_rhs* is used as the reference shape during object editing.

Examples:

`mged` **oed group1/group2 region1/shapea**

-- Place MGED into matrix edit mode, editing the matrix in *group2* that corresponds to *region1*, using *shapea* as the reference shape.

---

opendb [*database.g*]

The "opendb" command closes the current database file and opens *database.g*. If *database.g* is not found, the current database is left open. If *database.g* is not specified on the command line, the name of the current database file is returned.

Examples:

`mged` **opendb model.g**

-- Close the current database file and open *model.g*.

`mged` **opendb**
-- Return the name of the current database file.

---

orientation *x y z w*

The "orientation" command sets the view direction for MGED from the quaternion specified on the command line.

Examples:

`mged` **orientation 1 0 0 0**

-- Set viewing direction to bottom.

---

orot [*-i*] *xdeg ydeg zdeg*

The "orot" command performs a rotation of an object during matrix edit. The rotation is performed, in order: *xdeg* about the *x* axis, then *ydeg* about the *y* axis, and finally *zdeg* about the *z* axis. If the *-i* flag is given, then the angles are interpreted as increments to the last object rotation. The rotobj command is a synonym for *orot*.

Examples:

`mged` **orot 0 0 35**

-- Rotate currently edited object by 35˚ about the Z-axis from the original orientation.

---

oscale *scale_factor*

The "oscale" command of matrix edit mode modifies the matrix to perform a uniform scale operation. A *scale_factor* of 2 doubles the size of the associated object, and a *scale_factor* of 0.5 reduces it by half.

Examples:

`mged` **oscale 3**

-- Increase the size of the curently edited object by a factor of 3.

---

overlay *plot_file* [*name*]

The "overlay" command plots the specified UNIX *plot_file* in the MGED display. Phony object names are created for each part of the plot file that is in a unique color. The names are created by adding a color to the specified *name*, or to "_PLOT_OVER" if no name is provided. The color suffix is built by converting the RGB color to a six digit hex number. Each color corresponds to 2 hex digits, so that white becomes "ffffff," red becomes "ff0000," green is "00ff00," etc.

Examples:

`mged` **overlay plot.upl tmp**

-- Plot the Unix plot file *plot.upl* in the MGED display, using *tmp* as the base for the phony object names.

---

p *value1* [*value2 value3*]

The "p" command provides precise control over solid editing operations that would normally be done using the mouse or knobs. For example, a shape rotate may be performed by selecting *rotate* from the solid edit menu, then providing the rotation angles with the *p* command. A command of "p 0 30 0" would rotate the edited shape through 30˚ about the *y* axis. Similarly, many of the individual parameters of the edited shape may be set exactly using the *p* command. If the *scale H* menu item is selected while editing a TGC, then the *value1* supplied with a *p* command specifies the actual length of the height vector for that *TGC*. This method is the recommended technique to set precise values for shape parameters. The translate and rotobj commands provide a similar capability for object editing.

Examples:

`mged` **p 30**

-- Set the currently selected shape parameter of the currently edited shape to 30 units.

---

pathlist *<combinations>*

The "pathlist" command lists all existing paths that start from the specified combinations and end at a primitive shape.

Examples:

`mged` **pathlist group1 region2**

-- List all existing paths that start from the *combinations group1* and *region2* and end at *primitive shapes.*

---

paths *path_start*

The "paths" command lists all existing paths that start from the specified *path_start* and end at a primitive shape. The *path_start* may be specified by "/" separated components, or they may be separated by spaces (but not both).

Examples:

`mged` **paths group1 region2**

-- List all existing paths that start from *group1/region2* and end at a *primitive shape.*

---

permute *tuple*

The "permute" command permutes the vertex labels for the face of an ARB shape that is currently being edited. The *tuple* indicates which face is affected and also indicates the desired result. The tuple is formed by concating the list of vertex numbers for the face in the order desired such that the first vertex listed will become vertex number one (and therefore the default keypoint). Only a sufficient number of vertices to disambiguate need be included in the tuple. Note that this has no effect on the geometry of the *ARB*, but may affect any texture mapping involving this shape.

Examples:

`mged` **permute 321**

-- Rearrange the vertices of the currently edited *ARB* such that vertex #3 becomes vertex #1, vertex #2 remains #2, and vertex #1 becomes #3.

---

pl [-*float*] [-*zclip*] [-*2d*] [-*grid*] *out_file* | "|" *filter*

The "pl" command creates a UNIX plot of the current MGED display. If an *output_file* is specified, the plot is stored in that file. If a *filter* is specified, the output is sent to that *filter*. The *-float* option requests a plot file with real numbers rather than integers. The *-zclip* option requests that the plot be clipped to the viewing cube in the Z-direction. The *-2d* option requests a two-dimensional plot (the default is 3D).

The *-grid* option is intended to include a grid in the plot, but is currently not implemented. This command uses the *dm-plot* display manager. The plot command performs the same function, but does not use the *dm-plot* display manager.

Examples:

`mged` **pl -float | pldebug**

-- Create a UNIX plot of the current MGED display and pipe the results to the *pldebug* command.

---

plot [*-float*] [*-zclip*] [*-2d*] [*-grid*] *out_file* | "|" *filter*

The "plot" command creates a UNIX plot of the current MGED display. If an *output_file* is specified, the plot is stored in that file. If a *filter* is specified, the output is sent to that *filter*. The *-float* option requests a plot file with real numbers rather than integers. The *-zclip* option requests that the plot be clipped to the viewing cube in the Z-direction. The *-2d* option requests a two-dimensional plot (the default is 3D). The *-grid* option is intended to include a grid in the plot, but is currently not implemented. This command does not use the *dm-plot* display manager. The pl command performs the same function, but does use the *dm-plot* display manager.

Examples:

`mged` **plot -float | pldebug**

-- Create a UNIX plot of the current MGED display and pipe the results to the *pldebug* command.

---

polybinout *file*

Store vlist polygons into polygon file (experimental).

---

pov *args*

Set the point-of-view (experimental).

---

prcolor

The "prcolor" command lists the entries in the ident-based color table. The ident number for a displayed region is used to find the appropriate color from the lookup table. The *low* and *high* values are the limits of region ident numbers to have the indicated *r g b* color (0-255) applied. The color table entries may be modified using the color command, and the entire color table may be edited using the edcolor command. If a color lookup table exists, its entries will override any color assigned using the mater command.

---

prefix *new_prefix <objects>*

The "prefix" command changes the name of all the *objects* listed by adding the specified *new_ prefix*. All references to the *objects* will also be changed. The mvall command performs a similar function.

Examples:

`mged` **prefix test_ group1 regiona shapeb**

-- Change the names of objects *group1, regiona*, and *shapeb* to "test_group1," "test_regiona," and "test_shapeb." All references to these objects will reflect the new names.

---

press *button_label*

The "press" command simulates the pressing of a button. All of these button actions can be run directly as a command. The *button_label* indicates which button to simulate. The available buttons are:

- help -- Provide a list of the available *button_labels.*
- 35,25 -- Switch to a view from an azimuth of 35˚ and an elevation of 25˚.
- 45,45 -- Switch to a view from an azimuth of 45˚ and an elevation of 45˚.
- accept -- Simulate the *accept* button (accepts edits and writes the edited object to the database).
- reject -- Simulate the *reject* button (discards edits).
- reset -- Resets view to *top* and resizes such that all displayed objects are within the view.
- save -- Remember the current view aspect and size.
- restore -- Restore the most recently saved view.
- adc -- Toggle display of the adc.
- front -- Switch to view from the front (synonym for ae 0 0).
- left -- Switch to view from the left (synonym for ae 90 0).
- rear -- Switch to view from the rear (synonym for ae 180 0).
- right -- Switch to view from the right (synonym for ae 270 0).
- bottom -- Switch to view from the bottom (synonym for ae -90 -90).
- top -- Switch to view from the top (synonym for ae -90 90).
- oill -- Enter object illuminate mode.
- orot -- Enter object rotate mode (must already be in matrix edit mode).
- oscale -- Enter object scale mode (must already be in matrix edit mode).
- oxscale -- Enter object scale (x-direction only) mode (must already be in matrix edit mode).
- oyscale -- Enter object scale (y-direction only) mode (must already be in matrix edit mode).

- ozscale -- Enter object scale (z-direction only) mode (must already be in matrix edit mode).
- oxy -- Enter object translate mode (must already be in matrix edit mode).
- ox -- Enter object translate (horizontal only) mode (must already be in matrix edit mode).
- oy -- Enter object translate (vertical only) mode (must already be in matrix edit mode).
- sill -- Enter solid illuminate mode.
- sedit -- (deprecated) Enter solid edit mode.
- srot -- Enter solid rotate mode (must already be in solid edit mode).
- sscale -- Enter solid scale mode (must already be in solid edit mode).
- sxy -- Enter solid translate mode (must already be in solid edit mode).
- zoomin -- Zoom in, synonym for zoom 2.
- zoomout -- Zoom out, synonym for zoom 0.5.
- rate -- Toggle between rate and absolute mode for knobs and sliders.
- edit – (deprecated) Toggle between edit and view modes for knobs and sliders (useful during editing to allow the knobs and sliders to be used for either editing operations (in edit mode) or to adjust the view without affecting the edited object (in view mode).

Examples:

`mged` **press top**

-- Switch to view from the top direction.

---

preview [*-v*] [*-d delay*] [*-D start_frame_number*] [*-K end_frame_number*] *rt_script_file*

The "preview" command allows the user to *preview* animation scripts in MGED. The *-d* option provides a delay in seconds to be applied between each frame (the default is no delay). The *-D* option allows the user to specify a starting frame number, and the *-K* option allows the specification of an ending frame number. The *-v* flag indicates that the objects displayed in the MGED graphics window should be displayed in "evaluated" mode, as would be the result of the ev command. Note that this may significantly slow the *preview*.

Examples:

`mged` **preview -D 101 -K 237 script.rt**

-- *Preview* the animation script stored in the file named *script.rt* from frame number 101 through frame number 237.

---

prj_add *shaderfile* [*image_file*] [*image_width*] [*image_height*]

The "prj_add" command appends information to the specified *shaderfile*. The information appended is in the form required by the "projection" shader (*prj*) and includes the *image_file* (typically a "pix" file), the *image_width* and *image_height*, and current view parameters from the MGED display. The resulting *shaderfile* may then be used as the parameter to the *prj* shader. Before executing this command, the region wireframe display in MGED should be aligned with the *image_file* (underlayed in MGED's framebuffer), and the *image_file* should have the same height and width as the MGED display.

---

ps [*-f font*] [*-t title*] [*-c creator*] [*-s size_in_inches*] [*-l line_width*] *output_file*

The "ps" command temporarily attaches the *Postscript* display manager and outputs the current MGED display to the specified *output_file* in *PostScript* format. The *-f* option allows the *font* to be user-specified. The *-t* option allows the user to provide a title (the default is "No Title"). The *-c* option allows the user to specify the creator of the file (the default is "LIBDM dm-ps"). The *–s* specifies the size of the drawing in inches. The *–l* specifies the width of the lines drawn.

Examples:

`mged` **ps -t "Test Title" test.ps**

-- Place a *PostScript* version of the current MGED display in a file named *test.ps* and give it the title "Test Title."

`mged` **ps -l 10 -t "Test Fat Lines" fat_lines.ps**
-- This time use fat lines.

---

push <*objects*>

The "push" command forces the effects of all transformation matrices that appear in any combinations in the trees from the specified *objects* down to the primitive shapes. This will change the parameters of the *primitive shapes* if any of the transformation matrices are not identity matrices. All the transformation matrices visited will be set to identity matrices. This command will fail, and no changes will be made, if any *primitive shape* referenced by the list of *objects* is positioned differently in two or more *combinations*. The xpush command will perform a similar function, even if some shapes are multiply referenced.

Examples:

`mged` **push group1 regiona**

-- Push the effects of any transformation matrices in the trees headed by *group1* and *regiona* down to the *primitive shapes.*

---

putmat *comb_name/member_name {I | m0 m1 m2 m3 ... m16}*

The "putmat" command replaces the existing transformation matrix in the combination specified that corresponds to the member specified. The transformation matrix may be specified with an "I" to indicate the identity matrix, or it may be specified as 16 elements listed row-by-row. The copymat command allows the user to copy an existing transformation matrix.

Examples:

`mged` **putmat group1/regiona I**

-- Set the transformation matrix for *regiona* in *group1* to the identity matrix.

---

q

The "q" command ends the MGED process. Note that there is no write database command in MGED. All changes are made to the database as the user performs them. Therefore, a *q* command will not restore the database to its pre-edited state. This is a synonym for the quit command.

Examples:

`mged` **q**

-- Quit the current MGED session.

---

qorot *x y z dx dy dz angle*

The "qorot" command rotates an object through the specified *angle* (in degrees). This command requires that MGED already be in *matrix edit* mode. The edited object is rotated about the axis defined by the start point (*x y z*) and the direction vector (*dx dy dz*).

Examples:

`mged` **qorot 1 2 3 0 0 1 25**

-- Rotate the currently edited object through 25˚ about the axis through the point (1, 2, 3) and in the Z-direction.

---

qray [*subcommand*]

Get/set *query ray* characteristics. Without a subcommand, the usage message is printed. The *qray* command accepts the following subcommands:

vars

Print a list of all query ray variables.

basename [*str*]

If *str* is specified, then set basename to *str*. Otherwise, return the basename. Note that the basename is the name used to create the fake shape names corresponding to the query ray. There will be one fake shape for every color used along the ray.

effects [*t*|*g*|*b*]

Set or get the type of *effects* that will occur when firing a query ray. The effects of firing a ray can be either *t* for textual output, *g* for graphical output or *b* for both textual and graphical.

echo [*0*|*1*]

Set or get the value of *echo*. If set to 1, the actual nirt command used will be echoed to the screen.

oddcolor [*r g b*]

Set or get the color of odd partitions.

evencolor [*r g b*]

Set or get the color of even partitions.

voidcolor [*r g b*]

Set or get the color of areas where the ray passes through nothing.

overlapcolor [*r g b*]

Set or get the color of areas that overlap.

fmt [*r*|*h*|*p*|*f*|*m*|*o* [*str*]]

Set or get the format string(s). See the *man* page of *nirt* for more details.

script [*str*]

Set or get the nirt script string.

help

Print the usage message.

Examples:

`mged` **qray**

-- Print usage message.

`mged` **qray fmt o**
-- Returns the overlap format string.

`mged` **qray oddcolor**
-- Returns the rgb color used to color odd partitions.

`mged` **qray oddcolor 255 0 0**
-- Sets the odd partition color to red.

---

query_ray [*nirt_args*]

The "query_ray" command runs the *nirt* program that is distributed with *BRL-CAD* to intersect a single ray with the displayed objects. By default, *nirt* is run using the current database and the currently displayed objects and uses the current eye point as the ray start point and the current viewing direction as the ray direction. This effectively fires a ray at the center of the MGED display. The resulting list of intersections between the ray and the objects is given. Additional arguments may be supplied on the *nirt* command line. See the *man* page of *nirt* for more details.

Examples:

`mged` **query_ray**

-- Fire a single ray through the center of the MGED display.

---

quit

The "quit" command ends the MGED process. Note that there is no write database command in MGED. All changes are made to the database as the user performs them. Therefore, a *quit* command will not restore the database to its pre-edited state. This is a synonym for the q command.

Examples:

`mged` **quit**

-- Quit the current MGED session.

---

qvrot *dx dy dz angle*

The "qvrot" command adjusts the current MGED viewing direction such that the eye is positioned along the direction vector (*dx dy dz*) from the view center and is looking towards the view center. The *angle* (in degrees) allows for a twist about the viewing direction. The ae command provides a similar capability.

Examples:

`mged` **qvrot 0 0 1 90**

-- Set the current view to the same as achieved by the press top command.

---

r *region_name <operation object>*

The "r" command creates a region with the specified *region_name*. The *region* is constructed using the list of Boolean operations and *object* pairs. The operators are represented by the single characters "+," "-," and "u" for intersection, subtraction, and union, respectively. The *object* associated with each operator may be a combination or a primitive shape. No parentheses or any grouping indication is allowed in the *r* command. The operator hierarchy for the *r* command has been established through the ancestry of *BRL-CAD* and does not conform to accepted standards (see the c command for a more standard implementation). Intersection and subtraction operations are performed first, proceeding left to right; then union operations are performed. *BRL-CAD* regions are special cases of *BRL-CAD combinations* and include special attributes. Default values for these attributes may be set using the regdef command. As new *regions* are built, the default ident number gets incremented. If *region_name* already exists, then the *operation*/*object* pairs get appended to its end.

Examples:

`mged` **r new_region u shape1 - shape2 u shape3 + group4**

-- Create a region named *new_region* that consists of two parts unioned together. The first part is *shape1* with *shape2* subtracted. The second part is the intersection of *shape3* and the combination *group4*.

---

rcodes *file_name*

The "rcodes" command reads the specified file and assigns the region attributes to the regions listed. The file is expected to be in the format produced by the wcodes command.

Examples:

`mged` **rcodes region_codes**

-- Read the file named *region_codes* and set the region specific attributes according to the values found in the file.

---

read_muves *MUVES_regionmap_file*

The "read_muves" command reads the indicated "MUVES_regionmap_file" and creates an internal list of all the MUVES components defined in the file along with the corresponding *BRL-CAD* regions. This list can then be used to display the *regions* in terms of the MUVES component names. See the e_muves, t_muves, and l_muves commands.

Examples:

`mged` **read_muves region_map**

-- Read the MUVES file named *region_map*.

---

red *combination*

The "red" command creates a file describing the specified combination and starts an editor for the user to modify the combination. The environment *EDITOR* variable will be used to select the editor. If *EDITOR* is not set, then */bin/ed* will be used. All the attributes of *BRL-CAD* regions and *combinations* may be edited in this way. The *region* specific attributes will be ignored if the *combination* is not a *region* and is not set to be a *region* during editing. It is not necessary to be in an editing mode to run this command. The rm, r, comb, c, and g commands provide some basic *combination* editing capabilities.

Examples:

`mged` **red group2**

-- Edit the combination *group2* with the user's editor of choice.

---

redraw_vlist *object*

Given the name(s) of database objects, re-generate the vlist associated with every shape in view that references the named object(s), either shapes or regions. Particularly useful with outboard .inmem database modifications.

---

refresh

The "refresh" command updates the MGED display.

Examples:

`mged` **refresh**

-- Update the MGED display.

---

regdebug [*debug_level*]

The "regdebug" command with no options toggles the display manager debug flag. If a *debug_level* is supplied, then the display manager debug flag is set to that value.

Examples:

`mged` **regdebug**

-- Toggle the display manager debug flag.

---

regdef *item* [*air* [*los* [*material_code*]]]

The "regdef" command sets the default region attributes used by the r and c commands when building a *BRL-CAD region*. The default *ident* number is incremented each time a new *region* is created with the *r* or *c* commands.

Examples:

`mged` **regdef 1003 0 100 8**

-- Set the *region* default attributes to an ident of 1003, an air code of 0, an los of 100%, and a material code of 8.

---

regions *output_file* <*objects*>

The "regions" command creates a summary of all the regions in the specified list of *objects*. The summary is written in the specified *output_file*. The summary includes, for each *region*, a sequential region number, its ident, air code, material code, los, the path from one of the *objects* to the *region*, and the Boolean formula for the *region*.

Examples:

`mged` **regions regions_file group1 group2**

-- Place a summary of all the *regions* from *group1* and *group2* in the file named *regions_file.*

---

release [*name*]

The "release" command is used to close a display manager. If invoked with no arguments, the current display manager is closed. Otherwise, *name* (i.e., the Tcl/Tk

path name of the display manager window) is closed.

Examples:

`mged` **release**

-- Close the current display manager.

`mged` **release .dm_X0**
-- Close *.dm_X0.*

---

rfarb

The "rfarb" command creates a new ARB8 shape based on rotation and fallback angles. The command prompts the user for all the required information. In addition to the name for the new shape and the rotation and fallback angles, the user is prompted for the coordinates of one corner of the *ARB8* and for two of the three coordinates of the other three vertices of one face of the *ARB8*. The other coordinate of each of these vertices is calculated in order to ensure that the face is planar. The user is then prompted for a thickness, and the first face is extruded normally by the specified thickness to complete the *ARB8*.

Examples:

`mged` **rfarb**

-- Create a new *ARB8* shape according to arguments supplied in answer to prompts.

---

rm *combination <members>*

The "rm" command deletes all occurences of the listed members from the specified combination. The red, r, comb, c, and g commands provide other *combination* editing capabilities.

Examples:

`mged` **rm group1 regiona**

-- Delete *regiona* from *group1*.

---

rmater *file*

The "rmater" command reads the specified *file* and sets the combinationshader, color, and inheritance values according to those listed in the *file*. The format of the *file* is expected to be as produced by the wmater command.

Examples:

`mged` **rmater mater_file**

-- Read the file named *mater_file* and set the *combination* attributes according to those listed in the file.

---

rmats *file*

The "rmats" command reads the specified *file* and sets the current MGED view to agree with the parameters in the *file*. The format of the file is expected to be as produced by the savekey command.

Examples:

`mged` **rmats key_file**

-- Read the file named *key_file* and set the current MGED viewing direction according to the parameters found there.

---

rot *x y z*

The "rot" command rotates the view or an object by *xyz* degrees. Exactly what is rotated and how it is rotated are dependent on MGED's state as well as the state of the display manager. See arot for a similar capability.

Examples:

`mged` **rot 0 0 45**

-- Rotate 45˚ about the Z axis.

`mged` **rot 45 45 0**
-- Rotate 45˚ about the *y* axis, then rotate 45˚ about the *x* axis.

---

rotobj [-*i*] *x-angle y-angle z-angle*

The "rotobj" command rotates the currently edited object by *z* angle degrees about the *z* direction, *y* angle about the *y* direction, and *x* angle degrees about the *x* direction in that order. If an -*i* option is included, then the rotations are treated as increments to the previous rotations. MGED must be in the matrix edit mode for this command to be useful. The p command provides a similar capability for solid editing.

Examples:

`mged` **rotobj 0 0 25**

-- Rotate the currently edited object by 25˚ about the *z* direction from the original orientation.

---

rrt *program* [*options*]

The "rrt" command executes the specified *program* with the provided *options* and includes the current database name and the list of currently displayed objects on the command line. This command effectively executes:

```
program options database_name objects.
```

The *rrt* command also provides the current MGED viewing parameters to the *program* on standard input. Many *BRL-CAD* programs use the *-M* option to indicate that viewing parameters will be provided on standard input. The rt command can be simulated with *rrt* as:

```
rrt /usr/brlcad/bin/rt -M -s50
```

provided that perspective is not currently being used. Any executable routine may be run using *rrt*; however, it will always be run with the provided *options* followed by the current database name and the list of currently displayed objects.

Examples:

`mged` **rrt echo**

-- Will list the current database name and the list of currently displayed objects.

---

rt [*options*] [-- *objects*]

The "rt" command executes the *BRL-CAD rt* program with the default options of "-s50 –M." If perspective is turned on, then the *-p* option will be included with the value of the perspective angle. The current database name is added to the end of the *rt* command line along with either the specified *objects* or, if none is specified, the list of currently displayed objects. The *rt* program is written such that options may be repeated, and the last occurrence of an option will override any earlier occurences. This allows the user to specify other *size* (*-s*) options. The rrt command performs a similar function, but may be used to execute other programs as well. The *-M* option tells *rt* to read the viewing parameters from standard input. See the *man* page on *rt* for details. A related command is saveview, which can be used to create a shell script (batch job) to raytrace this view in the background.

Examples:

`mged` **rt -s1024 -F/dev/Xl**

-- Run the *rt* program to produce a color-shaded image of the current view in the MGED display. The image will be 1024 pixels square and will be displayed on a lingering *X* framebuffer.

`mged` **rt -C 200/200/255 -- roof**
-- Run the *rt* program to produce a color-shaded image of the object *roof* using MGED's current viewing parameters. The image will have a sky-blue background and will be displayed on the framebuffer specified by the FB_FILE shell variable.

---

rtcheck [*options*]

The "rtcheck" command executes the *BRL-CAD rtcheck* program with the default options of "-s50 –M." The *-M* option tells *rtcheck* to read the viewing parameters from standard input so that rays are only fired from the current view. The current database name and the list of currently displayed objects are added to the end of the *rtcheck* command line. The *rtcheck* program is written such that options may be repeated, and the last occurrence of an option will override any earlier occurences. This allows the user to specify other *size* (*-s*) options. The rrt command performs a similar function, but may be used to execute other programs as well. The *rtcheck* program uses raytracing to check for overlapping regions in the list of objects passed on the command line. When invoked from within MGED, any discovered overlaps along a ray are represented as yellow lines that extend only in the areas of overlap. Details and a count of overlaps are also reported. Note that overlaps of less than 0.1 mm are currently ignored by *rtcheck*. The default option of *-s50* indicates that the checking rays should be fired from a uniform square grid with 50 rays on a side. This is very coarse and may miss significant overlaps. It is recommended that the user select appropriate options for the *rtcheck* program and execute it for a variety viewing aspects to perform a thorough check. The granularity of the grid may be controlled with the *-s*, *-w*, *-n*, *-g*, and *-G* options. See the *man* page on *rtcheck* for details.

Examples:

`mged` **rtcheck -g10 -G10**

-- Run the *rtcheck* program with rays fired from a uniform grid with the rays spaced every 10 mm.

---

savekey *file* [*time*]

The "savekey" command saves the current viewing parameters in the specified *file* in the format expected by the rmats command. If a *time* is included, it will also be written to the specified *file*. If the *file* already exists, the information will be

appended to its end. The parameters saved this way are useful as keypoints in constructing an animation. The *BRL-CAD anim_keyread* program will read a *file* constructed by using the *savekey* command with some number of different views in MGED with sequential times specified. The *anim_keyread* program will produce a table of keyframes suitable for use with other *BRL-CAD* animation tools.

Examples:

`mged` **savekey key_file 5**

-- Append the current viewing parameters to *key_file* and tag this as the key frame at 5 seconds.

---

saveview *file* [*args*]

The "saveview" command saves the current viewing parameters in the specified *file* in the form of a shell script that will run the *BRL-CADrt* program as if it had been executed from within MGED using the rt -s512 command. Any *args* included on the *saveview* command line will be copied to the *file* as options to the *rt* program. If the *file* already exists, the script will be appended to it. This is useful in setting up images to be raytraced later. The default script produced by "saveview test.rt" looks like:

```
#!/bin/sh
rt -M \
 -o test.rt.pix\
 $*\
 model.g\
 'object1' 'object2' \
 2 test.rt.log\
 <<EOF
viewsize 2.780320739746094e+02;
orientation 2.480973490458727e-01 4.765905732660483e-01
7.480973490458729e-01 3.894348305183902e-01;
eye_pt 1.234152656421214e+02 7.220202900588745e+01
3.845765464924686e+01;
start 0; clean;
end;
EOF
```

When this script is executed, the image will be stored in *test.rt.pix*, and all messages and errors generated by the *rt* program will be stored in *test.rt.log*. The above script will produce an image of *object1* and *object2* from the *BRL-CAD* database named *model.g*. The *viewsize, orientation*, and *eye_pt* parameters reproduce the view displayed by MGED when the *saveview* command was executed. The presence of "$*" in the script causes any additional command-line options given when the script is invoked to be interpreted as additional *rt* options. Typically, a "-s" option might be used to set the image size (the default is 512 pixels square). See the *man* page on *rt* for details on available options.

If you have a *saveview* script and wish to change MGED to that view, merely cut-and-paste, or *source*, the viewsize, orientation, and eye_pt lines from the *saveview* file into MGED.

Related MGED commands are preview, for viewing the effects of an entire animation script, and savekey. Related *BRL-CAD* programs are *tabsub* and *tabinterp*.

Examples:

`mged` **saveview rt_script -s1024**

-- Create (or append to) a file named *rt_script* that will contain a script to run the *rt* program and create a color shaded image of the current MGED display. The image produced will be 1024 pixels square.

---

sca *sfactor*

The "sca" command is used to apply a scaling factor. The effect is determined by the Transrom option in the Settings menu. This is normally affected by the current mode of operation in MGED (e.g., matrix edit, solid edit, or viewing).

Examples:

`mged` **sca 2**

-- In matrix edit mode, the object being affected will get twice as big.
-- In view mode, the size of the view will be doubled (showing twice the volume of space, hence making objects appear half their previous size on the display).

---

sed *path*

The "sed" command places MGED directly into the solid edit mode. The path must uniquely identify a primitive shape. If the *shape* is only referenced once in the objects being displayed, then *path* may simply be the shape name. If the *shape* is multiply referenced, then the *path* should be the full *path* from a top level displayed object to the *primitive shape* to be edited. The who command will return a list of the top-level objects currently being displayed.

Examples:

`mged` **sed shape1**

-- Enter solid edit state for shape1.

---

setview *x-angle y-angle z-angle*

The "setview" command sets the current view in MGED by specifying rotation angles (in degrees) about the *x*, *y*, and *z* axes. The rotations are performed about the *z* axis first, then the *y* axis, then the *x* axis. The "setview 0 0 0" command is a synonym for press top.

Examples:

`mged` **setview 90 180 90**

-- Set the current view to that set by ae 0 0.

---

shader *combination shader_name* [*"{shader_args}"*]

The "shader" command assigns shader parameters to the specified combination. The *shader_name* indicates which *shader* should be assigned. If *shader_args* are supplied, they will be assigned to parameters of the indicated shader. This performs a similar function as the mater command.

Examples:

`mged` **shader group1 checker "{a 0,255,0 b 0,0,255}"**

-- Assign the checkerboard shader to *group1* using green and blue colors for the squares.

---

shells *NMG_shape*

The "shells" command separates the specified NMG shape into its constituent shells. Each shell is written to the database as a separate *NMG* object with a name of the form "shell" with a number appended to make the name unique. If the *NMG* has only one shell, then only one new object will be created. This differs from the decompose command in that *decompose* will actually break the object into a number of separate shells if possible.

Examples:

`mged` **shells object.nmg**

-- Break the *NMG* shape named *object.nmg* into its constituent shells.

---

showmats *path*

The "showmats" command lists the transformation matrices encountered along the specified path and also lists the accumulated matrix at the end of the *path*. If any member occurs more than once in a combination along the path, then a matrix will be listed for each occurrence of that *member*, and the accumulated matrix will only use

the first occurrence. Related commands are putmat, copymat, and listeval.

Examples:

`mged` **showmats head/skull/jaw**

-- List the transformation matrices along the *path* "head/skull/jaw" and the accumulated matrix for the entire *path.*

---

size *view_size*

The "size" command sets the size of the current viewing cube to the specified *view_size* (in local units). This size is the length of any side of the square MGED display.

Examples:

`mged` **size 250**

-- Set the MGED display to be 250 units across.

---

solids *file <objects>*

The "solids" command lists a summary of all the primitive shapes used in regions referenced by the list of *objects*. The summary is written to the specified *file*. The summary is similar to that produced by the regions command, but with the addition of *primitive shape* parameters. The *shape* parameters listed will have the accumulated transformation matrices along the path from the listed *objects* to the *primitive shape* applied (as would be listed by the listeval command). The showmats command may be used to see the actual transformation matrices.

Examples:

`mged` **solids shapes_summary group1 regiona**

-- Write a summary of all the regions in *group1* and include the region named *regiona*. The summary will include detailed *shape* parameters for the shapes used in the regions.

---

status [*subcommands*]

Without a subcommand, the *status* command returns the following information: current state, view size of the current display manager, the conversion factor from local model units to the base units (mm) stored in the database, and the view matrices of the current display manager. *Status* accepts the following subcommands:

state

Get the current state of MGED (i.e., “VIEWING,” “SOL PICK,” “SOL EDIT,” “OBJ PICK,” “OBJ PATH,” “OBJ EDIT,” or “VERTPICK”).

Viewscale

Get the view scale.

base2local

Get the conversion factor from base units (mm) to local units.

local2base

Get the conversion factor from local units to base units (mm).

toViewcenter

Get the matrix that describes the location of the view center.

Viewrot

Get the matrix that describes the view orientation.

model2view

Get the model to view conversion matrix.

view2model

Get the view to model conversion matrix.

model2objview

Get the model to view conversion matrix. This matrix also includes changes made during editing.

objview2model

Get the view to model conversion matrix. This matrix also includes changes made during editing.

help

Print the usage message.

Examples:

`mged` **status**

-- Get default information (i.e., state, view size, local2base, toViewcenter, Viewrot, model2view and view2model).

`mged` **status Viewrot**
-- Get the view rotation matrix.

`mged` **status state**
-- Get the edit state.

---

summary [*s r g*]

The "summary" command with no arguments lists the number of primitive shapes, regions, and non-region combinations in the current database. If the *s* argument is supplied, then the name of each *primitive shape* is also listed. Similarly, the *r* flag asks for the *region* names, and *g* asks for the names of all the combinations (including *region*). The flags may be concatenated to get combined output.

Examples:

`mged` **summary sr**

-- List a summary of primitive shapes and regions for the current database.

---

sv *x y* [*z*]

The "sv" command moves the view center to (*x, y, z*). If *z* is not provided, then *z*=0 is used. The parameters *x, y, z* are integer values relative to the screen. For example, the center of the screen is (0, 0, 0) and the upper left corner is (-2048, 2047, 0).

Examples:

`mged` **sv 0 0 0**

-- The view is unchanged.

`mged` **sv 1024 0 0**
-- The view center is moved half way between the current view center and the right side of the view screen.

---

sync

The "sync" command causes all information in memory that should be on disk to be written out.

Examples:

`mged` **sync**

-- Make sure disk files are up to date.

---

t [*-a -c -r -s*] [*objects*]

The "t" command with no *object* argument lists the name of every object in the database (in alphabetical order). If the *object* argument is supplied, only those *objects* are listed. The *object* argument may include regular expressions for matching. The following options are also allowed:

- a - list all objects in the database.
- c - list all combinations in the database.
- r - list all regions in the database.
- s - list all shapes in the database.

The *t* command is a synonym for the ls command. Note that when any of the above options is used, the output is not formatted.

Examples:

`mged` **t shape***

-- List all objects with names beginning with "shape."
(output is formatted)

`mged` **t -a shape***
-- List all objects with names beginning with "shape."

`mged` **t -s wheel***
-- List all shapes with names beginning with "wheel."

`mged` **t -r wheel***
-- List all regions with names beginning with "wheel."

`mged` **t -c suspension***
-- List all combinations with names beginning with "suspension."

---

ted

The "ted" command places the parameters of the currently edited primitive shape into a file, then starts a text editor for the user to modify the parameters. The editor used is whatever the user has set in the environment variable *EDITOR*. If *EDITOR* is not set, then */bin/ed* is used. MGED must be in the Solid Edit Mode prior to using this command. The red command performs a similar function for combinations.

Examples:

`mged` **ted**

-- Use a text editor to modify the currently edited shape.

---

title [*string*]

The "title" command, with no arguments, returns the title string for the current database. If command line arguments are supplied, they will become the new title string for the current database. Quotation marks must be doubly escaped.

Examples:

`mged` **title This is my \\"database\\"**

-- Set the title of the current database to *This is my "database."*

---

tol [*abs #*] [*rel #*] [*norm #*] [*dist #*] [*perp #*]

The "tol" command, with no arguments, lists the current tolerance settings. If the command line includes any of the keywords followed by a number, then that tolerance setting will be modified. The keywords are:

- Tessellation tolerances:
  The tessellation tolerances are used to control the facetization of primitive shapes. If more than one tolerance value is specified, the tessellation is performed to meet the most stringent.
  - abs -- This *absolute* tolerance is specified in model units and represents the maximum allowable error in the distance from the actual shape surface to the tessellated surface. An *absolute* tolerance of 0 means that the *absolute* tolerance should be ignored.
  - rel -- This *relative* tolerance is specified in terms of a fraction of the shape size. The value is multiplied by the size of the shape to determine another bound on the maximum allowable error in the distance from the actual shape surface to the tessellated surface. A *relative* tolerance of 0 means that the *relative* tolerance should be ignored.
  - norm -- This *normal* tolerance is specified in degrees and represents the maximum angle between the actual shape surface normal and the tessellated surface normal. A *normal* tolerance of 0 means that the *normal* tolerance should be ignored.
- Calculational tolerances:
  The calculational tolerances are used in evaluating the Boolean operations specified in a combination. This is used, for example, in the ev, facetize, and bev commands.

- dist -- The *distance* tolerance is specified in model units and represents the minimum distance required between two vertices to consider them distinct.
- perp -- The *perpendicularity* tolerance is specified as the cosine of an angle. Two objects will be considered perpendicular if the cosine of the angle between them is less than the *perpendicularity* tolerance. Similarly, two objects will be considered parallel if the cosine of the angle between them is greater than 1.0, the *perpendicularity* tolerance.

Examples:

`mged` **tol rel .05 perp 1e-6**

-- Set the *relative* tolerance to 5% and the *perpendicularity* tolerance to 1e-06 (cosine of 89.9999˚).

---

tops

The "tops" command displays a list of all the *top-level* objects in the current database. The top-level objects are all those objects that are not referenced by some other combination. The hierarchical structure of *BRL-CAD* databases usually means that there will be a top-level object that includes all (or at least most) of the objects in the database.

Examples:

`mged` **tops**

-- List all the top-level objects in the current database.

---

tra *dx dy dz*

The "tra" command translates the view or an object. Exactly what is done is determined by MGED's state as well as the state of the current display manager. The parameters *dx*, *dy*, and *dz* are in local units.

Examples:

`mged` **tra 10 0 0**

-- Translate by 10 units along the *x* axis.

---

track [*parameters*]

The "track" command builds a simple representation of the linked track of a vehicle such as a tank. With no command line arguments, the *track* command will prompt for all the required input. The vehicle is assumed to be axis-aligned with the front in the +*x* direction. A combination name for the track is built by appending a unique

number to the string "track." The information about the track may be included on the command line, and is order-dependent. The parameters are (in order):

- *x* coordinate of center of frontmost roadwheel.
- *x* coordinate of center of rearmost roadwheel.
- *z* coordinate of center of all roadwheels.
- radius of all roadwheels.
- *x* coordinate of center of drive wheel (rear).
- *z* coordinate of center of drive wheel (rear).
- radius of drive wheel.
- *x* coordinate of center of idler wheel (front).
- *z* coordinate of center of idler wheel (front).
- radius of idler wheel.
- *y* coordinate of right side of track.
- *y* coordinate of left side of track.
- track thickness.

Examples:

`mged` **track 500 0 10 10 -50 50 10 550 50 10 -50 -20 2**

-- Build a simple track using the provided arguments.

---

translate *x y z*

The "translate" command is used to precisely control the translation of an object in both solid edit and matrix edit modes. The keypoint of the edited object or shape is translated to the specified coordinates.

Examples:

`mged` **translate 10 20 30**

-- Move the currently edited object to the model coordinates (10 20 30).

---

tree [-*c*] [-*i #*] [-*o outfile*] *object(s)*

The "tree"command will list the contents of the specified *objects* in a tree-like format that displays the hierarchical structure of the *objects*, and all objects referenced by them, down to the primitive shape level. If -*c* is given, the shapes are not printed. The -*o outfile* option prints the results to *outfile*. The -*i #* option allows the user to set the number of spaces to indent.

Examples:

`mged` **tree group1**

-- Show the structure of the tree rooted at *group1* down to the primitive shape level.

`mged` **tree -i 2 group1**
-- This time use two spaces for each level of indentation.

`mged` **tree -c group1**
-- No shapes are printed.

---

t_muves

The "t_muves" command lists all the MUVES components that are known as a result of a prior read_muves command.

Examples:

`mged` **t_muves**

-- List all the known MUVES components.

---

units [*units_type*]

The "units" command, with no arguments, will return the current type of units that MGED is using. If a *units_type* is specified, MGED will switch to editing in the indicated units. The actual database is always stored in millimeters, and the display is adjusted to the users choice of units. If the *units_type* specified on the command line is one of the types allowed, it will be written to the database file as the preferred units and succeeding invocations will use those units. The *units_type* strings that will be remembered as the preferred editing unit are:

- mm -- millimeters.
- millimeter -- millimeters.
- cm -- centimeters.
- centimeter -- centimeters.
- m -- meters.
- meter -- meters.
- in -- inches.
- inch -- inches.
- ft -- feet.
- foot -- feet.
- feet -- feet.
- um -- micrometers.

Units_type strings that may be used, but will not be remembered as the preferred editing units, are:

- angstrom.
- decinanometer.

- nanometer.
- nm.
- micron.
- micrometer.
- km.
- kilometer.
- cubit.
- yd.
- yard.
- rd.
- rod.
- mi.
- mile.
- nmile.
- nautical mile.
- au.
- astronomical unit.
- lightyear.
- pc.
- parsec.

Examples:

`mged` **units in**

-- Switch to editing in "inches" and remember this as the preferred editing units for this database.

---

vars [*variable=value*]

The "vars" command, with no arguments, will list all the MGED variables and their values. If a *variable=value* string is included on the command line, then that *value* is assigned to the specified *variable*. Note that no spaces are allowed around the "=". The available *variables* are:

- autosize -- if nonzero, then MGED will set the viewsize whenever it draws to an empty display.
- rateknobs -- if nonzero, then the knobs and sliders act as rate adjustments; otherwise, they act as absolute adjustments.
- sliders -- if nonzero, the sliders are displayed.
- faceplate -- if nonzero, the MGED faceplate is displayed.
- orig_gui -- if nonzero, the "viewing" menu is displayed.
- linewidth -- indicates how wide to draw lines.
- linestyle -- set line style of wireframe shapes. Currently not being used.

- hot_key -- the X11 keysym definition for the key to toggle the *send_key* value. The default is "0xFFC6" (65478 decimal), which is the F9 key. The keysym values are defined in the X11 file named *keysymdef.h*.
- context -- if nonzero (the default), then solid editing parameters entered via the p command will be applied to the edited shape in the context of the combination tree above it in the displayed hierarchy. This means that, for example, a translation applied to a shape will translate the shape to some point such that when the transformation matrices for that path are applied, the edited shape will appear at the specified location. If *context* is set to zero, then the solid edit operations will be applied directly to the edited shape. This means that a translation to a specific point may result in the edited shape being drawn at a point different from that specified (due to transformations in the *combination* tree above it). Note that this only affects solid edit operations that use the *p* command.
- dlist -- if nonzero, use display lists.
- use_air -- if nonzero, use air while raytracing.
- listen -- if nonzero, listen for connections to MGED's built-in fbserv.
- port -- port for the built-in fbserv to use.
- fb -- if nonzero, then framebuffer is active.
- fb_all -- if nonzero, use entire geometry window for the framebuffer; otherwise, use only the rectangular area.
- fb_overlay -- if nonzero, overlay framebuffer image over geometry; otherwise, draw geometry over the framebuffer image.
- mouse_behavior -- see the following list of mouse behaviors:
    - c -- fire ray for combination edit selection.
    - d -- default behavior (i.e., as found in classic MGED).
    - m -- fire ray for matrix edit selection.
    - p -- paint rectangular area.
    - q -- fire query rays.
    - r -- raytrace rectangular area.
    - s -- fire ray for solid edit selection.
    - z -- zoom rectangular area.
- coords -- see the following list of coordinate systems to use for transformations:
    - m -- model coordinates.
    - v -- view coordinates.
    - o -- object coordinates.
- rotate_about -- see the following list of centers of rotation:
    - v -- view center.
    - e -- eye.
    - m -- model origin.
    - k -- keypoint.
- transform -- see the following list of things to transform:
    - a -- transform the angle distance cursor if active; otherwise same as v.
    - e -- apply transformations to the edit.

    - v -- apply transformations to the view.
- predictor -- if nonzero, the predictor frame will be displayed.
- predictor_advance -- the number of seconds into the future to advance the predictor frame.
- predictor_length -- not currently used.
- perspective -- if greater than zero, this is the perspective angle in degrees; otherwise, perspective is turned off.
- perspective_mode -- if nonzero, turn perspective on; otherwise, turn it off.
- toggle_perspective -- used to toggle among the four canned perspective angles (i.e., 30, 45, 60, and 90).
- nmg_eu_dist -- when the *-u* option to the ev command is used, the NMG edgeuses are drawn this distance (mm) away from the actual edge.
- eye_sep_dist -- if greater than zero, this is the eye separation distance (mm) for stereo viewing; otherwise, stereo is off.
- union_op -- not currently used.
- intersection_op -- not currently used.
- difference_op -- not currently used.

Examples:

`mged` **vars sliders=1**

-- Turn on the *sliders*.

---

vdraw *command* [*args*]

The "vdraw" command allows drawing of lines and polygons (optionally with per vertex normals) in the MGED graphics display. It is used to build a named list of drawing commands for MGED, send the list to the MGED display, modify the list, or delete all or part of the list. All vertices in the *vdraw* command are in millimeters. The MGED drawing commands are represented by integers in the *vdraw* command. The MGED drawing commands and the integers that *vdraw* uses for them are:

MGED Drawing Command	Vdraw integer	MGED Action
RT_VLIST_LINE_MOVE	0	begin a new line at this point
RT_VLIST_LINE_DRAW	1	draw line from previous point to this point
RT_VLIST_POLY_START	2	start polygon (argument is surface normal)
RT_VLIST_POLY_MOVE	3	move to first polygon vertex
RT_VLIST_POLY_DRAW	4	subsequent polygon vertices
RT_VLIST_POLY_END	5	last polygon vertex (should be same as first)
RT_VLIST_POLY_VERTNORM	6	vertex normal (for shading interpolation)

The vdraw commands are:

- open -- with no arguments, this returns "1" if there is a open list; "0" otherwise. If an argument is supplied, a command list is opened with the provided name.
- write -- with arguments of *i c x y z*, the MGED drawing command #*c* is placed in the ith position of the command list with the vertex as (*x y z*).
  -- with arguments of *next c x y z*, the command is placed at the end of the list.
- insert -- with arguments of *i c x y z*, the MGED drawing command #*c* is inserted just before the ith position of the command list.
- delete -- with an integer argument of *i*, the ith command is deleted.
  -- with an argument of "last," the last command on the list is deleted.
  -- with an argument of "all," all the commands on the list are deleted.
- params -- with an argument of *color rrggbb*, the color of all objects on this list is set. The *rrggbb* is a hex number representing the color, "ffffff" is white, "ff0000" is red, "00ff00" is green, etc.
  -- with a single string argument, the name of the current list is changed.
- read -- with an integer argument of *i*, the ith command is returned.
  -- with an argument of "color," the current color is returned.
  -- with an argument of "length," the number of commands in the current list is returned.
  -- with an argument of "name," the name of the current command list is returned.
- show -- send the current command list to the MGED display manager.
- vlist -- with an argument of "list," return a list of the names of all existing command lists.
  -- with an argument of *delete list_name*, delete the specified command list.

All textual arguments may be abbreviated by their first letter.

Examples:

`mged` **vdraw open square**

-- Open a list named *square.*

`mged` **vdraw params color ff00**
-- Set color to green.

`mged` **vdraw write next 0 0 0 0**
-- Start a line at the origin.

`mged` **vdraw write next 1 100 0 0**
-- Draw line to (100 0 0).

`mged` **vdraw write next 1 100 100 0**
-- Draw line to (100 100 0).

`mged` **vdraw write next 1 0 100 0**
-- Draw line to (0 100 0).

`mged` **vdraw write next 1 0 0 0**
-- Draw line to (0 0 0).

`mged` **vdraw show**
-- Draw the square in the MGED display.

---

view *subcommand*

Get/set view parameters (local units). The v*iew* command accepts the following subcommands:

center [*x y z*]--get/set the view center of the current view.

size [*val*]--get/set the view size of the current view.

eye [*x y z*]--get/set the eye point of the current view.

ypr [*y p r*]--get/set the yaw, pitch, and roll of the current view.

quat [*v1 v2 v3 v4*]--get/set the view in the form of a quaternion.

aet [*a e t*]--get/set the azimuth, elevation, and twist of the current view.

Examples:

`mged` **view center**

-- Get the view center.

`mged` **view center 0 0 0**
-- Set the view center at the origin of model space.

---

vnirt [*nirt args*] *x y*

This command interprets *x* and *y* as view coordinates (i.e., +-2047) and converts them to model coordinates (local units) using a value of 2047 for view *z* before passing them to nirt. All other arguments are passed to nirt without modification.

---

vquery_ray *x y*

Same as vnirt.

---

vrmgr *host* {*master* | *slave* | *overview*}

The "vrmgr" command establishes a link between the current MGED display and a *vrmgr* process running on the specified *host*. The *vrmgr* program is a manager for virtual reality displays using MGED. The *vrmgr* process must be started on *host* prior to executing the *vrmgr* command in MGED. The second command line argument to the *vrmgr* command is the role of the current MGED display. The *master* display controls the viewing parameters of itself and all the *slave* displays. The *overview* display acts as an observer of the entire virtual reality process.

Examples:

`mged` **vrmgr host1.arl.mil master**

-- Set the current MGED display as the *master* for the *vrmgr* process running on the host named *host1.arl.army.mil.*

---

vrot *xrot yrot zrot*

The "vrot" command rotates the view on the current geometry display window. The parameters *xrot, yrot*, and *zrot* are rotations (specified in degrees) about the *viewing coordinate axes*.

If the display is in rotate-about-center mode, then the rotation will occur about the center of the viewing volume. In rotate-about-eye mode, the view on the display will be rotated about the eye. The vars command (or a menu button) allows the user to toggle between the two modes.

Examples:

`mged` **vrot 90 0 0**

-- Rotate 90˚ about view *x* axis.

`mged` **vrot 0 180 0**
-- Rotate 180˚ about view *y* axis.

---

wcodes *file <objects>*

The "wcodes" command writes ident, air code, material code, LOS, and name of all the regions in the list of *objects* to the specified *file*. The format used is compatible with the rcodes command.

Examples:

`mged` **wcodes code_file group1 group2**

-- Write region data for all the regions in *group1* and *group2* to *code_file.*

---

whatid *region_name*

The "whatid" command lists the ident number of the specified *region*.

Examples:

`mged` **whatid regiona**

-- Get the *ident* number for *regiona.*

---

which_shader <*shaders*>

The "which_shader" command lists all the regions that use one of the *shaders* specified.

Examples:

`mged` **which_shader plastic light**

-- List all *regions* in the current database that use the *plastic* or *light* shaders.

---

whichair <*air_codes*>

The "whichair" command lists all the regions that use one of the air codes specified. The eac command will perform a similar search, but will draw the qualifying regions in the MGED display rather than listing them. Regions that have nonzero ident numbers will not be listed by this command.

Examples:

`mged` **whichair 2 3**

-- List all *regions* in the current database that have *air codes* of 2 or 3.

---

whichid <*idents*>

The "whichid" command lists all the regions that use one of the idents specified.

Examples:

`mged` **whichid 1002 1003**

-- List all *regions* in the current database that have *idents* of 1002 or 1003.

---

who [*real* | *phony* | *both*]

The "who" command lists the top-level objects that are currently displayed. The *phony* flag asks for just *phony* objects. *Phony* objects are typically objects that are drawn in the MGED display, but are not actual database objects. Some *phony* objects are drawings from the vdraw command and the edgeuses drawn by the ev -u command. The *real* flag asks for just *real* objects, and the *both* flag asks for both *real* and *phony* objects. The default is just *real* objects. Any of the flags may be abbreviated by its first letter. The x command also lists displayed shapes, but in more detail.

Examples:

`mged` **who p**

-- List all top-level *phony* objects currently displayed.

---

wmater *file <objects>*

The "wmater" command lists the shader name and parameters, RGB color, *RGB_valid* flag, and the inheritance flag to the specified *file* for the listed *objects*. The format used is compatible with the rmater command. If *file* already exists, the new data is appended to it.

Examples:

`mged` **wmater mater_file group1 regiona**

-- List the shader parameters of *group1* and *regiona* to *mater_file.*

---

x [*level*]

The "x"command lists all the primitive shapes currently drawn in the MGED display. The *level* determines how much detail should be included in the list. For *level* zero (the default), only a list of paths to shapes in the display list is produced. Each shape is prefixed by "VIEW" or "-no-," indicating that the shape is actually being drawn or that it is being skipped, respectively. If *level* is greater than zero, the center, size, ident number, RGB color assigned to the region, and the actual color used to draw the shape are also listed. If *level* is greater than one, the number of vlist structures and the number of points in each vlist structure are also listed for each shape. If *level* is greater than two, then the actual lines drawn for each shape are also listed. The who command performs a similar function, but lists only the top-level objects that are displayed.

Examples:

`mged` **x**

-- List the paths to the displayed shapes.

---

xpush *object*

The "xpush" command "pushes" the effects of transformation matrices in the paths, from the specified *object* to the primitive shapes, into the shapes and replaces all the transformation matrices with identity matrices. The push command performs a similar function, but will refuse to make any changes if any shape needs to be transformed into more than one location/orientation/scale. The *xpush* command will recognize such situations and create extra copies of such shapes to accept the different transformation effects. New shapes created by this command will have a suffix appended to the original name to make the new name unique. Combinations referring to the new shape will also be modified to reflect the name change. The push command performs the same function but will refuse to make any changes if it cannot accomplish the "push" without creating any new shapes.

Examples:

`mged` **xpush group1**

-- Move all the effects of the transformation matrices in the tree rooted at *group1* down to the shape level (creating new shapes if needed).

---

zoom *scale_factor*

The "zoom" command changes the size of the viewing cube for the MGED display, resulting in a "zoom in" or "zoom out" effect. A *scale_factor* greater than one reduces the size of the viewing cube ("zoom in"). A *scale_factor* of less than one increases the size of the viewing cube ("zoom out").

Examples:

`mged` **zoom 2**

-- Reduces the size of the current viewing cube by half (effectively doubling the size of objects in the display).

MGED Developer Commands

aip	cmd_win	collaborate	get_comb	get_dm_list
get_edit_solid	get_more_default	grid2model_lu	grid2view_lu	gui_destroy
hist	make_name	mged_update	mmenu_get	mmenu_set
model2grid_lu	model2view	model2view_lu	output_hook	put_comb
put_edit_solid	reset_edit_solid	rset	set_more_default	share
solids_on_ray	stuff_str	svb	tie	view2grid_lu
view2model	view2model_lu	view2model_vec	view_ring	viewget
viewset	winset			

---

aip [*f*|*b*]

The "aip" command advances the illumination pointer when MGED is in the solid illuminate state or the object illuminate state. In either of the *illuminate* states, a single primitive shape is highlighted at one time, and the path to that shape is displayed. Moving the mouse vertically in the MGED display causes different shapes to be highlighted and their paths to be displayed. The *aip* command causes the next shape in the list to be highlighted (if used with no arguments or the *f* argument), or it causes the previous shape in the list to be highlighted (if used with the *b* argument). When the desired shape is highlighted, it is selected by clicking the middle mouse button or by using the M command. If MGED is in the object path state, the place along the path where the edit is to take place will advance. Once the desired path position is displayed, it is selected by clicking the middle mouse button or by using the M command.

Examples:

mged **press sill**

-- Enter solid illuminate mode.

mged **aip**
-- Hightlight the next shape.

mged **aip b**
-- Highlight the previous shape.

mged **M 1 0 0**
-- Select the highlighted shape.

---

cmd_win *subcommand*

This command is used to maintain internal command window structures. The *cmd_win* command accepts the following subcommands:

open *id* -- This subcommand is used to create the internal data structures for a new command window. If *id* is already in use, nothing is changed.

close *id* -- This subcommand releases *id*'s internal data structures. If the referenced command window is tied to a display manager, then that association is also removed.

set *id* -- This subcommand sets the current command window to *id*. If this command window is tied to a display manager, that display manager becomes the current display manager.

get -- This subcommand returns the id of the current command window.

Examples:

mged **cmd_win open my_id**

-- Create a command window named *my_id*.

mged **cmd_win close my_id**
-- Closes the command window *my_id.*

mged **cmd_win set my_id**
-- *my_id* becomes the current command window.

mged **cmd_win get**
-- Gets the current command window.

---

collaborate *subcommand*

This command is used to maintain the "collaborative session." The collaborative session is a list whose members share a view and view ring among the upper right display manager panes. The *collaborate* command accepts the following subcommands:

join *id* -- This causes the GUI associated with *id* to join the collaborative session.

quit *id* -- This causes the GUI associated with *id* to quit the collaborative session.

show -- This returns the list of participants (ids) in the collaborative session.

Examples:

mged **collaborate join bill**

-- *bill* is added to the collaborative session.

mged **collaborate quit bill**
-- *bill* is removed from the collaborative session.

mged **collaborate show**
-- Show list of collaborative participants.

---

get_comb *comb_name*

The "get_comb" command returns a Tcl list of information about *comb_name*. If *comb_name* is a region, the following information is returned:

NAME REGION REGION_ID AIRCODE GIFT_MATERIAL
LOS COLOR SHADER INHERIT BOOLEAN_FORMULA

Otherwise, the following shorter list is returned:

NAME REGION COLOR SHADER INHERIT BOOLEAN_FORMULA

Examples:

mged **get_comb some_region**

some_region Yes 1000 0 1 100 {0 220 220} plastic No { u box - ball }

mged **get_comb some_non_region**
some_non_region No {0 220 220} plastic No { u box2 - ball2 }

---

get_dm_list

The "get_dm_list" command returns a list of all open display managers. The members of this list are the actual Tcl/Tk window names of the open display managers.

---

get_more_default

Returns the current default input value.

---

grid2model_lu *gx gy*

Given a point in grid coordinates (local units), convert it to model coordinates (local units).

---

grid2view_lu *gx gy*

Given a point in grid coordinates (local units), convert it to view coordinates (local units).

---

gui_destroy *id*

Destroy the GUI (Graphical User Interface) represented by *id*. Note that this GUI must have been created with the gui command.

---

hist *subcommand*

This command is used to maintain command history. *Hist* accepts the following subcommands:

add *command*

This adds *command* to the history list of commands executed during the current MGED session. If *command* is more than one word, it must be surrounded by braces (i.e., {make box arb8}).

next

This returns the next command in the command history list.

prev

This returns the previous command in the command history list.

Examples:

mged **hist add {ae 35 25}**

-- Add the command "ae 35 25" to the history list.

mged **ae 0 90**
mged **hist prev**
-- Return the previous command (i.e., **ae 0 90**).

---

make_name *template*

make_name *-s [num]*

This command generates an object name that does not occur in the database. The name, which is generated in the format specified by *template*, contains an integer count. By default, this count appears at the end of the generated name, but if *template* contains the '@' character, then the count appears at that position in the name.

Examples:

mged **make_name wheel**

-- Returns, say "wheel5."

mged **make_name tree@trunk**
-- Returns "tree@trunk." The two-character string '@@' is interpreted as the literal '@', and thus is ignored for the purposes of positioning the name count. The integer counter starts at 0, and, by default, it is incremented each time *make_name* is executed. The *-s* option resets this counter. If the argument *num* is specified, the counter is set to this value. Otherwise, it is set to 0.

---

mged_update *non_blocking*

This command is used to handle outstanding events and to refresh the MGED display(s). This may be useful in certain Tcl scripts to maintain interactivity while waiting for user input. Note that if *non_blocking* is negative, outstanding events will not be processed. That is, only the MGED display(s) will be refreshed.

Examples:

mged **mged_update 0**

-- Update the MGED display, blocking (i.e., handle all outstanding events; if none, wait for one).

mged **mged_update 1**
-- Update the MGED display, nonblocking (i.e., handle all outstanding events; if none, return immediately).

---

mmenu_get [*i*]

This command is used to get MGED's internal menus. If *i* is not specified, return all three internal menus. Otherwise, return the ith menu. Note - valid values for *i* are 0, 1, or 2.

Examples:

mged **mmenu_get**

-- Return all internal menus.

mged **mmenu_get 2**
-- Return the internal menu number 2.

---

mmenu_set *id i*

This Tcl proc is used to set/install MGED's i^{th} internal menu in the Tcl/Tk button menu owned by *id*.

mged **mmenu_set bill 0**

-- Install MGED's 0th internal menu into *id*'s button menu.

---

model2grid_lu *mx my mz*

Convert a point in model coords (local units) to a point in grid coords (local units).

---

model2view *x y z*

The "model2view" command converts the supplied point (in model coordinates) to view coordinates. View coordinates are the coordinates in the viewing cube with values between -1.0 and +1.0 being inside the viewing cube.

Examples:

mged **model2view 10 20 30**

-- Display the view coordinates that correspond to the point (10 20 30) in model space.

---

model2view_lu *mx my mz*

Convert a point in model coordinates (local units) to a point in view coordinates (local units).

---

output_hook [*hook_cmd*]

Set up to have output from bu_log sent to *hook_cmd*. If *hook_cmd* is not specified, the output hook is deleted.

---

put_comb *comb_name is_Region* [*id air gift los*] *color shader inherit Boolean_expr*

The "put_comb" command defines the combination *comb_name*. If *is_Region* is *Yes*, then *id, air, gift* and *los* must be specified. If *is_Region* is *No*, then *id, air, gift,* and *los* must not be specified.

Examples:

mged **put_comb not_region No \"0 220 220\" plastic No \"u box\\n- ball\"**

-- Defines a combination called *not_region.*

mged **put_comb my_region Yes 1000 0 1 100 \"0 220 220\" plastic No \"u box\\n-ball\"**
-- Defines a region called *my_region.*

---

reset_edit_solid

Reset the parameters for the currently edited shape (i.e. es_int) to the database values.

---

rset [*res_type* [*res* [*vals*]]]

Provides a mechanism to get/set resource values for the given resource types. The supported resource types are: ax (Axes), c (Color Schemes), g (Grid), r (Rubber Band), and var (MGED Variables). Basically, *rset* always gets a value unless enough information is given to set a value. For example, with no parameters, *rset* returns a list of all resource values for the supported resource types. If *rset* is executed with only the *res_type* given, a list of all resource values for the given *res_type* is returned.

Examples:

mged **rset g**

Grid
draw=0
snap=0
anchor=0,0,0
rh=1
rv=1
mrh=5
mrv=5

mged **rset g snap**
-- Get value for grid snapping 0.

mged **rset g snap 1**
-- Enable snapping.

mged **rset r**
Rubber Band
draw=0

linewidth=0
linestyle='s'
pos=0,0
dim=0,0

mged **rset r draw**
-- Get value for "Rubber Band's" draw variable 0.

mged **rset r draw 1**
-- Draw rubber band.

mged **rset ax**
Axes
model_draw=0
model_size=500
model_linewidth=1
model_pos=0,0,0
view_draw=0
view_size=500
view_linewidth=1
view_pos=0,0
edit_draw=0
edit_size1=500
edit_size2=500
edit_linewidth1=1
edit_linewidth2=1.
--Prints the values of the axes

mged **rset ax model_size**
-- Get size of model axes 500.

---

set_more_default *more_default*

Set the current default input value.

---

share *[-u] resource dm1 [dm2]*

The "share" command provides a mechanism to share (or unshare with the **–u** option) resources among display managers. When a resource is shared between two or more display managers, any change to that resource is seen only in the sharing display managers. The supported resource types are: ad (ADC), ax (Axes), c (Color Schemes), d (Display Lists), g (Grid), m (Menu), r (Rubber Band), vi (View), and var (MGED Variables).

Examples:

mged **share g .dm_ogl0 .dm_ogl1**

-- .dm_ogl0 shares its grid resource with .dm_ogl1.

mged **share -u g .dm_ogl1**
-- .dm_ogl1 acquires a private copy of the grid resource.

---

solids_on_ray *h v*

List all displayed shapes along a ray.

---

stuff_str *string*

Sends a string to MGED's tty, while leaving the current command line alone. This is used to relay the activity of Tcl/Tk command windows to MGED's tty. If MGED is no longer attached to a tty, nothing happens.

---

svb

The "svb" command sets the view reference base variables, which are used internally by the knob command to implement absolute rotation, translation, and scale.

Examples:

mged **svb**

-- Set the view reference base variables with respect to the current view.

---

tie [[-*u*] *command_window* [*display_window*]]

The "tie" command is used to create (or untie/destroy with the **-u** option) an association between a command window and a display window. When there exists such an association, all commands entered from the command window will be directed at the associated display window. The *command_window* can be specified with MGED to refer to the tty from which MGED was started or an id associated with a Tcl/Tk interface window created with *gui* . The *display_window* is specified with its Tcl/Tk pathname. If no parameters are given, a listing of the current *command_window*/*display_window* pairs is returned. If only the *command_window* is given, the *display_window* associated with *command_window* is returned. If both parameters are given, the *command_window*/*display_window* association is created.

Examples:

mged **tie my_id .my_display_window**

-- Create the association between my_id and .my_display_window.

mged **tie my_id**
.my_display_window
-- Returns the display window associated with my_id.

mged **tie**
{my_id .my_window} {mged {}}
-- List all of the command_window/display_window pairs.

---

view2grid_lu *vx vy vz*

Given a point in view coordinates (local units), converts to grid coordinates (local units).

---

view2model *x y z*

The "view2model" command converts the specified point (*x y z*) in view coordinates to model coordinates (mm). The part of view space displayed by MGED is the cube -1.0 <= x,y,z <= +1.0.

Examples:

mged **view2model 1 1 0**

-- List the model coordinates of the upper right corner of the MGED display (in a plane at the center of the viewing cube).

---

view2model_lu *vx vy vz*

Given a point in view coordinates (local units), converts to model coordinates (local units).

---

view2model_vec *vx vy vz*

Given a vector in view coordinates, convert it to model coordinates.

---

view_ring *subcommand*

This manipulates the view ring for the current display manager. The view ring is a

list of views owned by a display manager. Views can be added or removed and can also be traversed or queried. *View_ring* accepts the following subcommands:

add

This subcommand adds the current view to the view ring.

next

This subcommand makes the next view on the view ring the current view.

prev

This subcommand makes the previous view on the view ring the current view.

toggle

This subcommand toggles between the current view and the last view.

delete *vid*

This subcommand removes/deletes the view with a view id of *vid* from the view ring. The last view cannot be removed (i.e., there is always one view on the view ring).

goto *vid*

This subcommand makes the view with a view id of *vid* the current view.

get [-*a*]

Returns the id of the current view. If -*a* is specified, all view ids on the view ring are returned.

Examples:

mged **view_ring add**

-- Add the current view to the view ring.

mged **view_ring goto 1**
-- Go to view 1.

mged **view_ring delete 1**
-- Delete view 1 from the view ring.

---

viewget *parameter*

The "viewget" command displays various *mged* view parameters. The possible parameters are:

- aet -- list the azimuth, elevation, and twist for the current viewing aspect.

- center -- list the model coordinates (mm) of the center of the viewing cube.
- size -- list the size (mm) of a side the current MGED display.
- eye -- list the model coordinates (mm) of the current eye point.
- ypr -- list the yaw, pitch, and roll angles (degrees) of the current viewing aspect.
- quat -- list the quaternion for the current viewing aspect.

Examples:

mged **viewget center**

-- List the model coordinates of the center of the MGED viewing cube.

---

viewset <*parameter value*>

The "viewset" command sets various MGED view parameters. More than one parameter may be set with one command. The possible parameters are:

- aet -- set the azimuth, elevation, and twist for the current viewing aspect.
- center -- set the model coordinates (mm) of the center of the viewing cube.
- size -- set the size (mm) of a side of the current MGED display.
- eye -- set the model coordinates (mm) of the current eye point.
- ypr -- set the yaw, pitch, and roll angles (degrees) of the current viewing aspect.
- quat -- set the quaternion for the current viewing aspect.

Examples:

mged **viewset center 1 2 3 size 100**

-- Set the model coordinates of the center of the MGED viewing cube to the point (1 2 3) and set the size of the viewing cube to 100 mm.

---

winset [*pathName*]

The "winset" command sets the current display manager to *pathName*. If *pathName* is not given, the current display manager is returned.

Examples:

mged **winset .my_window**

-- .my_window is now the current display manager.

mged **winset**
-- Returns the current display manager (i.e., .my_window).

Intentionally Left Blank

Appendix B:

Emacs and Vi Commands

Intentionally Left Blank.

MGED emulates two popular text editors, emacs and vi. The default emulation is emacs, which is available for use as soon as you launch MGED. If you prefer the vi emulation, move your mouse cursor to **File** on the menu bar and select **Preferences**. From the drop-down menu that appears, select **Command Line Edit** and then **vi**. The vi emulator will then be available for use.

The commands for emacs emulation are:

Command	**Description**
BACKSPACE	backward delete a character
DELETE	backward delete a character
Left	go backward one character
Right	go forward one character
Up	repeat the previous command
Down	go to the next command
Home	go to the beginning of the line
End	go to the end of the line
CTRL+a	go to the beginning of the line
CTRL+b	go backward one character
CTRL+c	interrupt command (not really an emacs command, but it works)
CTRL+d	delete character under cursor
CTRL+e	go to the end of line
CTRL+f	go forward a character
CTRL+h	delete character to left of cursor
CTRL+k	delete characters to end of line
CTRL+p	go to previous command
CTRL+t	transpose next two characters
CTRL+u	undelete line
CTRL+w	delete to beginning of line

The commands for vi emulation are:

Command	**Description**
Insert Mode	
ESCAPE	command
Left	move backward one character
Right	move forward one character
BACKSPACE	delete character to left of cursor

Command Mode

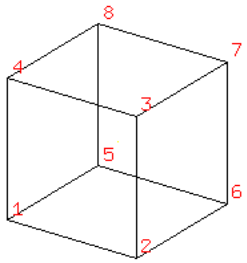
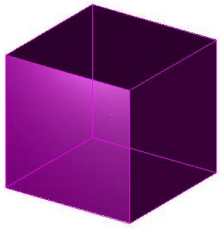
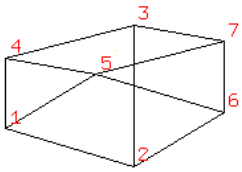
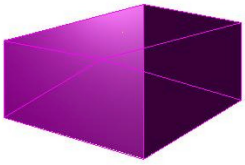
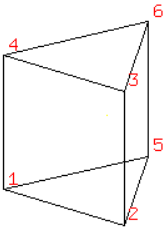
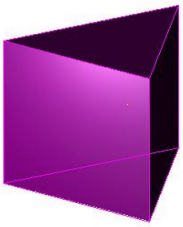
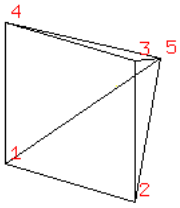
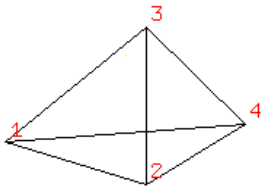
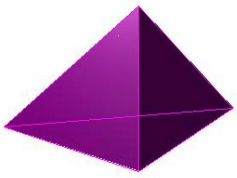
Left	move backward one character
Right	move forward one character
BACKSPACE	move backward one character
Space	move forward one character
A	insert text at end of line
C	end of line
D	delete text to end of line
F	search backward character
I	go to beginning of line and insert character(s)
R	overwrite text to right of cursor
X	backward delete a character
O	go to beginning of line
$	go to end of line
;	continue search in same direction
,	continue search in opposite direction
a	go forward one character and append character(s)
b	go backward one word
c	change word
d	delete
e	go to end of word
f	search forward one character
h	go backward one character
i	insert character(s)
j	next command
k	previous command
l	go forward a character
r	replace a character
s	delete a character and insert new character(s)
w	move forward a word
x	delete a character

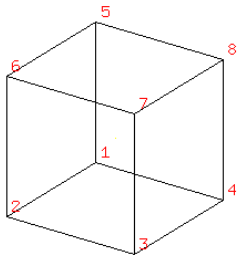
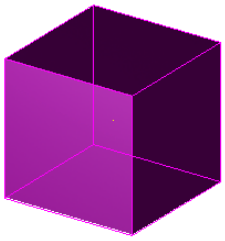
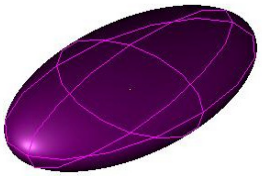
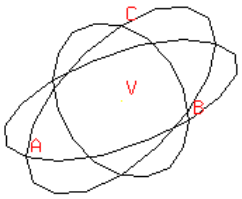
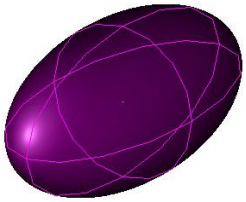
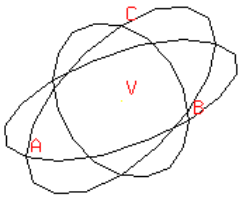
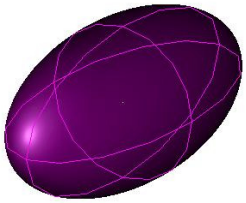
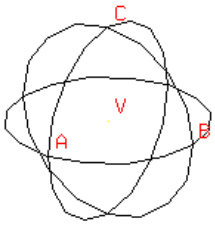
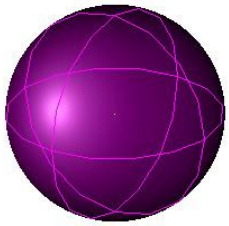
Appendix C:

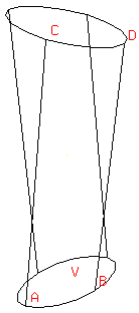
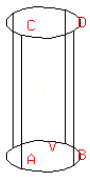
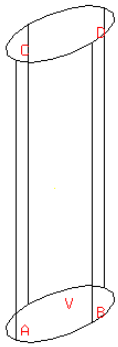
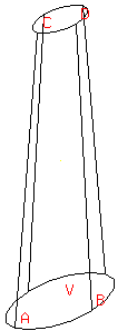
Primitive Shapes

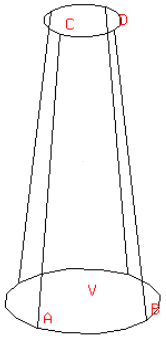
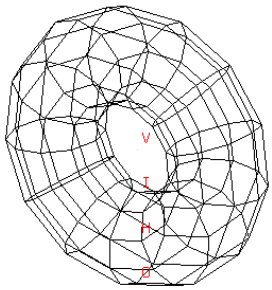
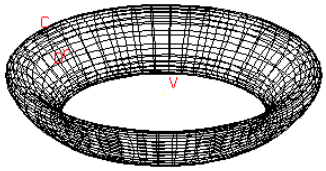
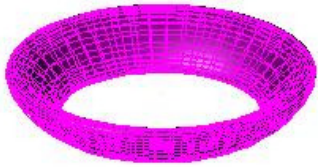
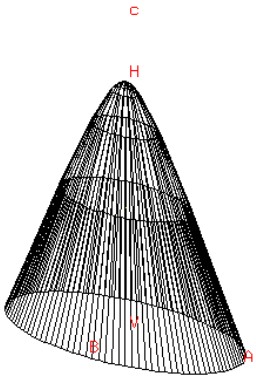
Intentionally Left Blank

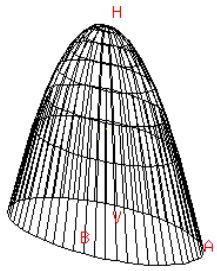
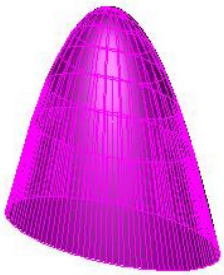
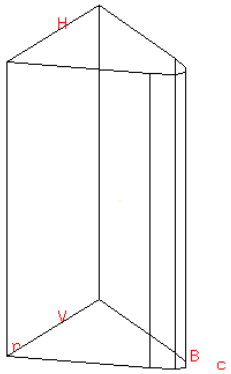
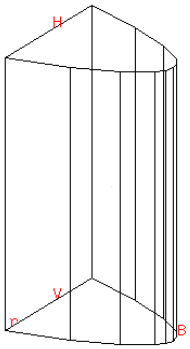
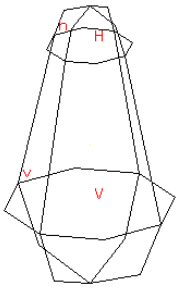
The following is a list of primitive shapes that MGED currently recognizes. Note in the Definitions column that a number of these shapes are stored in the database as other shapes because they are, in fact, special cases of those shapes. For example, a sphere (sph) is stored as an ellipsoid (ell) because, in actuality, it is an ellipsoid with equal-length vectors.

Primitive Shape/ BRL-CAD Abbreviation	**Input Parameters/ Definitions**	**Raytraced Figure**
 Arbitrary Convex Polyhedron, 8pts (arb8)	8 vertices	
 Arbitrary Convex Polyhedron, 7pts (arb7)	7 vertices Stored as an **arb8**	
 Arbitrary Convex Polyhedron, 6pts (arb6)	6 vertices Stored as an **arb8**	
 Arbitrary Convex Polyhedron, 5pts (arb5)	5 vertices Stored as an **arb8**	
 Arbitrary Convex Polyhedron, 4pts (arb4)	4 vertices Stored as an **arb8**	

 Rectangular Parallelepiped (rpp)	8 vertices Stored as an **arb8**	
 Ellipsoid (ell)	-Vertex, V -Vector A -Vector B -Vector C Vectors A, B, and C are perpendicular.	
 Ellipsoid Generic (ellg)	-Focus point 1 -Focus point 2 -Chord length, l (which is longer than the distance between the foci) Stored as an **ell**, where vectors A, B, and C are perpendicular and B and C have equal lengths.	
 Ellipsoid 1 (ell1)	-Vertex, V -Vector A -Radius, r Stored as an **ell**, where vectors A, B, and C are perpendicular and B and C have equal lengths.	
 Sphere (sph)	-Vertex, V -Radius, r Stored as an **ell,** where vectors A, B, and C are perpendicular and have equal lengths.	

 Truncated General Cone (tgc)	-Vertex, V -Height vector, H -Vector A -Vector B -Scalar c -Scalar d A and B are perpendicular vectors specifying an ellipse at the base. Vectors A and C are parallel, and vectors B and D are parallel. C is formed by scaling A by c. D is formed by scaling B by d.	
 Right Circular Cylinder (rcc)	-Vertex, V -Height vector, H -Radius, r Stored as a **tgc**, where vectors A and B have equal lengths, C and D have equal lengths, and all vectors are perpendicular to H.	
 Right Elliptical Cylinder (rec)	-Vertex ,V -Height vector, H -Major axis for the ellipse -Minor axis for the ellipse Stored as a **tgc**, where vectors A and C have equal lengths, B and D have equal lengths, and all vectors are perpendicular to H.	
 Truncated Elliptical Cone (tec)	-Vertex, V -Height vector, H -Major axis for the ellipse -Minor axis for the ellipse -Ratio for the ellipse Stored as a **tgc,** where the length of vector C equals the length of vector A × a ratio and the length of vector D equals the length of vector B × the same ratio.	

 Truncated Right Cone (trc)	-Vertex, V -Height vector, H -Radius of base -Radius of top Stored as a **tgc,** where vectors A, B, C, and D are perpendicular to H, vectors A and B have equal lengths, the length of vector C equals the length of vector A × a ratio, and the length of vector D equals the length of B × the same ratio.	
 Torus (tor)	-Vertex, V (center of hole) -Normal direction for the plane of the ring -Radius 1 (radius from V to center of tube) -Radius 2 (radius of tube)	
 Elliptical Torus (eto)	-Vertex, V -Normal vector -Radius of revolution, r -Vector C (the major axis of the ellipse) -Elliptical semi-minor axis (magnitude of the semi-minor axis of the ellipse), D The magnitude of C must be greater than that of D.	
 Elliptical Hyperboloid (ehy)	-Vertex, V -Height vector, H -Vector A -Scalar b (the magnitude of the perpendicular vector, B) -Apex to asymptote distance, c The length of A is greater than that of B.	

 Elliptical Paraboloid (epa)	-Vertex, V -Height vector, H -Vector A -Scalar b (which is the magnitude of the perpendicular vector B)	
 Right Hyperbolic Cylinder (rhc)	-Vertex, V -Height vector, H -Vector B -Rectangular half width, r, -Apex to asymptote distance, c	
 Right Parabolic Cylinder (rpc)	-Vertex, V -Height vector, H -Vector B -Rectangular half width, r	
 Particle (part)	-Vertex, V -Height vector, H -Radius at vertex V end of particle, v -Radius at opposite end of particle, h	

	There are additional shapes available in MGED that are not listed in the preceding table. In general, they are for more advanced modeling, are still in development, or have significant performance implications. They include the following: bot, dsp, extrude, grip, half, pipe, hf, joint, nmg, poly, sketch

Intentionally Left Blank

3ptarb 150
accept (*see also* apply *and* OK) ... 48, 73, 87, 89, 117
adc 153, 155, 171, 172, 202
advanced settings 126
ae (*see also* azimuth and elevation)... 20, 155
aet 230, 246, 247
aip 183, 236
air 227, 232
ambient (*see also* light) 62, 126
anim_command 158
anim_keyread 214
animmate 149, 156
apply 6, 87, 89, 117
arb (*see* arbitrary convex polyhedron)
arb4 ..... 183, 192, 256
arb5 ..... 183, 192, 256
arb6 ..... 183, 192, 256
arb7 ..... 183, 192, 256
arb8 ..... 68, 94, 183, 192, 256, 257
arbitrary convex polyhedron 94, 150, 156, 157, 158, 185
arbn 184
area 158, 227
arot 159, 212
ars 184
articulation/animation 185
aspect 202, 246, 247
assembly (*see also* group *and* combination) 38, 44, 45, 133, 144
asymptote 259, 260
autosize 226
axes 18, 21, 158, 231
axis 18, 156, 171, 258
azimuth 15, 18, 19, 155, 246
background color 53, 111
bev 160, 222
bindings 21, 152, 160
blast 33, 34
blocking 240
boolean 31, 37, 38, 51, 71, 132, 158, 161, 164, 177
brightness 64, 126
BRL-CAD ..... iii, v, 31, 158
bump 124, 193
camera 13
cancel (*see also* dismiss, reject, *and* reset) 117
checker 100, 124, 193
checkerboard 193, 217
classic 150, 227
clear (*see also* blast, fbclear, *and* zap) . 6, 14, 33, 35, 36, 153
close (*see also* exit *and* quit).... 117, 166, 210, 237
cm 225
cmd_win 236, 237
collaborate 236, 237, 238
color (*see also* RGB)... 32, 101, 139, 152 163, 173, 174, 206, 229
comb (*see also* combination) 52, 133, 161, 163, 180, 209, 211
combination (*see also* assembly *and* group) 38, 44, 45, 161, 163
combination editor 42, 61, 71, 74, 87 89, 91, 125
command line.. 1, 4, 8, 20, 107, 108, 251
command window 1, 3, 6, 7, 8, 12, 13, 23, 28
complex 37, 38, 39, 113, 120
con 171, 172
cone 72, 184, 192
constrain (*see also* shift grip) 21, 50
conventions (*see also* naming conventions) 9, 23, 29
coordinate 11, 13, 18, 227, 231
copy (*see also* cp) 50, 104, 137, 195, 205
copyeval 149, 164
copymat 149, 164, 204, 217
cosine 222, 223
cp (*see also* copy) 50, 73, 104, 137, 149, 164
cpi 149, 165
create (*see also* make *and* insert) 3, 6, 23, 25, 48, 115

cursor (*see also* mouse)... 6, 7, 9, 49, 153 171, 228
curves 177
cut..... 115, 131, 136, 139
cutaway 130, 136, 139
cylinder (*see also* right circular cylinder, right hyperbolic cylinder, *and* right parabolic cylinder) 77, 87, 184, 192
database.. 2, 3, 4, 6, 8, 71, 165, 166, 176, 187, 197, 222
dbconcat 149, 167, 173
deactivate 72
delete (*see also* erase *and* kill) 6, 8, 9, 165, 176, 229, 246, 251, 252
dents 108
deprecated 184, 202, 203
diffuse reflectivity..... 61, 62, 67
dismiss (*see also* cancel, reject, *and* reset)..... 117, 160
distadc 189
dlist..... 227
dm 170, 244
dragging 13, 170
draw..... 5, 6, 33, 34, 152, 153, 172, 173 181, 228
draw grid 104, 110
dsp 184, 261
ebm..... 184
edcodes..... 149, 174
edcolor..... 149, 163, 174, 201
edcomb 149, 174, 175
edit..... 20, 21, 22, 49, 87, 88, 89, 95, 104 116, 118, 174, 175, 209, 221
edmater..... 149, 175
ehy (*see* elliptical hyperboloid)
elevation. 15, 16, 19, 155, 156, 157, 230, 246, 247
ell (*see* ellipsoid)
ellipsoid..... 7, 134, 184, 192, 255, 257
elliptical hyperboloid 184, 192, 259
elliptical paraboloid 184, 192, 260
elliptical torus.. 51, 79, 95, 184, 192, 259
emacs..... 2, 8, 251
emission (*see also* light)..... 62
enter (*see* OK)
environment 21, 117, 193, 209, 221
envmap 193
epa (*see* elliptical paraboloid)
equation..... 175, 176, 179
erase (*see also* delete *and* kill)..... 6, 139, 165, 176
eto (*see* elliptical torus)
exit 9, 29, 150, 177
extinction (*see also* light).. 61, 62, 64, 67
extrude..... 149, 178, 193, 261
eye 155, 162, 178, 190, 196, 207, 227, 228, 230, 231, 247
eye_pt..... 149, 178, 191, 215
facedef..... 149, 178, 179
faceplate 170, 226
fbclear (*see also* zap) 35, 36, 85
fbmbump 193
fbmcolor 193
fbserv..... 227
fmt 206
foci 184, 257
font 83, 204
framebuffer 34, 35, 36, 227
geometry 11, 227
geometry window (*see* graphics window)
get_comb..... 238
get_dm_list..... 238
get_edit_solid..... 236
get_more_default 236, 238
gift 241
glass..... 63, 71, 124, 193
glob_compat_mode..... 162
glossier (*see also* light) 62
graphics window. 1, 2, 4, 5, 6, 11, 12, 13 33, 35, 36, 89, 124
group (*see also* assembly *and* combination) 38, 45, 133, 161, 180
GUI (*see* graphical user interface)
graphical user interface.. 2, 3, 4, 5, 6, 87, 88, 117, 181, 226, 237, 239, 244,
gui_destroy..... 236, 239
help..... 124

hex_code 167, 168, 169
hierarchy (*see also* tree) 208, 227
hist..... 182, 239
hist_add 182
history 9, 182, 239
hook..... 236, 241
hook_cmd..... 241
horizontal (*see also* azimuth) 89, 154, 188, 189
host..... 160, 231
hot_key..... 227
hv..... 154
ident... 161, 163, 173, 174, 175, 183, 185 189, 201, 208, 210, 232, 234
ident_number 185
idents 182, 183, 233
illuminate 183, 202, 236
inactive 35
inherit 33, 241
inmem 209
insert (*see also* create *and* make) .. 23, 25 29, 114, 229, 252
interlay (*see also* framebuffer)..... 34, 35
intersection (*see also* boolean)..... 31, 38, 39, 40, 43, 44, 132, 208, 228
is_toplevel 159
journal 186
keep 186
keyframes 214
keypoint..... 186, 187, 200, 224, 227
keystrokes 8
keysym 227
keysymdef 227
keyword..... 157
kill (*see also* delete *and* erase) 8, 165, 176, 187, 188
killall 149, 187, 188
killtree 149, 187, 188
knob..... 153, 171, 188, 189, 244
l_muves 149, 192, 209
labelvert..... 149, 190
language 107
libbu 168, 169
library..... 152, 157, 182, 190
librt..... 168, 169
light ... 44, 62, 63, 64, 101, 124, 126, 194 232
list..... 7, 8, 185, 189, 191, 192, 221, 223 224, 226, 229, 239
loadtk..... 149, 190
local2base..... 219
los..... 80, 161, 174, 175, 185, 210, 241
ls (*see also* list)..... 8, 150, 191, 221
make (*see also* create *and* insert) ... 5, 23, 25, 192, 193, 240
make solid 6
make_bb 149, 193
make_name 236, 239, 240
mater (*see also* material). 32, 33, 83, 163 193, 201, 211, 217, 233
material 31, 37, 45, 83, 130, 161, 174, 175, 183, 185, 189, 210, 232
material_code 174, 175, 185, 210
matrix 158, 185, 194, 202
memdebug..... 169
menu..... 6, 14, 35, 41, 51, 87, 95, 134
mged prompt 1
midpoint 48
mirface 149, 195
mirror 59, 62, 124, 134, 194, 195
mirror reflectance..... 58, 59
model2objview..... 219
model2view..... 219, 236, 241
modes 20, 104, 110
mouse (*see also* cursor)... 20, 21, 89, 124 152, 170, 171, 172, 227
mouse_behavior 227
move (*see* mv, mvall, *and* translate)
move faces 94
mrot 149, 195
multipane..... 20, 104
mv 149, 195, 196
mvall 149, 195, 196, 201
name 41
naming conventions 29
nirt 149, 196, 206, 207, 231
nmg (*see n*-manifold geometry)
n-manifold geometry..... 160, 179, 180,

192, 196, 197, 217, 228, 261
OK (*see also* accept *and* apply) 3, 117
opaque 58
opendb 149, 197, 198
operator 38, 39, 40, 41, 44, 46, 132
optical 63
orientation 5, 198, 215, 219
origin 151, 188, 227, 229, 230
orot 149, 198, 202
oscale 149, 198, 202
overlay (*see also* framebuffer) 34, 35
parallelepiped 184, 193
part (*see* particle)
particle 98, 184, 192, 260
path 3, 4, 164, 166, 177, 183, 189, 190 193, 197, 199, 216, 217, 218, 236
permute 149, 200
perspective 171, 213, 228
pigment (*see also* color) 101, 124
pixel 35
plot 198, 199, 200, 201
poly 149, 197, 201, 261
polybinout 149, 201
polyhedron 94, 184
prcolor 149, 163, 174, 201
prefix 167, 169, 173, 180, 196, 201
primitive .. 5, 37, 176, 179, 218, 233, 255
projection 124, 171, 203
prompt 1, 2, 4, 5, 150, 179, 183, 223
putmat 149, 204, 205, 217
q (*see* quit)
qorot 149, 205
qray 149, 205, 206, 207
quaternion 198, 230, 247
query_ray 149, 207
quit (*see also* exit) 1, 9, 177, 205, 207, 237, 238
qvrot 149, 207
rateknobs 226
raytrace 31, 34, 42, 43, 213, 227
raytrace control panel 34, 35, 36
rcc (*see* right circuluar cylinder)
rcodes 149, 208, 232
rec (*see* right elliptical cylinder)
rectangular parallelepiped ... 12, 114, 115 184, 257
reflectance (*see also* light) 55, 59, 62, 64
refraction (*see also* light) 63
region 31, 37, 40, 45, 130, 161, 174, 175, 185, 189, 201, 208, 210, 220, 232, 234
regular_expression 177
reject (*see also* cancel, dismiss, and reset) 87, 88, 89, 186, 202
release 4, 115, 125, 152, 159, 210
rendering 83
reset (*see also* cancel, dismiss, and reject) 49, 88, 117, 155, 242
restore 202, 205, 207
rfarb 149, 211
RGB (*see also* color) 101, 163, 164, 175, 177, 207
rhc (*see* right hyperbolic cylinder)
right circular cylinder .. 12, 23, 24, 26, 27 28, 29, 48, 87, 116, 184, 192, 258
right elliptical cylinder 258
right hyperbolic cylinder ... 184, 192, 260
right parabolic cylinder 184, 192, 260
rmater 149, 211, 233
rotate 21, 73, 195, 198, 202, 203, 205, 212, 227, 231, 232
rotobj 149, 198, 199, 212
rpc (*see* right parabolic cylinder)
rpp (*see* rectangular parallelpiped)
rrt 149, 213, 214
rt (*see* raytrace)
rtcheck 149, 214
rtrans 194
savekey 149, 212, 214, 215
scale. 21, 49, 89, 100, 171, 188, 189, 198 199, 202, 203, 219, 234
scene 44, 124, 126
sedit 202
set H (*see also* scale) 48, 87
setview 149, 216, 217
shader ... 32, 56, 61, 62, 66, 83, 124, 125, 175, 193, 194, 217, 232, 233
shape (*see* primitive)

shell 2, 150, 184
shift grip 13, 20, 21, 22
shininess (*see also* light) 62, 64
showmats..... 149, 217, 218
snap 242
solid (*see* primitive)
solid editor 49, 87, 105
solid selection..... 50
specular reflectivity..... 62, 63, 64, 67
sph (*see* sphere)
sphere 2, 5, 7, 24, 192, 255, 257
spin (*see also* rotate) 6
square 157, 158, 159, 216, 229
srot..... 203
sscale 203
stack (*see also* shader) 125, 194
stacker 113, 124, 125, 127
stop (*see also* exit *and* quit) 6
subtract..... 141, 177
subtraction (*see also* boolean). 31, 38, 39 40, 44, 132, 208
svb 236, 244
sxy 203
t_muves 149, 209, 225
tabinterp 215
terminal window 1, 2, 11, 55
tec (*see* truncated elliptical cone)
tessellation..... 222
texture 124, 193, 194, 200
tgc (*see* truncated general cone)
title (*see also* naming conventions)..... 4, 204, 222
tolerance 158, 222, 223
tops 166, 223
tor (*see* torus)
torus..... 12, 184, 192, 259
translate .. 21, 70, 73, 153, 171, 199, 202, 203, 223, 224
transmission (*see also* light)..... 178, 192
transparency 58, 59, 62, 63, 194
trc (*see* truncated right cone)
tree..... 52, 224, 225
triangle 196
truncated elliptical cone 258
truncated general cone 165, 184, 192, 197, 258, 259
truncated right cone..... 72, 184, 192, 259
tty (*see* terminal window)
twist..... 15, 155, 156, 207, 230, 246, 247
undelete (*see also* undo)..... 251
underlay (*see also* framebuffer) ... 34, 35, 134
undo..... 8, 187, 188
union (*see also* boolean) .. 31, 38, 40, 46, 132, 208, 228
units. 5, 77, 162, 188, 218, 219, 225, 226
variable..... 170, 226
vector..... 25, 26, 258
view size..... 14, 15, 218, 219, 230
viewing cube 155, 162, 218
viewset 236, 247
viewsize..... 149, 191, 192, 215, 226
visible 136, 139, 162
vrmgr 149, 231
vrot 149, 231, 232
wireframe 35, 36, 43
wish 29, 215
wmater..... 149, 211, 233
wood..... 194
xrot 231
zap (*see also* blast, clear, *and* fbclear) . 6, 14, 188
zoom..... 14, 15, 203, 227, 234, 235

Intentionally Left Blank

NO. OF COPIES	ORGANIZATION
1	DEFENSE TECHNICAL INFORMATION CENTER DTIC OCA 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	DIRECTOR US ARMY RESEARCH LAB AMSRL D R W WHALIN 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRL D D SMITH 2800 POWDER MILL RD ADELPHI MD 20783-1197
3	DIRECTOR US ARMY RESEARCH LAB AMSRL CI LL 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRL CI AI R 2800 POWDER MILL RD ADELPHI MD 20783-1197
2	DIRECTOR US ARMY RESEARCH LAB AMSRL CI AP 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRL SL C HOPPER WSMR NM 88002-5513
1	NAWC WEAPONS DIVISION CODE 418300D A WEARNER BLDG 91073 1 ADMINISTRATION CIRCLE CHINA LAKE CA 93555-6100
1	NSWC DAHLGREN DIVISION CODE G24 T WASMUND 17320 DAHLGREN RD DAHLGREN VA 22448-5100

NO. OF COPIES	ORGANIZATION
1	USAF 46 OG OGMLV B THORN 104 CHEROKEE AVE EGLIN AFB FL 32542-5600
1	USAF WRIGHT LABORATORY 46TH OG OGM AL AC M LENTZ 2700 D STREET BLDG 22B WRIGHT PAT AFB OH 45433-7605
1	SURVIAC ABERDEEN SATELLITE OFC A LAGRANGE 1003 OLD PHILADELPHIA RD SUITE 3 ABERDEEN MD 21001
5	THE SURVICE ENGNRG CO D KREGEL B STRAUSSER C BOYER M HARDIN M BUTKIEWICZ 1003 OLD PHILADELPHIA RD SUITE 3 ABERDEEN MD 21001
	ABERDEEN PROVING GROUND
4	DIR USARL AMSRL CI LP (305)
220	DIR AMSRL AMSRL SL DR WADE J BEILFUSS AMSRL SL E M STARKS AMSRL SL EC E PANUSKA AMSRL SL EM J FEENEY AMSRL SL B (5 cps) AMSRL SL BA (25 cps) AMSRL SL BD (15 cps) AMSRL SL BE (25 cps) L BUTLER (100 cps) AMSRL SL BG (25 cps) AMSRL SL BN (20 cps)

Intentionally Left Blank

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project(0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 2001	Final, August 2000 - March 2001

4. TITLE AND SUBTITLE

BRL-CAD Tutorial Series: Volume II - Introduction to MGED

5. FUNDING NUMBERS

1L162618AH80

6. AUTHOR(S)

Lee A. Butler, Eric W. Edwards, and Betty J. Schueler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-SL-BE
Aberdeen Proving Ground, MD 21005-5068

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10.SPONSORING/MONITORING AGENCY REPORT NUMBER

ARL-SR-102

11. SUPPLEMENTARY NOTES

*Mr. Edwards performed this work while employed with the SURVICE Engineering Company, and Mrs. Schueler performed this work while employed with Quantum Research International.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT*(Maximum 200 words)*

Since 1979, the U.S. Army Research Laboratory has been developing and distributing the BRL-CAD constructive solid geometry (CSG) modeling package for a wide range of military and industrial applications. The package includes a large collection of tools and utilities including an interactive geometry editor, ray-tracing and generic framebuffer libraries, a network-distributed image-processing and signal-processing capability, and an embedded scripting language.

As part of this effort, a multivolume tutorial series is being developed to assist users in the many features of the BRL-CAD package. The "Introduction to MGED," which is the second volume in the series, is intended to provide new users with a basic understanding of the Multi-Device Geometry Editor (MGED), which is the heart of BRL-CAD. Other volumes focus on installation procedures, advanced features, and programming.

14. SUBJECT TERMS

BRL-CAD, computer-assisted design, modeling and simulation, constructive solid geometry, solid modeling

15. NUMBER OF PAGES

261

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL