Geomview Manual

Geomview Version 1.4.1

for Silicon Graphics Workstations

November 20, 1993

Mark Phillips

What Is Geomview?

Geomview, pronounced gé-om-view, is an interactive program for viewing and manipulating geometric objects, written by staff members of the Geometry Center. It can be used as a standalone viewer for static objects or as a display engine for other programs which produce dynamically changing geometry. It runs on Silicon Graphics (SGI) IRIS workstatons and NeXT workstations. This manual describes SGI Geomview version 1.4.1. Although we do not yet have a manual for the NeXT version, much this of manual applies to that version as well as to the SGI version.

Geomview and this manual are available for free via anonymous ftp on the Internet from host '`geom.umn.edu`' (IP address 128.101.25.35). Permission is granted to make copies of this manual.

If you have questions or comments about Geomview or this manual, you can email them to '`software@geom.umn.edu`'. We are always glad to hear from users. There is also a '`geomview-users`' mailing list for announcements regarding Geomview and for Geomview users to communicate with each other. If you use Geomview please send an email note to '`geomview-users-request@geom.umn.edu`' requesting to be added to this mailing list.

Authors

Tamara Munzner, Stuart Levy, and Mark Phillips are the original authors of Geomview. Celeste Fowler, Charlie Gunn, and Nathaniel Thurston also made significant contributions. Daniel Krech and Scott Wisdom did the NeXTStep and RenderMan port, and Daeron Meyer is currently working on a port to X windows.

Mark Phillips wrote this manual, with substantial help from Stuart Levy and Tamara Munzner. Countless Geomview users have also been of great help by reading it and pointing out mistakes.

Let Us Hear From You

We are very interested in hearing about how you are using Geomview. Think of Geometry Center software as a new kind of shareware: you share your science and successes with us, and we share our software and support with you. The Geometry Center is funded by the National Science Foundation, and it is important that we be able to report to NSF the ways in which our software is being used.

If you use Geomview, please send us a letter telling us what you are doing with it. We need to know:

1. What you are working on - an abstract of your work would be fine.
2. How Geomview has helped you, for example, by increasing your productivity or allowing you to do things you could not do before. In particular, if you feel that Geomview has had a direct bearing on your work, please tell us about this.

Please send the letter either via email to `register@geom.umn.edu`, or via regular mail to the address below.

Moreover, if you use Geomview or other Geometry Center software in your work, we encourage you to cite its use in your publications.

Thank you!

```
Software Development Group
Geometry Center
1300 South 2nd St, Suite 500
Minneapolis, MN  55454
USA
```

1 Overview

Geomview's main purpose is to display objects whose geometry is given, allowing interactive control over details such as point of view, speed of movement, appearance of surfaces and lines, and so on. Geomview can handle any number of objects and allows both separate and collective control over them.

The simplest way to use Geomview is as a standalone viewer to see and manipulate objects. It can display objects described in a variety of file formats. It comes with a wide variety of example objects, and you can create your own objects.

You can also use Geomview to handle the display of data coming from another program that is running simultaneously. As the other program changes the data, the Geomview image reflects the changes. Programs that generate objects and use Geomview to display them are called *external modules*. External modules can control almost all aspects of Geomview. The idea here is that many aspects of the display and interaction parts of geometry software are independent of the geometric content and can be collected together in a single piece of software that can be used in a wide variety of situations. The author of the external module can then concentrate on implementing the desired algorithms and leave the display aspects to Geomview. Geomview comes with a collection of sample external modules, and this manual describes how to write your own.

Geomview represents the current state of an ongoing effort at the Geometry Center to provide interactive geometry software that is particularly appropriate for mathematics research and education. In particular, Geomview can display things in hyperbolic and spherical space as well as Euclidean space.

Geomview allows multiple independently controllable objects and cameras. It provides interactive control for motion, appearances (including lighting, shading, and materials), picking on an object, edge or vertex level, snapshots in SGI image file or Renderman RIB format, and adding or deleting objects is provided through direct mouse manipulation, control panels, and keyboard shortcuts.

Geomview supports the following simple data types: polyhedra with shared vertices (.off), quadrilaterals, rectangular meshes, vectors, and Bezier surface patches of arbitrary degree including rational patches. Object hierarchies can be constructed with lists of objects and instances of object(s) transformed by one or many 4x4 matrices. Arbitrary portions of changing hierarchies may be transmitted by creating named references.

Geomview can display Mathematica graphics output.

2 Tutorial

This chapter leads you through some of the basics of using Geomview. Work through this chapter in front of a workstation where you can try out the examples given here to get a feel for what you can do with Geomview.

To start Geomview, login to the computer and get a shell window. A shell window is a window in which you can type unix commands; the prompt in the window usually ends with a '%'. In the shell window (the mouse cursor must be in the window) type the following (`RET` here means hit the "Enter" key):

```
geomview tetra dodec RET
```

This command starts up Geomview and loads two example objects, a tetrahedron and a dodecahedron. After a few seconds three windows should appear as shown in Figure 1.

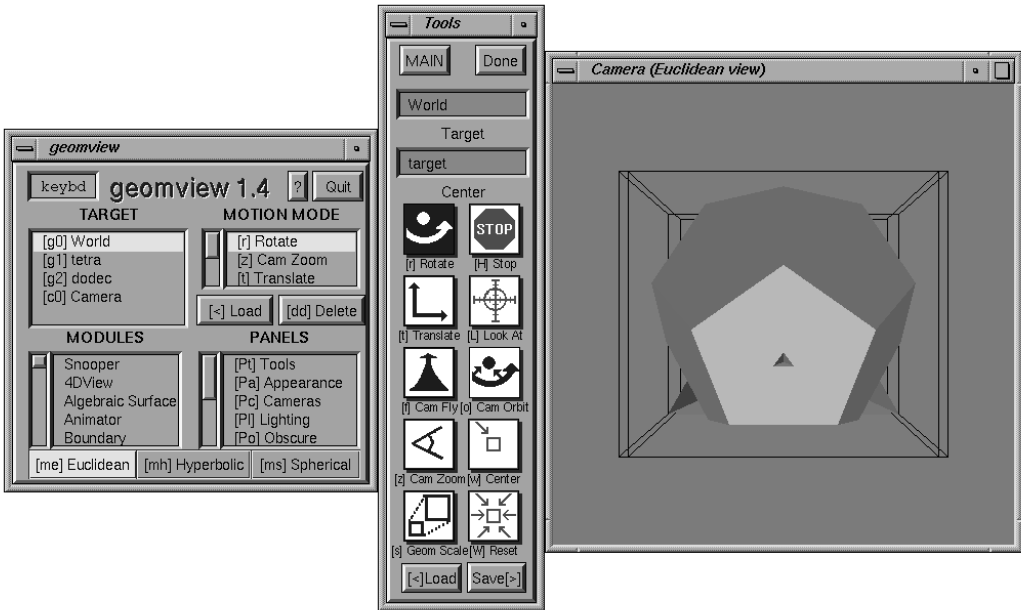


Figure 1: Initial Geomview display

The panel on the left is Geomview's main control panel; it's called the *Main* panel. The skinny panel in the middle is the *Tools* panel and is for selecting different kinds of motions. The window on the right is the camera window and in it you see a large tetrahedron and a dodecahedron which is partially obscured by the tetrahedron.

Geomview has lots of panels but by default it displays only these three. We'll describe some aspects of these and a couple of the others in this tutorial. You can read more about these and other panels in the later chapters of this manual.

Put the mouse cursor in the camera window and press down and hold the left mouse button. Now, while holding down the button, slowly move the mouse around. You should see the picture rotate in the direction you move the mouse. If you lift up on the mouse button while moving the mouse, the picture continues rotating. To stop it, hold the mouse very still and click down and up on the left mouse button.

Geomview uses the *glass sphere* model for mouse-based motion. This means you are supposed to think of the object as being inside an invisible sphere and the mouse cursor is a gripper outside the sphere. When you hold down the left mouse button, the gripper grabs the sphere; when you let go of the button, the gripper releases the sphere. Moving the mouse while holding the button down causes the sphere (and hence the object) to move in the same direction as the mouse.

In addition to the two solids you should also see two wireframe boxes in the camera window. These are the "bounding boxes" of the two objects. By default Geomview puts a bounding box around each object that it displays so that you have an idea of how large it is.

Notice that as you move the mouse around the tetrahedron and dodecahedron move as a unit. That is because by default what you are actually moving is the "World". To move an individual object instead of the whole world, move the mouse cursor to the *TARGET* browser in the *Main* panel. Click (any button) on the word *tetra*. This makes the tetrahedron be the "target object". Now move the cursor back to the camera window and you can rotate just the tetrahedron.

The motion that you have been applying up to now has been rotation, because that is the motion mode that is selected in the *Tools* panel. To translate instead, click on the *Translate* button. Now when you move the mouse in the camera window while holding down the left button, the tetrahedron (which should still be the target object from before) will translate in the direction you move the mouse. Notice that you can translate it beyond the edge of the window as long as you keep holding the left mouse button down. If you lift up on the mouse button while moving the mouse, the tetrahedron will keep going. It moves rather rapidly so it is very easy to lose track of where it is.

If you accidentally lose the tetrahedron by translating it too far out of the view of the window, you can get it back by clicking on the *Center* button in the *Tools* panel. This causes it to come back to its initial position.

Click on the *Center* button to bring the tetrahedron home, and then translate it off to one side so that you can completely see the dodecahedron.

Your world now has two objects in it that are beside each other. You should see the dodecahedron in the middle of the window and maybe part of the tetrahedron off to one side. Go back to the *TARGET* browser in the *Main* panel and click on "World" to select the whole world again. Now click on the *Look At* button in the *Tools* panel. You should see something like Figure 2 — the dodecahedron and the tetrahedron in the middle of the window next to each other. The *Look At* button positions the camera in such a way that the target object is centered in the window.

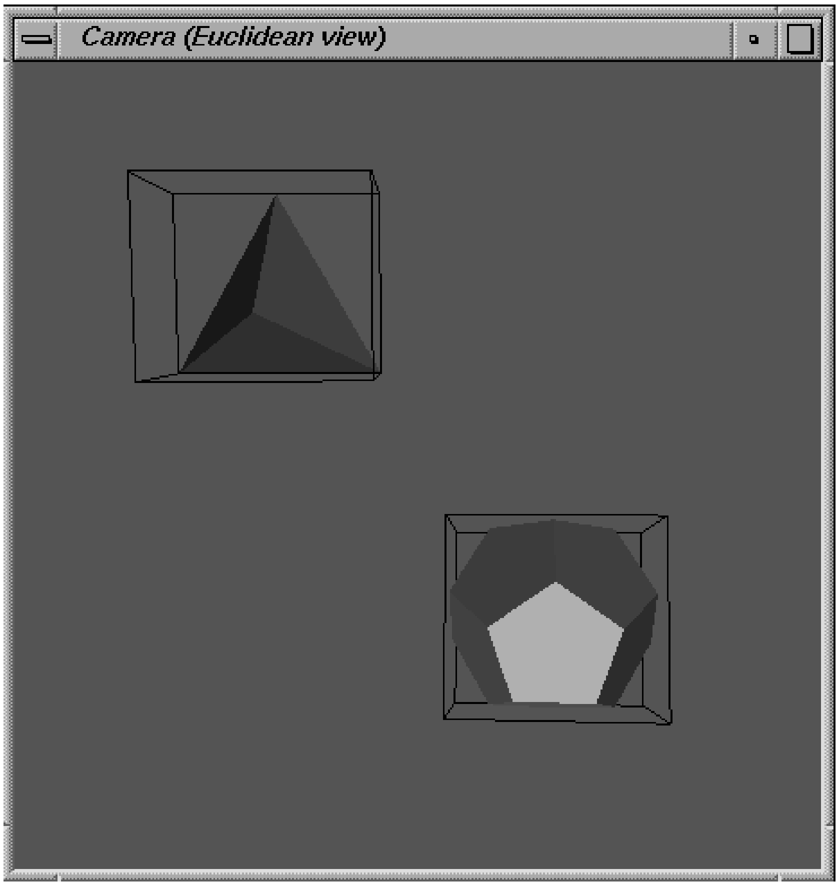
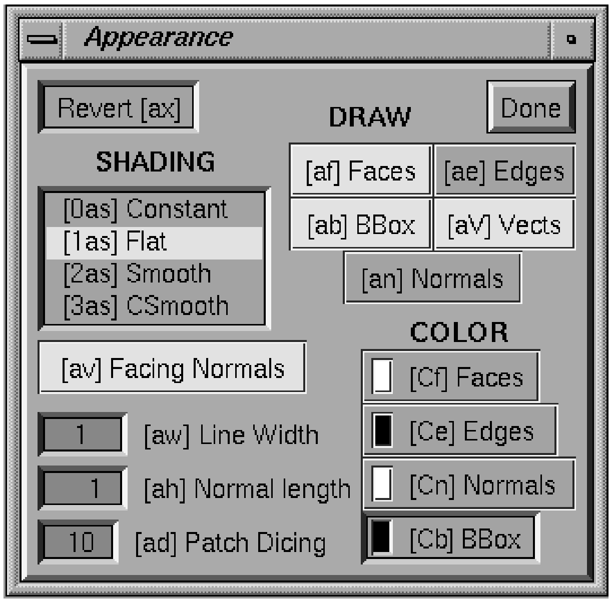


Figure 2: Looking at the world

Now put the cursor over the middle of the dodecahedron and double-click the right mouse button. This means click it down-and-up two times in rapid succession. Notice that the dodecahedron becomes the target object; you can see this in the *TARGET* browser in the *Main* panel. Double-clicking the right mouse button on an object is another way to make it the target object.



1 [aw] Line Width
1 [ah] Normal length
10 [ad] Patch Dicing

Figure 3: The Appearance Panel

Go to the *PANELS* browser in the *Main* panel and click on the word *Appearance*. This brings up the *Appearance* panel. When it appears, if it partially obscures another Geomview window you can move it off to one side by dragging its frame with the middle mouse button down.

The *Appearance* panel lets you control various things about the way Geomview draws objects. In the upper right corner, under the *Done* button are buttons labeled *[af] Faces* and *[ae] Edges*. Click on the *[ae] Edges* one, and notice that Geomview is now drawing the edges of the dodecahedron. Click on it again and the edges go away. Click several times and watch the edges come and go. When you've had enough of this, leave the edges on and click the *[af] Faces* button. This toggles the faces on and off. Click the button again to turn them back on.

Now click on the *[Cf] Faces* button under the word *COLOR*. A color chooser panel like the one in Figure 4 should appear.

Put the cursor in the color hexagon in this panel and hold down the left mouse button. Slowly move the mouse around. This drags the little black point around to choose a new color for the dodecahedron. The previous colors were specified in the file '`dodec`' that you loaded when we started Geomview. The color that you specify with the color panel overrides the old colors. You can adjust the intensity of the color with the *Intensity* slider. When you find a color that you like, click the *OK* button.

Figure 4: Color Chooser Panel

Now put the cursor somewhere over the gray background and double-click the right mouse button; this picks "World" as the target object. Click the *Look At* button to look at the world again.

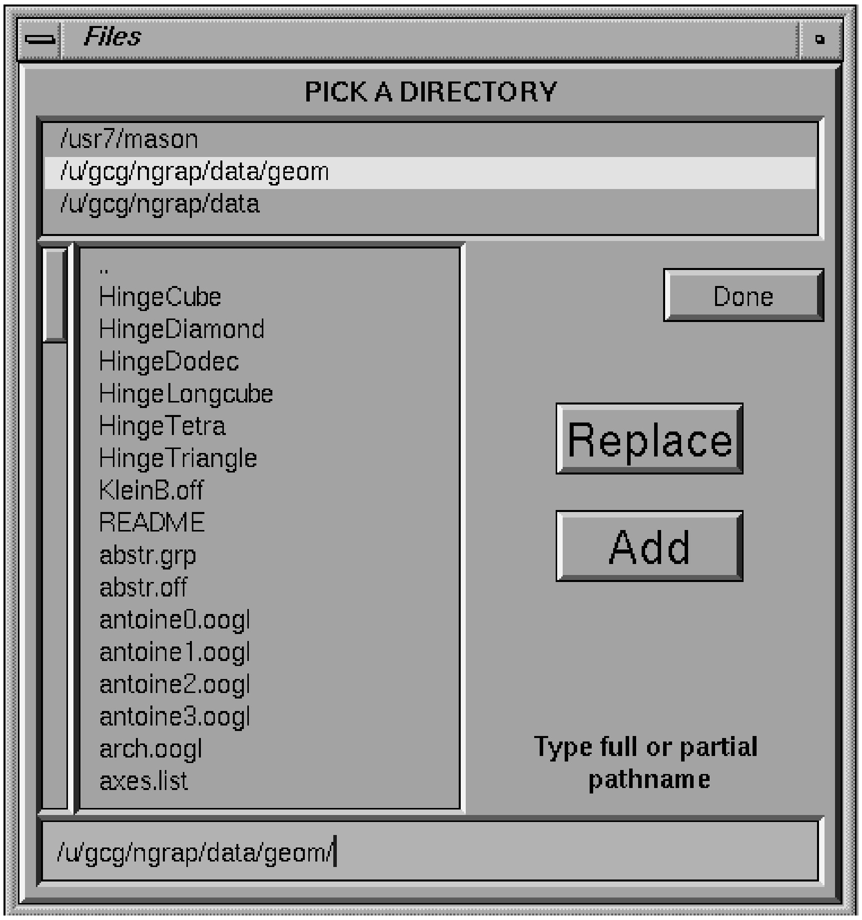
Notice that in the *Appearance* panel the settings of the buttons have changed from the way you left them with the dodecahedron. That's because the *Appearance* panel always displays the settings for the target object, which is now the world, which still has its default settings.

Click on the *[ab] BBox* button under the word *Draw*. The bounding boxes go away. Now put the cursor back in the camera window. At the keyboard, type the keys `a b`. Notice that the bounding boxes come back. `a b` is the keyboard shortcut for the bounding box toggle button; the string "[ab]" appears on the button to indicate this. Most of Geomview's buttons have keyboard shortcuts that you can use instead if you want. This is useful once you are familiar with Geomview and don't want to have to move around among lots of panels.

Now select the tetrahedron, either by double-clicking the right mouse button on it, or by selecting "tetra" in the *TARGET* browser. Then click on the *Delete* button in the *Main* panel. The tetrahedron should disappear. This is how you get rid of an object.

You can also load objects from within Geomview. Click on the *Load* button in the main panel. The *Files* panel will appear.

Near the top of this panel is a browser with three lines in it; the second line is a directory with lots of Geomview example files in it. Click on that second line. Your *Files* panel should then look something like Figure 5. Scroll down in the list of files until you see '`tref.off`'. Click on that line,



/usr7/mason
/u/gcg/ngrap/data/geom
/u/gcg/ngrap/data

..
HingeCube
HingeDiamond
HingeDodec
HingeLongcube
HingeTetra
HingeTriangle
KleinB.off
README
abstr.grp
abstr.off
antoine0.oogl
antoine1.oogl
antoine2.oogl
antoine3.oogl
arch.oogl
axes.list

/u/gcg/ngrap/data/geom/

Figure 5: The Files Panel

and then click on the *Add* button. A large trefoil-shaped tube will appear in your window. Click the *Done* button in the *Files* panel to dismiss the panel.

Now click on the *Reset* button in the *Tools* panel. This causes everything to return to its home position. You should see something like Figure 6 at this point: a dodecahedron and a trefoil knot.

Play around with the trefoil knot and the dodecahedron. Experiment with some of the other buttons in the *Tools* panel. Try coloring the trefoil knot with the *Appearance* panel.

For a tutorial on how to create your own objects to load into Geomview, see file '`doc/oogltour`' distributed with Geomview ('`/u/gcg/ngrap/doc/oogltour`' on the Geometry Center system). The things in that file will be incorporated into a future version of this manual.

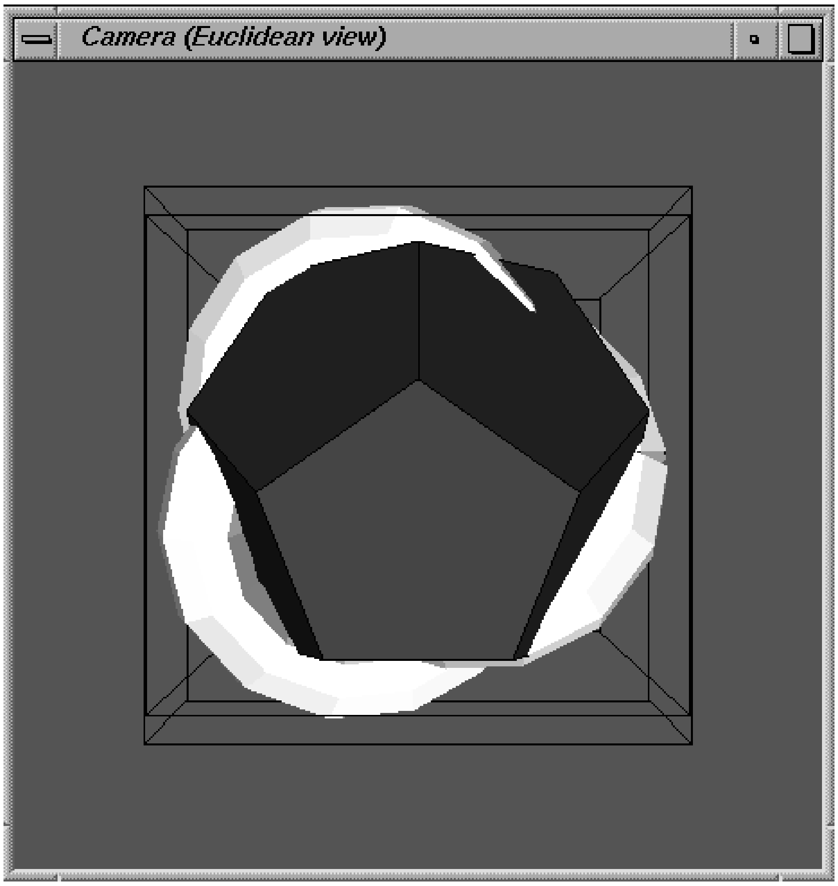


Figure 6: Trefoil and Dodecahedron

3 Interaction

This chapter describes how you interact with Geomview through the mouse and keyboard.

3.1 Starting Geomview

The usual way to start Geomview is to type `geomview RET` in a shell window (RET means hit the "Enter" key). It may take Geomview a few seconds to start up; one or more windows will appear and you can begin interacting with Geomview immediately.

It is also possible to specify actions for Geomview to perform at startup time by giving arguments in the shell command line. See Section 3.2 [Command Line Options], page 11.

3.2 Command Line Options

Here are the command line options that Geomview allows:

‘`-b` *r g b*’ Set the window background color to the given *r g b* values.

‘`-c` *file*’ Interpret the gcl commands in *file*, which may be the special symbol ‘`-`’ for standard input. For a description of gcl, See Chapter 7 [GCL], page 86.

‘`-c` *command*’

Commands may also be supplied literally, as in

```
-c "(ui-panel main off)"
```

Since *command* includes parentheses, which have special meaning to the shell, *command* must be quoted.

‘`-wins` *nwins*’

Causes Geomview to initially display *nwins* camera windows.

‘`-wpos` *width*`,`*height*`[@`*xmin*`,`*ymin*`]`’

Specifies the initial location and size of the first camera window. The values for *width*, *height*, *xmin*, and *ymin* are in screen (pixel) coordinates.

‘`-wpos -`’ You are prompted for placement of the first camera window.

‘`-M` *objectname*’

Display (possibly dynamically changing) geometry sent from the `geomstuff` program. This actually listens to the named pipe ‘`/tmp/geomview/objectname`’; you can achieve the same effect with the shell commands:

```
mkdir /tmp/geomview
mknod /tmp/geomview/objectname p
```

(assuming the directory and named pipe don't already exist), then executing the gcl command: `(geometry objectname < /tmp/geomview/objectname)`

'`-Mc` *pipename*'

Like '`-M`' above, but expects gcl commands, rather than OOGL geometry data, on the connection.

'`-nopanels`'

Start up displaying no panels, only graphics windows. Panels may be invoked later as usual with the `Px` keyboard shortcuts or with the `ui-panel` command.

'`-e` *module*'

Start an external module; *module* is the name associated with the module, appearing in the main panel's Applications browser, as defined by the `emodule-define` command.

'`-start` *module args* ...'

Like -e but allows you to pass arguments to the external module. "–" signals the end of the argument list; the "–" may be omitted if it would be the last argument on the Geomview command line.

'`-run` *shell-command args* ...'

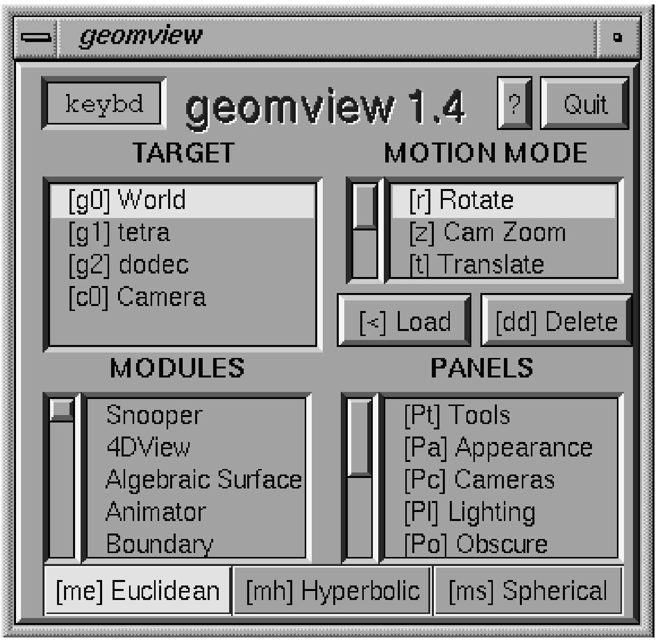
Like -start but takes the pathname of executable of the external module instead of the module's name.

3.3 Basic Interaction: The Main Panel

Normally when you invoke Geomview, three windows appear: the *Main* panel, the *Tools* panel, and one camera window. Geomview has many other windows but most things can be done with these three and so by default the others do not appear. This section of the manual introduces some basic concepts that are used throughout the rest of the manual and describes the *Main* panel.

Geomview can display an arbitrary number of objects simultaneously. The *TARGET* browser in the *Main* panel displays a list of all the objects that Geomview currently knows about. This browser has a line for each object that you have loaded, plus some lines for other objects. One of the other objects is called `World` and corresponds to the all the currently loaded objects, treated as if they were one object. Most of the operations that you can do to one object, such as applying a motion or changing a color, can also be done to the "World" object.

The *TARGET* browser also has an entry for each camera. By default there is only one camera; it is possible to add more of them via the *Camera* panel. Geomview treats cameras in much the



The Main Panel

same way as it does geometric objects. For example, you can move cameras around and add them and delete them just as with geometric objects. Cameras do not usually show up in the display as an object that you see. Each camera has a separate camera window which displays the view as seen by that camera. (It is possible for each camera to display a geometric representation of other cameras; See Section 3.7 [Cameras], page 27.)

Because Geomview treats cameras and geometric objects very similarly, the term *object* in this documentation is used to refer to either one. When we need to distinguish between the two kinds of objects, we use the term *geom* to denote a geometric object and the word *camera* to denote a camera.

The object which is selected (highlighted) in the *TARGET* browser is called the *target* object. This is the object that receives any actions that you do with the mouse or keyboard. You can change the target object by selecting a different line in the *TARGET* browser. Another way to change the target object is to put the mouse cursor directly over a geom in a camera window and rapidly double-click the right mouse button. This process is called *picking*; the picked object becomes the new target.

Geomview objects are all known by two names, both of which are shown in the *TARGET* browser. The first name given there, which appears in square brackets ([ ]), is a short name assigned by Geomview when you load the object. It consists of the letter ‘g’ for geoms and ‘c’ for cameras, followed by a number. The second name is a longer more descriptive name; by default this is the name of the file that the object was loaded from. The two names are equivalent as far as Geomview is concerned; at any point where you need to specify a name you can give either one.

To manipulate an object, make sure you that the object you want to move is the target object, and put the mouse cursor in a camera window. Motions are applied by holding down either the left or middle mouse button and moving the mouse. There are several different motion "modes", each for applying a different kind of motion. The *MOTION MODE* browser in the *Main* panel indicates the current motion mode. The default is "Rotate". You can change the current motion mode by selecting a new one in the *MOTION MODE* browser, or by using the *Tools* panel. For more information about motion modes, See Section 3.5 [Mouse Motions], page 17.

The *Load* button on the *Main* panel brings up the *Files* panel for loading a file. The file can contain either a geom, a camera, or gcl commands. For details, See Section 3.4 [Loading], page 15.

The *Delete* button causes the target object to be deleted. Geomview selects another object to be the new target. You can delete cameras as well as geoms in this way. If you hit the *Delete* button while the target object is "World", Geomview deletes all geoms.

The *Panels* browser on the *Main* panel lists all the Geomview panels. Click on a panel's entry to bring that panel up.

The *Modules* browser lists Geomview external modules. An external module is a separate program that interacts with Geomview to extend its functionality. For information on external modules, See Chapter 6 [Modules], page 64.

The three buttons at the bottom of the *Main* panel, labeled *Euclidean*, *Hyperbolic*, and *Spherical*, allow you to change the geometry of the space that Geomview displays. By default *Euclidean* is selected. For details about using *Hyperbolic* and *Spherical* spaces, See Chapter 8 [Non-Euclidean Geometry], page 104.

Most actions that you can do through Geomview's panels have equivalent keyboard shortcuts so that you can do the same action by typing a sequence of keys on the keyboard. This is useful for advanced users who are familiar with Geomview's capabilities and want to work quickly without having to have lots of panels cluttering up the screen. Keyboard shortcuts are usually indicated in square brackets ([ ]) near the corresponding item in a panel. For example, the keyboard shortcut for *Rotate* mode is 'r'; this is indicated by "[r]" appearing before the word "Rotate" in the *MOTION MODE* browser. To use this keyboard shortcut, just hit the `r` key while the mouse cursor is in any Geomview window. Do not hit the `RET` key afterwards.

Some keyboard shortcuts consist of more than one key. In these cases just type the keys one after the other, with no `RET` afterwards. Keyboard shortcuts are case sensitive.

The *keyboard* field in the upper left corner of the *Main* panel echos the current state of keyboard shortcuts.

The button labeled *?* near the top right corner of the *Main* panel causes Geomview to print out a list of all keyboard shortcuts to standard output.

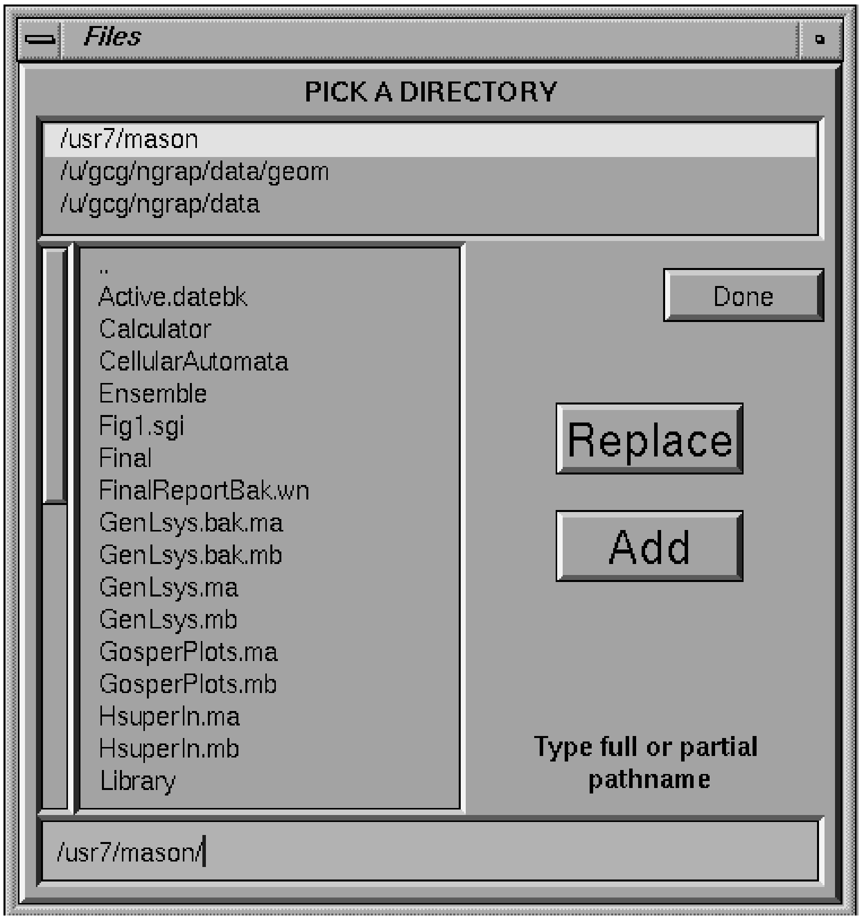
The *Quit* button on the main panel terminates Geomview.

3.4 Loading Objects Into Geomview

There are several ways to load an object into Geomview.

the *Files* panel

> If you click the *Load* button in Geomview's *Main* panel, the *Files* panel will appear.
>
> This panel lets you select a file from a variety of directories. The short browser at the top of the panel shows the selected directory, and the longer browser beneath it shows the files in that directory. To select a file, click on it. After a file is selected, you can load it into Geomview by clicking either the *Add* or *Replace* button. The *Add* button causes Geomview to load the file as a new object; a new entry will appear in the *Objects* browser in the *Main* panel, and all objects currently loaded will remain unchanged. The *Replace* button causes Geomview to replace the current object with the object in the selected file. If the current object is the *World*, Geomview deletes all objects and loads the one in the file as a new object. If the current object is a camera, the *Replace* button has no effect.
>
> If the file that you select contains gcl commands rather than an OOGL object, the *Add* and *Replace* buttons both cause Geomview to interpret the commands in the file. For more information about this, See Chapter 7 [GCL], page 86.
>
> When the *Files* panel first appears, the directory selected in the directory browser is the current directory — the one from which you invoked Geomview. The file browser shows *all* the files in this directory, including ones that are not Geomview files. If you try to load a file that doesn't contain either an OOGL object or Geomview commands, Geomview will print out an error message.
>
> The directory browser also lists a second and third directory in addition to the current directory. The second one, which ends in '`data/geom`', is the Geomview example data



/usr7/mason/

The Files Panel

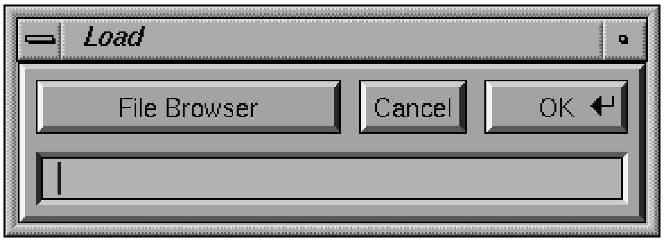
directory. This contains a wide variety of sample objects. It also contains several subdirectories. In particular, the '`hyperbolic`' and '`spherical`' subdirectories have sample hyperbolic and spherical objects, respectively. Directory entries in the browser look just like file entries; to view a subdirectory, click on it.

The third directory shown in the directory browser, which ends in '`geom`', contains several subdirectories with other Geomview files in them. These are used less frequently than the ones in the '`data/geom`' directory.

You can change the list of directories shown the *Files* panel's directory browser by using the `set-load-path` command; see Chapter 7 [GCL], page 86.

the `<` keyboard shortcut:

If you type `<` in any Geomview window, the *Load* panel will appear. This is a small version of the *Files* panel; it contains a text field in which you can enter the name of a file to load. After typing the name of the file to load, type `RET`; Geomview will load the file as if you had loaded it with the *Add* button in the *Files* panel. If, after bringing



The Load Panel

up the small load panel with `<`, you decide you want to use the larger *Files* panel after all, press the *Files Browser* button.

geometry loading commands:

The `load`, `geometry`, `new-geometry`, and `read` gcl commands allow you to load an object into Geomview; See Chapter 7 [GCL], page 86.

3.5 Using the Mouse to Manipulate Objects

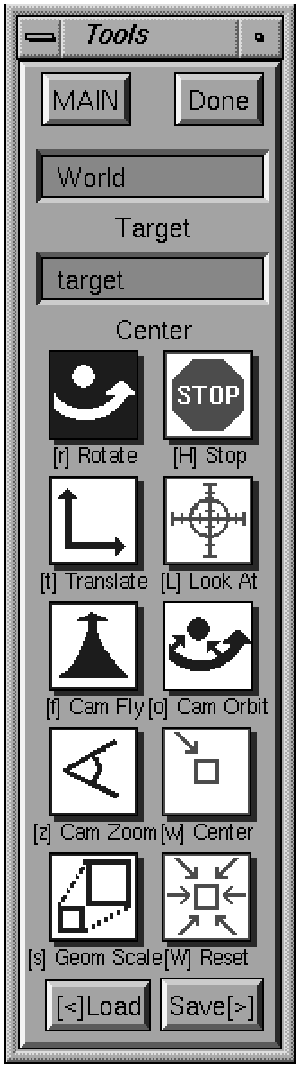
Geomview lets you manipulate objects with the mouse. There are six different mouse motion modes: *Rotate*, *Translate*, *Cam Fly*, *Cam Zoom*, *Geom Scale*, and *Cam Orbit*. The tools panel has a button for each of these modes; to switch modes, click on the corresponding button. You can also select these through the *Motion Mode* browser on the *Main* panel.

This section describes basic mouse interaction. There are some more advanced features available on the *Obscure* panel. For details, see Section 3.9 [Command Obscure], page 32.

Each of the motion modes uses a common paradigm for how the motion is applied. In particular, each depends on the current *target* object and the current *center* object. These are explained in the following paragraphs.

The current target object is shown in the *Target* field in the *Tools* panel. This is the same as the selected object in the *TARGET OBJECTS* browser in the *Main* panel, and you can change it by either selecting a new object in the browser, by typing a new entry in the field, or by picking an object in a camera window by double-clicking the right mouse button with the cursor over the object.

The current center object is shown in the *Center* field in the *Tools* panel. Its default value is the special word "target", which means that the center object is whatever the target object is. You can change the center to any object by typing it in the *Center* field. The origin of the center object is held fixed in *Rotate* and *Orbit* modes. Normally the center object is one of the existing geoms



The Tools Panel

listed in the *TARGET OBJECTS* browser, and the actual center of rotations is the origin of that object's coordinate system. It is possible, however, to select an arbitrary point of interest on an object as the center. For details, see Section 3.5.1 [Point of Interest], page 21.

You apply a mouse motion by holding down either the left or middle mouse button with the cursor in a camera window and moving the mouse. Most of the modes have *inertia*, which means that if you let go of the button while moving the mouse, the motion will continue. It may be helpful to imagine the mouse cursor as being a gripper; when you hold a mouse button down, it grips the target object and you can move it. When you let go of the mouse button, the gripper releases the object. Letting go of the mouse button while moving the mouse is like throwing the object — the object continues moving independent of the mouse. Inertia can be turned off; see the button on the *Obscure* panel.

Most of the mouse motions have a slow motion version which you get by holding down the shift key while doing the motion as usual. This is useful for finer control.

You can pick any point on an object (not just its origin) as the center of motion by holding down the shift key while clicking the right mouse button; this chooses a point of interest.

Rotate In *Rotate* mode, hold the left mouse button down to rotate the target object about the center object. Rotation proceeds in the direction that you move the mouse. Specifically, the axis of rotation passes through the origin of the center object, is parallel to the camera view plane, and is perpendicular to the direction of motion of the mouse. When the center is "target", this means that the target object rotates about its own origin.

The middle mouse button in *Rotate* mode rotates the target object about an axis perpendicular to the view plane.

Translate In *Translate* mode, hold the left mouse button down to translate the target object in the direction of mouse motion. The middle mouse button translates the target along an axis perpendicular to the view plane.

In Euclidean space, the center object is essentially irrelevant for translations. In hyperbolic and spherical spaces, where translations have a unique axis, this axis is chosen to go through the origin of the center object.

Cam Fly *Cam Fly* is a crude flight simulator that lets you fly around the scene. It works by moving the camera. Move the mouse while holding the left mouse button down to point the camera in a different direction. To move forward or backward, hold down the middle button and move the mouse vertically. Both of these motions have inertia; typically the easiest way to fly around a scene is to give the camera a slight forward push by letting go of the middle button while moving the mouse upward, and then using the left button to steer.

Cam Fly affects the camera window that the mouse is in; it ignores the target object and the center object.

Cam Orbit

Cam Orbit mode lets you rotate the current camera around the current center. The left mouse button does this rotation. The middle mouse button in *Cam Orbit* mode is the same as it is in *Cam Fly* mode: it moves the camera forward or backward.

In general *Cam Orbit* does not move the target object, although if the current camera is selected as the target and the center is also the target, it will rotate that camera about itself just as in *Cam Fly* mode.

Cam Zoom

Cam Zoom mode lets you change the current camera's field of view with the mouse; hold the left mouse button down and move the mouse to change it. The numeric value of the field of view is shown in the *FOV* field in the *Camera* panel.

Geom Scale

Geom Scale mode lets you enlarge or shrink a geom. It operates on the target object if that object is a geom. If the target is a camera, *Geom Scale* operates on the geom that was most recently the target object. Moving the mouse while holding down the left mouse button scales the object either up or down, depending on the direction of mouse motion. The center of the applied scaling transformation is the center object.

Scaling is meaningful only in Euclidean space; attempts to scale are ignored in other spaces.

Geom Scale mode does not have inertia.

The *Stop*, *Look At*, *Center*, and *Reset* buttons on the *Tools* panel perform actions related to motions but do not change the current motion mode.

Stop The *Stop* button causes all motions to stop. It affects all moving objects; not just the target object. It's keyboard shortcut is `H`.

The keyboard command `h`, which does not correspond to a panel button, stops the current motion for the target object only.

Look At The *Look At* button causes the current camera to be moved to a position such that it is looking at the target object, and such that the target object more or less fills the window.

The Look At command is unreliable in non-Euclidean spaces.

Center The *Center* button undoes the target object's transformation, moving it back to its home position, which is where it was when you originally loaded it into Geomview.

Reset The *Reset* button stops all motion and causes all objects to move back to their home positions.

The other four buttons on the *Tools* panel are:

MAIN This button brings up the *Main* panel in case you have dismissed it or in case it is buried underneath other panels.

Done This button dismisses the *Tools* panel. You can bring the panel back by selecting it in the *More Panels* browser in the *Main* panel, or via the `P t` keyboard shortcut.

Load This is the same as the *Load* button on the *Main* panel; it brings up the *Files* panel.

Save This is the same as the *Save* button on the *Main* panel; it brings up the *Save* panel.

3.5.1 Selecting a Point of Interest

It is sometimes useful to specify a particular point on some object in a geomview window as the center point for mouse motions. You can do this by shift-clicking the right mouse button (i.e. click it once while holding down the shift key on the keyboard) with the cursor over the desired point. This point then becomes the *point of interest*. The point of interest must be on an existing object.

Selecting a point of interest simplifies examining a small portion of a larger object. Shift-right-click on an interesting point, and select *Orbit* mode. Use the middle mouse button to approach, and the left mouse to orbit the point, examining the region from different directions.

When you have selected a point of interest, the current center object changes to an object named "CENTER", which is an invisible object located at the point of interest. In addition, mouse motions for the window in which you made the selection are adjusted so that the point of interest follows the mouse.

You can change the point of interest at any time by selecting a new one by shift-clicking the right mouse button again. You can cancel the point of interest altogether by shift-clicking the right mouse button with the cursor on the background (i.e. not on any object). This changes the center object back to its default value, "target".

The object named "CENTER", which serves as the center object for the point of interest, is a special kind of geom called an "alien". It does not appear in the *TARGET OBJECTS* browser. By default it has no geometry associated with it and hence is invisible. You can, however, explicitly give it some geometry using a GCL command, causing it to appear. Use the `geometry` command for this: `(geometry CENTER` *geometry*`)`, where *geometry* is any valid geometry. For example, `(geometry CENTER { < xyz.vect })` causes the file '`xyz.vect`', which is one of the standard example files distributed with geomview, to be used at the geometry for CENTER.

What actually happens internally when you select a point of interest is that the center is set to the object called CENTER, and that object is positioned at the point of interest. In addition, in order for mouse motions to track the point of interest, the current camera's focal length is set to be the distance from the camera to the point of interest. You can accomplish this via GCL with the following commands:

```
(if (real-id CENTER) nil (new-alien CENTER ))
(ui-center CENTER)
(transform-set CENTER universe universe translate x y z)
(merge camera cam-id { focus d })
```

where (x,y,z) are the (universe) coordinates of the point of interest, and d is the distance from that point to the current camera, *cam-id*. The first command above creates the "alien" CENTER if it does not yet exist.

3.6 Changing the Way Things Look

Geomview uses a hierarchy of appearances to control the way things look. An *appearance* is a specification of information about how something should be drawn. This can include many things such things as color, lighting, material properties, and more. Appearances work in a hierarchal manner: if a certain appearance property, for example face color, is not specified in a particular object's appearance, that object is drawn using that property from the parent appearance. If both the parent and the child appearance specify a property, the child's setting takes precedence unless the parent appearance is set to override.

Every geom in Geomview has an appearance associated with it. There is also an appearance associated with the "World" geom, which serves as the parent of each individual geom's appearance. Finally, there is an global "base" appearance, which is the parent of the World appearance.

The base appearance specifies reasonable values for all appearance information, and by default none of the other appearances specify anything, which means they inherit their values from the base appearance. This means that by default all objects are drawn using the base appearance.

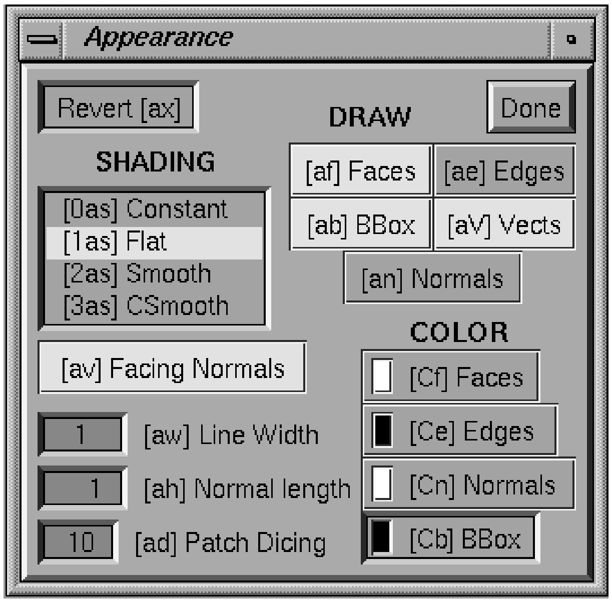
If you change a certain appearance property for a geom, that property is used in drawing that geom. The parent appearance is used for any properties that you do not explicitly set.

Geomview has three panels which let you modify appearances.

3.6.1 The Appearance Panel

The *Appearance* panel lets you change most common appearance properties of the target object.

If the target is an individual geom, then changes you make in the appearance panel apply to that geom's appearance. If the target is the World, then appearance panel changes apply to the World appearance *and* to all individual geom appearances. (Users have found that this is more desirable than having the changes only apply to the World appearance.) If the target is a camera, then appearance panel changes apply to the geom that was most recently the target.



1 [aw] Line Width

1 [ah] Normal length

10 [ad] Patch Dicing

The Appearance Panel

The five buttons near the upper right corner under the word *Draw* control what parts of the target geom are drawn.

Faces	This button specifies whether faces are drawn.
Edges	This button specifies whether edges are drawn.
BBox	This button specifies whether the bounding box is drawn.
Vects	This button specifies whether VECT objects are drawn. VECTs are a type of OOGL object that represent points and line segments in 3-space; they are distinct from edges of other kinds of objects, and it is sometimes desirable to have separate control over whether they are drawn.
Normals	This button specifies whether surface normal vectors are drawn.

The four buttons under *COLOR* labeled *Faces*, *Edges*, *Normals*, and *BBox* let you specify the color of the corresponding aspect of the target geom. Clicking on one of them brings up a color chooser panel.

The black dot in the center of the hexagon represents the current color; you can move it around by dragging it with the left mouse button down. The slider specifies the intensity. To accept the color that you have chosen, click the *OK* button. To cancel your color selection and return to the previous color, click the *Cancel* button.

Color Chooser Panel

The *SHADING* browser lets you specify the shading model that Geomview uses to paint the target geom.

Constant	Every face of the object is drawn with a constant color which does not depend on the location of the face, the camera, or the light sources. If the object does not contain per-face or per-vertex colors, the diffuse color of the object's appearance is used. If the object contains per-face colors, they are used. If the object contains per-vertex colors, each face is painted using the color of its first vertex.
Flat	Each face of the object is drawn with a color that depends on the relative location of the face, the camera, and the light sources. The color is constant across the face but may change as the face, camera, or lights move.
Smooth	Each face of the object is drawn with smoothly interpolated colors based on the normal vectors at each vertex. If the object does not contain per-vertex normals, this has the same effect as flat shading. If the object has reasonable per-vertex normals, the effect is to smooth over the edges between the faces.
CSmooth	Each face of the object is drawn with exactly the specified color(s), independent of lighting, orientation, and material properties. If the object is defined with per-vertex colors, the colors are interpolated smoothly across the face; otherwise the effect is the same as in Constant shading style.

The *Facing Normals* button on the *Appearance* panel indicates whether or not Geomview should arrange that normal vectors always face the viewer. If a normal vector points away from the viewer the color of the corresponding face or vertex usually is darker than is desired. Geomview can avoid this by using the opposite normal in shading calculations. This is the default. To change it so that the actual normals are used, press this button. This has no effect in constant shading mode.

The three text fields in the lower left corner of the *Appearance* panel are:

Line Width
: The width, in pixels, for lines drawn by Geomview.

Normal Length
: This is actually a scale factor; when normal vectors are drawn, Geomview draws them to have a length that is their natural length times this number.

Patch Dicing
: Geomview draws Bezier patches by first converting them to meshes. This parameter specifies the resolution of the mesh: if *Patch Dicing* is n, then an n by n mesh is used to draw each Bezier patch. if *Patch Dicing* is 1, the resolution reverts to a built-in default value.

The *Revert* button on the *Appearance* panel undoes all settings in the target appearance. This causes the target geom to inherit all its appearance properties from its parent.

The *Appearance* panel's *Override* button determines whether appearance controls should override settings in the objects themselves – for example, setting the face color will affect all faces of objects with multi-colored facets. Otherwise, appearance controls only provide settings which the objects themselves do not specify. By default, *Override* is enabled. This button applies to all objects, and to all appearance-related panels.

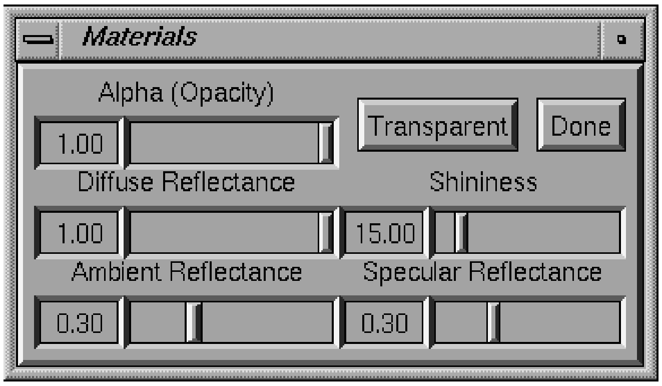
3.6.2 The Materials Panel

The *Materials* panel controls material properties of surfaces. It works with the target object in the same way that the *Appearance* panel does.

Transparent
: This button determines whether transparency is enabled. Geomview does not fully support transparency yet and on some machines it does not work at all.

Alpha
: This slider determines the opacity/transparency when transparency is enabled. 0 means totally transparent, 1 means totally opaque.

Diffuse Reflectance
: This slider controls the diffuse reflectance of a surface. This has to do with how much the surface scatters light that it reflects.



Alpha (Opacity) 1.00
Diffuse Reflectance 1.00
Shininess 15.00
Ambient Reflectance 0.30
Specular Reflectance 0.30

The Materials Panel

Shininess	This slider controls how shiny a surface is. This determines the size of specular highlights on the surface. Lower values give the surface a duller appearance.
Ambient Reflectance	This slider controls how much of the ambient light a surface reflects.
Specular Reflectance	This slider controls the specular reflectance of a surface. This has to do with how directly the surface reflects light rays. Higher values give brighter specular highlights.
Done	This button dismisses the *Materials* panel.

3.6.3 The Lighting Panel

The *Lighting* panel controls the number, position, and color of the light sources used in shading.

Intensity 1.00

The Lighting Panel

The *Lighting* panel is different from the *Appearance* and *Material* panels in that it always works with the base appearance. This is because it usually makes sense to use the same set of lights for drawing all objects in your scene.

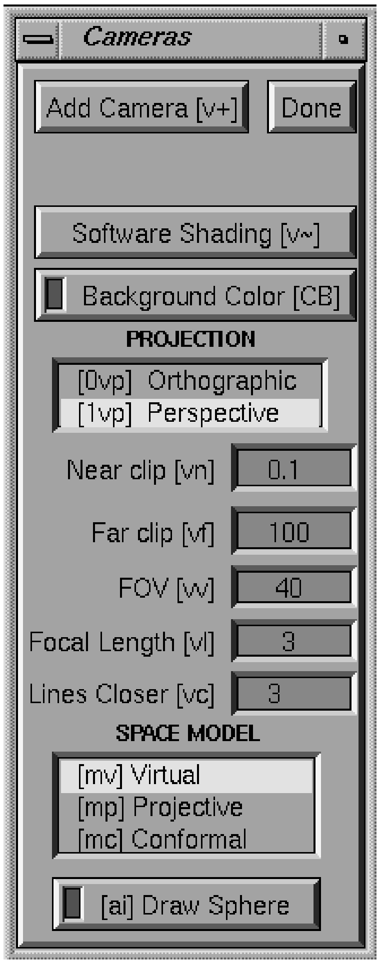
LIGHTS	The *LIGHTS* browser shows the currently selected light. Changes made using the other widgets on this panel apply to this light. There is always at least one light, the ambient light.
Intensity	This slider controls the intensity of the current light.
Color	This button brings up a color chooser which lets you select the color of the current light.
Add	This button adds a light.
Delete	This button deletes the current light.
Show Lights	This button lets you see and change the positions of the light sources in a camera window. Each light is drawn as long cylinder which is supposed to remind you of a light beam. When you click on the *Show Lights* button Geomview goes into "light edit" mode, during which you can rotate current light by holding down the left mouse button and moving the mouse. Lights placed in this way are infinitely distant, so what you are changing is the angular position. Click on the *Show Lights* button again to return to the previous motion mode and to quit drawing the light beams.
Done	This button dismisses the *Lighting* panel.

Geomview's *Appearance*, *Materials*, and *Lighting* panels are constructed to allow you to easily do most of the appearance related things that you might want to do. The appearance hierarchy that Geomview supports internally, however, is very complex and there are certain operations that you cannot do with the panels. The Geomview command language (gcl) provides complete support for appearance operations. In particular, the `merge-baseap` command can be used to change the base appearance (which, except for lighting, cannot be changed by Geomview's panels). The `merge-ap` command can be used to change an individual geom's appearance. Appearances can also be specified in OOGL files; for details, see Section 4.1.8 [Appearances], page 42.

3.7 Cameras

A camera in Geomview is the object that corresponds to a camera window. By default there is only one camera, but it is possible to have as many as you want. You can control certain aspects of the way the world is drawn in each camera window via the *Cameras* panel.

If the target object is a camera, the *Cameras* panel affects that camera. If the target object is not a camera, the *Cameras* panel affects the *current camera*. The current camera is the camera of the window that the mouse cursor is in, or was in most recently if the cursor is not in a camera



Near clip [vn] 0.1
Far clip [vf] 100
FOV [vv] 40
Focal Length [vl] 3
Lines Closer [vc] 3

The Cameras Panel

window. Thus, if you use the keyboard shortcuts for the actions in the *Cameras* panel while the cursor is in a camera window, the actions apply to that camera, unless you have explicitly selected another camera.

Add Camera

> Clicking on this button causes Geomview to create a new camera. The new camera's window appears, and an entry for it appears in the *TARGET* browser of the *Main* panel.

Software Shading

> This button controls whether Geomview does shading calculations in software. The default is to let the hardware handle them, and in Euclidean space this is almost certainly best because it is faster. In hyperbolic and spherical space, however, the shading calculations that the hardware does are incorrect. Click this button to turn on correct but slower software shading.

Background Color

This button brings up a color chooser which you can use to set the background color of the camera's window.

PROJECTION

This browser lets you pick between perspective and orthogonal projection for this camera.

Near clip This determines the distance in world coordinates of the near clipping plane from the eye point. It must be a positive number.

Far clip This determines the distance in world coordinates of the far clipping plane from the eye point. It must be a positive number and in general should be larger than the *Near clip* value.

FOV This is the camera's field of view, measured in its shorter direction. In perspective mode, it is an angle in degrees. In orthographic mode, it is the linear size of the field of view. This number can be modified with the mouse in *Cam Zoom* mode.

Focal Length

The focal length is intended to suggest the distance from the camera to an imaginary plane of interest. Its value is used when switching between orthographic and perspective views (and during stereo viewing), so as to preserve apparent size of objects lying at the focal distance from the camera.

Lines Closer

This number has to do with the way lines are drawn. Normally Geomview's z-buffering algorithm can get confused when drawing lines that lie exactly on surfaces (such as the edges of an object); due to machine round-off error, sometimes the lines appear to be in front of the surface and sometimes they appear behind it. The *Lines Closer* value is a fudge factor — Geomview nudges all the lines that it draws closer to the camera by this amount. The number should be a small integer; try 5 or 10. 0 turns this feature off completely. Choosing too large a value will make lines visible even though they should be hidden.

SPACE MODEL

This determines the model used to draw the world. It is most useful in hyperbolic and spherical spaces. You probably don't need to touch this browser if you stay in Euclidean space. For more information about these models, see Chapter 8 [Non-Euclidean Geometry], page 104.

Virtual This is the default model and represents the natural view from inside the space.

Projective The projective model of hyperbolic and spherical space. Geoms move under isometries of the space, and cameras move by Euclidean motions. By default in the projective model, the Euclidean unit sphere is drawn. In hyperbolic space this is the sphere at infinity. In Euclidean space the projective model is the same as the virtual model except that the sphere is drawn by default.

Conformal

The conformal model of hyperbolic and spherical space. Geoms move under isometries of the space, and cameras move by Euclidean motions. In Euclidean space, the conformal model amounts to inverting everything in the unit sphere.

Draw Sphere

This controls whether Geomview draws the unit sphere. By default the unit sphere appears in the projective and conformal models. In hyperbolic space this is the sphere at infinity. In spherical space it is the equatorial sphere.

Done This button dismisses the *Cameras* panel.

3.8 Saving your work

Geomview's *Save* panel lets you store Geomview objects and other information in files that you can read back into Geomview or other programs.

The Save Panel

To use the *Save* panel you select the desired format in the browser next to the word *Save*, enter the name of the object you want to save in the text field next to the word *for*, and enter the name of the file you wish to save to in the long text field next to the word *in*. You can then either hit `RET` or click on the *OK* button. When the file has been written, the *Save* panel disappears. If you want to dismiss the *Save* panel without writing a file, click the *Cancel* button.

If you specify '`-`' as the file name, Geomview will write the file to standard output, i.e. in the shell window from which you invoked Geomview.

The possible formats are given below. The kind of object that can be written with each format is given in parentheses.

Commands (any object)
: This write a file of gcl commands containing all information about the object. Loading this file later will restore the object as well as all other information about it, such as appearance, transformations, etc.

Geometry alone (geom)
: This writes an OOGL file containing just the geometry of the object.

Geometry [in world] (geom)
: This writes an OOGL file containing just the geometry of the object, transformed under Geomview's current transformation for this object. Use this if you have moved the object from its initial position and want to save the new position relative to the world.

Geometry [in universe] (geom)
: This writes an OOGL file containing just the geometry of the geom, transformed under both the object's transformaton and the world's transformation.

RMan [->tiff] (camera)
: Writes a RenderMan file which when rendered creates a tiff image.

RMan [->frame] (camera)
: Writes a RenderMan file which when rendered causes an image to appear in an Iris window.

SGI snapshot (camera)
: Write an SGI raster file. A bell rings when the snapshot is complete.

Camera (camera)
: Writes an OOGL file of a camera.

Transform [to world] (any object)
: Writes an OOGL transform file giving Geomview's transform for the object.

Transform [to universe] (any object)
: Writes an OOGL transform file giving a transform which is the composition of Geomview's transform for the object and the transform for the world.

Window (camera)
: Writes an OOGL window file for a camera.

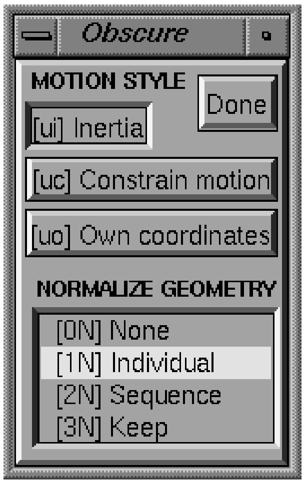
Panels
: Writes a gcl file containing commands which record the state of all the Geomview panels. Loading this file later will restore the positions of all the panels.

3.9 The Commands and Obscure Panels

The *Commands* panel lets you type in a gcl command. When you hit `RET`, Geomview interprets the command and prints any resulting output or error messages on standard output. You can edit the text and hit `RET` as many times as you like, in general, whenever you hit `RET` with the cursor in the *Commands* panel, Geomview tries to interpret whatever text you have typed in the text field as a command.

The Commands Panel

The *Obscure* panel is for relatively obscure things that don't really belong on any of the other panels. In the present version of Geomview, the *Obscure* panel includes the *NORMALIZE GEOMETRY* browser, which controls the kind of geometry normalization that Geomview does, and several buttons affecting motion style.



The Obscure Panel

Normalization is a kind of scaling; Geomview can scale an object so that it fits within a certain region. The main point of normalization is to allow you to easily view all of an object without having to worry about how big it is. We are gradually replacing Geomview's normalization feature with more robust camera positioning features. In general, the best way to make sure you are seeing all of an object is to use the *Look At* button on the *Tools* panel. Normalization may be completely replaced by this and other features in a future version of Geomview.

Normalization is a property that applies to each geom separately. The *NORMALIZE GEOMETRY* browser affects the normalization property of target geom. If the target geom is "World", it affects all geoms.

None	Do no normalization.
Individual	Normalize this geom to fit within a unit sphere.
Sequence	This resembles "Individual", except when an object is changing. Then, "Individual" tightly fits the bounding box around the object whenever it changes and normalizes accordingly, while "Sequence" normalizes the union of all variants of the object and normalizes accordingly.
Keep	This leaves the current normalization transform unchanged when the object changes. It may be useful to apply "Individual" or "Sequence" normalization to the first version of a changing object to bring it in view, then switch to "Keep".

The *Motion Style* controls include the following buttons.

[ui] Inertia

Normally, moving objects have inertia: if the mouse is still moving when the button is released, the selected object continues to move. When *Inertia* is off, objects cease to move as soon as you release the mouse.

[uc] Constrain Motion

It's sometimes handy to move an object in a direction aligned with a coordinate axis: exactly horizontally or vertically. Selecting *Constrain Motion* changes the interpretation of mouse motions to allow this; approximately-horizontal or approximately-vertical mouse dragging becomes exactly horizontal or vertical motion. Note that the motion is still along the X or Y axes of the camera in which you move the mouse, not necessarily the object's own coordinate system.

[uo] Own Coordinates

It's sometimes handy to move objects with respect to the coordinate system where they were defined, rather than with respect to some camera. While *Own Coordinates* is selected, all motions are interpreted that way: dragging the mouse rightward in translate mode moves the object in its own +X direction, and so on. May be especially useful in conjunction with the *Constrain Motion* button.

3.10 Keyboard Shortcuts

Most actions that you can do through Geomview's panels have equivalent keyboard shortcuts so that you can do the same action by typing a sequence of keys on the keyboard. This is useful for advanced users who are familiar with Geomview's capabilities and want to work quickly without having to have lots of panels cluttering up the screen. Keyboard shortcuts usually are indicated in square brackets ([ ]) near the corresponding item in a panel. For example, the keyboard shortcut for *Rotate* mode is 'r'; this is indicated by "[r]" appearing before the word "Rotate" in the *MOTION MODE* browser. To use this keyboard shortcut just hit the `r` key while the mouse cursor is in any Geomview window. Do not hit the `RET` key afterwards.

Some keyboard shortcuts consist of more than one key. In these cases just type the keys one after the other, with no `RET` afterwards. Keyboard shortcuts are case sensitive. You can cancel a multi-key keyboard shortcut that you have started by typing any invalid key, for example `^`.

Keyboard commands apply while the cursor is in any camera window and most control panels.

Many keyboard shortcuts allow numeric arguments which you type as a prefix to the command key(s). For example, the shortcut for *Near clip* in the camera panel is `v n`. To set the near clip plane to "0.5", type `0.5 v n`. Commands that don't take a numeric prefix toggle or reset the current value.

Most commands allow one of the following selection prefixes. If none is provided the command applies to the target object.

`g`	world geom
`g#`	#'th geom
`g*`	All geoms
`c`	current camera
`c#`	#'th camera
`c*`	All cameras

For example, `g 4 a f` means toggle the face drawing of object *g4*.

The text field in the upper left corner of the *Main* panel shows the state of the current keyboard shortcut.

In addition to the keyboard shortcuts for the panel commands, there is also a shortcut for picking a target object: type the short name of the object followed by `p`. For example, to select object *g3*, type `g 3 p`. This only works with the short names — the ones that appear in square brackets ([ ]) in the *TARGET* browser of the *Main* panel.

Below is a summary of all keyboard shortcuts.

Draw		
	`af`	Faces
	`ae`	Edges
	`an`	Normals
	`ab`	Bounding Boxes
	`aV`	Vectors
Shading		
	`0as`	Constant
	`1as`	Flat
	`2as`	Smooth
	`3as`	Smooth, non-lighted
	`aT`	allow transparency
Other		
	`av`	eVert normals: always face viewer
	`#aw`	Line Width (pixels)
	`#ac`	edges Closer than faces (try 5-100)
Color		
	`Cf`	faces
	`Ce`	edges
	`Cn`	normals
	`Cb`	bounding boxes
	`CB`	background
Motions		
	`r`	rotate
	`t`	translate
	`z`	zoom FOV
	`f`	fly
	`o`	orbit

	`s`	scale
	`w`	recenter target
	`W`	recenter all
	`h`	halt
	`H`	halt all
	`@`	select center of motion (e.g. `g 3 @`)
	`L`	Look At object
Viewing		
	`0vp`	Orthographic view
	`1vp`	Perspective view
	`vd`	Draw other views' cameras
	`#vv`	field of View
	`#vn`	near clip distance
	`#vf`	far clip distance
	`v+`	add new camera
	`vx`	cursor on/off
	`vb`	backfacing poly cull on/off
	`#vl`	focal length
	`v~`	Software shading on/off
Panels		
	`Pm`	Main
	`Pa`	Appearance
	`Pl`	Lighting
	`Po`	Obscure
	`Pt`	Tools
	`Pc`	Cameras
	`PC`	Commands
	`Pf`	Files
	`Ps`	Save
	`P-`	read commands from tty
Lights		
	`ls`	show lights
	`le`	edit lights
Space		

	`me`	Euclidean
	`mh`	Hyperbolic
	`ms`	Spherical
Model		
	`mv`	Virtual
	`mp`	Projective
	`mc`	Conformal
Other		
	`0N`	normalizaton: none
	`1N`	normalization: each
	`2N all`	normalization: all
	`ui`	motion: Inertia
	`uc`	motion: Constrain to axis
	`uo`	motion: object's Own coordinates
	`<`	
	`Pf`	load geometry/command file
	`dd`	delete target object
	`>`	
	`Ps`	save state to file
	`TV`	NTSC mode toggle
	`p`	pick as target object (e.g. `g 3 p`) With no prefix, selects the object under the mouse cursor (like double-clicking the right mouse)

4 OOGL File Formats

The objects that you can load into Geomview are called OOGL objects. OOGL stands for "Object Oriented Graphics Library"; it is the library upon which Geomview is built.

There are many different kinds of OOGL objects. This chapter gives syntactic descriptions of file formats for OOGL objects.

Examples of most file types live in Geomview's '`data/geom`' directory.

4.1 Conventions

4.1.1 Syntax Common to All OOGL File Formats

Most OOGL object file formats are free-format ASCII — any amount of white space (blanks, tabs, newlines) may appear between tokens (numbers, key words, etc.). Line breaks are almost always insignificant, with a couple of exceptions as noted. Comments begin with # and continue to the end of the line; they're allowed anywhere a newline is.

Binary formats are also defined for several objects; See Section 4.1.6 [Binary format], page 40, and the individual object descriptions.

Typical OOGL objects begin with a key word designating object type, possibly with modifiers indicating presence of color information etc. In some formats the key word is optional, for compatibility with file formats defined elsewhere. Object type is then determined by guessing from the file suffix (if any) or from the data itself.

Key words are case sensitive. Some have optional prefix letters indicating presence of color or other data; in this case the order of prefixes is significant, e.g. `CNMESH` is meaningful but `NCMESH` is invalid.

4.1.2 File Names

When OOGL objects are read from disk files, the OOGL library uses the file suffix to guess at the file type.

If the suffix is unrecognized, or if no suffix is available (e.g. for an object being read from a pipe, or embedded in another OOGL object), all known types of objects are tried in turn until one accepts the data as valid.

4.1.3 Vertices

Several objects share a common style of representing vertices with optional per-vertex surface-normal and color. All vertices within an object have the same format, specified by the header key word.

All data for a vertex is grouped together (as opposed to e.g. giving coordinates for all vertices, then colors for all vertices, and so on).

The syntax is

'*x y z*'	(3-D floating-point vertex coordinates) or
'*x y z w*'	(4-D floating-point vertex coordinates)

optionally followed by

'*nx ny nz*'	(normalized 3-D surface-normal if present)

optionally followed by

'*r g b a*'	(4-component floating-point color if present, each component in range 0..1. The *a* (alpha) component represents opacity: 0 transparent, 1 opaque.)

Values are separated by white space, and line breaks are immaterial.

4.1.4 Surface normal directions

Geomview uses normal vectors to determine how an object is shaded. The direction of the normal is significant in this calculation.

When normals are supplied with an object, the direction of the normal is determined by the data given.

When normals are not supplied with the object, Geomview computes normal vectors automatically; in this case normals point toward the side from which the vertices appear in counterclockwise order.

On parametric surfaces (Bezier patches), the normal at point P(u,v) is in the direction dP/du cross dP/dv.

4.1.5 Transformation matrices

Some objects incorporate 4x4 real matrices for homogeneous object transformations. These matrices act by multiplication on the right of vectors. Thus, if p is a 4-element row vector representing homogeneous coordinates of a point in the OOGL object, and A is the 4x4 matrix, then the transformed point is p' = p A. This matrix convention is common in computer graphics; it's the transpose of that often used in mathematics, where points are column vectors multiplied on the right of matrices.

Thus for Euclidean transformations, the translation components appear in the fourth row (last four elements) of A. A's last column (4th, 8th, 12th and 16th elements) are typically 0, 0, 0, and 1 respectively.

4.1.6 Binary format

Many OOGL objects accept binary as well as ASCII file formats. These files begin with the usual ASCII token (e.g. `CQUAD`) followed by the word `BINARY`. Binary data begins at the byte following the first newline after `BINARY`. White space and a single comment may intervene, e.g.

```
OFF BINARY # binary-format "OFF" data follows
```

Binary data comprise 32-bit integers and 32-bit IEEE-format floats, both in big-endian format (i.e., with most significant byte first). This is the native format for 'int's and 'float's on Sun-3's, Sun-4's, and Irises, among others.

Binary data formats resemble the corresponding ASCII formats, with ints and floats in just the places you'd expect. There are some exceptions though, specifically in the `QUAD` and `OFF` file formats. Details are given in the individual file format descriptions. See Section 4.2.1 [QUAD], page 45, and See Section 4.2.4 [OFF], page 49.

Binary OOGL objects may be freely mixed in ASCII object streams:

```
LIST
{ = MESH BINARY
... binary data for mesh here ...
}
{ = QUAD
1 0 0   0 0 1   0 1 0  0 1 0
}
```

Note that ASCII data resumes immediately following the last byte of binary data.

Naturally, it's impossible to embed comments inside a binary-format OOGL object, though comments may appear in the header before the beginning of binary data.

4.1.7 Embedded objects and external-object references

Some object types (`LIST`, `INST`) allow references to other OOGL objects, which may appear literally in the data stream, be loaded from named disk files, or be communicated from elsewhere via named objects. Gcl commands also accept geometry in these forms.

The general syntax is

```
 <oogl-object>  ::=
[ "{" ]
    [ "define" symbolname ]
    [ "appearance" appearance ]
    [ ["="] object-keyword ...
 | "<" filename
 | ":" symbolname ]
[ "}" ]
```

where "quoted" items are literal strings (which appear without the quotes), [bracketed] items are optional, and `|` denotes alternatives. Curly braces, when present, must match; the outermost set of curly braces is generally required when the object is in a larger context, e.g. when it is part of a larger object or embedded in a Geomview command stream.

For example, each of the following three lines:

```
{ define fred   QUAD 1 0 0  0 0 1  0 1 0  1 0 0 }

{ appearance { +edge } LIST { < "file1" } { : fred } }

VECT 1 2 0   2 0   0 0 0   1 1 2
```

is a valid OOGL object. The last example is only valid when it is delimited unambiguously by residing in its own disk file.

The "`<`" construct causes a disk file to be read. Note that this isn't a general textual "include" mechanism; a complete OOGL object must appear in the referenced file.

Files read using "`<`" are sought first in the directory of the file which referred to them, if any; failing that, the normal search path (set by Geomview's `load-path` command) is used. The default search looks first in the current directory, then in the Geomview data directories.

The ":" construct allows references to symbols, created with `define`. A symbol's initial value is a null object. When a symbol is (re)defined, all references to it are automatically changed; this is a crucial part of the support for interprocess communication. Some future version of the documentation should explain this better...

Again, white space and line breaks are insignificant, and "#" comments may appear anywhere.

4.1.8 Appearances

Geometric objects can have associated "appearance" information, specifying shading, lighting, color, wireframe vs. shaded-surface display, and so on. Appearances are inherited through object hierarchies, e.g. attaching an appearance to a `LIST` means that the appearance is applied to all the `LIST`'s members.

Some appearance-related properties are relegated to "material" and "lighting" substructures. Take care to note which properties belong to which structure.

Here's an example appearance structure including values for all attributes. Order of attributes is unimportant. As usual, white space is irrelevant. Boolean attributes may be preceded by "+" or "-" to turn them on or off; "+" is assumed if only the attribute name appears. Other attributes expect values.

A "*" prefix on any attribute, e.g. "*+edge" or "*linewidth 2", selects "override" status for that attribute.

```
appearance {
  +face               # draw faces of polygons
  -edge               # don't draw edges of polygons
  -transparent        # don't attempt transparency (not implemented)
  +normal             # do draw surface-normal vectors
  normscale .25       # ... with length .25 in object coordinates

  +evert              # do evert polygon normals where needed so as
                      #   to always face the camera

  shading smooth      # or ``shading constant'' or ``shading flat''.
                      # smooth = Gouraud shading; flat = faceted.

  linewidth 3         # lines & edges are 3 pixels wide.

  material {         # Here's a material definition;
                      # it could also be read from a file as in
                      #  ``material < file.mat''

      ka  1.0         # ambient reflection coefficient.
      ambient .3 .5 .3 # ambient color (red, green, blue components)
                      # The ambient contribution to the shading is
                      # the product of ka, the ambient color,
                      # and the color of the ambient light.

      kd  0.8         # diffuse-reflection coefficient.
      diffuse .9 1 .4 # diffuse color.
                        # (In ``shading constant'' mode, the surface
                        # is colored with the diffuse color.)

      ks 1.0          # specular reflection coefficient.
      specular 1 1 1  # specular (highlight) color.
      shininess  25   # specular exponent; larger values give
                      # sharper highlights.

      alpha   1.0     # opacity; 0 = transparent (invisible)
```

```
                           # when transparency is enabled.

           edgecolor   1 1 0  # line & edge color

           normalcolor 0 0 0  # color for surface-normal vectors
       }

       lighting {         # Lighting model

           ambient  .3 .3 .3  # ambient light

           replacelights   # ``Use only the following lights to
                           # illuminate the objects under this
                           # appearance.''
                           # Without "replacelights", any lights
                           # are added to those already in the scene.

                           # Now a collection of real lights
           light {
               color  1 .7 .6      # light color
               position  1 0 .5 0  # light position [distant light]
                                   # given in homogeneous coordinates.
                                   # With fourth component = 0,
                                   # this means a light coming from
                                   # direction (1,0,.5).
           }

           light {                         # Another light.
               color 1 1 1
               position  0 0 .5 1  # light at finite position ...
               location camera     # specified in camera coordinates.
                                   # (Since the camera looks toward -Z,
                                   # this example places the light
                                   # .5 unit behind the eye.)
               # Possible "location" keywords:
               #  global    light position is in world coordinates
               #             This is the default if no location specified.
               #  camera   position is in the camera's coordinate system
               #  local    position is in the coordinate system where
               #                  the appearance was defined
           }
       }                   # end lighting model
     }                     # end appearance
```

There are rules for inheritance of appearance attributes when several are imposed at different levels in the hierarchy.

For example, Geomview installs a backstop appearance which provides default values for most parameters; its control panels install other appearances which supply new values for a few attributes; user-supplied geometry may also contain appearances.

The general rule is that the child's appearance (the one closest to the geometric primitives) wins. So setting an appearance attribute in an object's definition will prevent the viewer controls from affecting that object's display.

4.2 Object File Formats

4.2.1 `QUAD`: collection of quadrilaterals

The conventional suffix for a `QUAD` file is '`.quad`'.

The file syntax is

```
[C][N][4]QUAD  -or-  [C][N][4]POLY    # Key word
vertex  vertex  vertex  vertex  # 4*N vertices for some N
vertex  vertex  vertex  vertex
...
```

The leading key word is `[C][N][4]QUAD` or `[C][N][4]POLY`, where the optional `C` and `N` prefixes indicate that each vertex includes colors and normals respectively. That is, these files begin with one of the words

`QUAD CQUAD NQUAD CNQUAD POLY CPOLY NPOLY CNPOLY`

(but not `NCQUAD` or `NCPOLY`). `QUAD` and `POLY` are synonymous; both forms are allowed just for compatibility with ChapReyes.

Following the key word is an arbitrary number of groups of four vertices, each group describing a quadrilateral. See the Vertex syntax above. The object ends at end-of-file, or with a closebrace if incorporated into an object reference (see above).

A `QUAD BINARY` file format is accepted; See Section 4.1.6 [Binary format], page 40. The first word of binary data must be a 32-bit integer giving the number of quads in the object; following that is a series of 32-bit floats, arranged just as in the ASCII format.

4.2.2 `MESH`: rectangularly-connected mesh

The conventional suffix for a `MESH` file is '`.mesh`'.

The file syntax is

```
[C][N][Z][4][U][u][v]MESH    # Key word
Nu Nv             # Mesh dimensions
                             # Nu*Nv vertices, in format specified
                             # by initial key word
vertex(u=0,v=0)  vertex(1,0)  ... vertex(Nu-1,0)
vertex(0,1) ...    vertex(Nu-1,1)
...
vertex(0,Nv-1) ... vertex(Nu-1,Nv-1)
```

The key word is `[C][N][Z][4][U][u][v]MESH`. The optional prefix characters mean:

'`C`'	Each vertex (see Vertices above) includes a 4-component color.
'`N`'	Each vertex includes a surface normal vector.
'`Z`'	Of the 3 vertex position values, only the Z component is present; X and Y are omitted, and assumed to equal the mesh (u,v) coordinate so X ranges from 0 .. (Nu-1), Y from 0 .. (Nv-1) where Nu and Nv are the mesh dimensions – see below.
'`4`'	Vertices are 4D, each consists of 4 floating values. `Z` and `4` cannot both be present.
'`U`'	Each vertex includes a 3-component texture space parameter. This is not yet implemented and should not be used.
'`u`'	The mesh is wrapped in the u-direction, so the (0,v)'th vertex is connected to the (*Nu*-1,v)'th for all v.
'`v`'	The mesh is wrapped in the v-direction, so the (u,0)'th vertex is connected to the (u,*Nv*-1)'th for all u. Thus a u-wrapped or v-wrapped mesh is topologically a cylinder, while a uv-wrapped mesh is a torus.

Note that the order of prefix characters is significant; a colored, u-wrapped mesh is a `CuMESH` not a `uCMESH`.

Following the mesh header are integers *Nu* and *Nv*, the dimensions of the mesh.

Then follow *Nu***Nv* vertices, each in the form given by the header. They appear in v-major order, i.e. if we name each vertex by (u,v) then the vertices appear in the order

```
(0,0) (1,0) (2,0) (3,0) ...  (Nu-1,0)
(0,1) (1,1) (2,1) (3,1) ...  (Nu-1,1)
...
(0,Nv-1) ...  (Nu-1,Nv-1)
```

A `MESH BINARY` format is accepted; See Section 4.1.6 [Binary format], page 40. The values of *Nu* and *Nv* are 32-bit integers; all other values are 32-bit floats.

4.2.3 Bezier Surfaces

The conventional file suffixes for Bezier surface files are '`.bbp`' or '`.bez`'. A file with either suffix may contain either type of patch.

Syntax:

```
  [ST]BBP -or- [C]BEZ<Nu><Nv><Nd>[_ST]
# Nu, Nv are u- and v-direction
# polynomial degrees in range 1..6
# Nd = dimension: 3->3-D, 4->4-D (rational)
# (The '<' and '>' do not appear in the input.)
# Nu,Nv,Nd are each a single decimal digit.
# BBP form implies Nu=Nv=Nd=3 so BBP = BEZ333.

# Any number of patches follow the header
# (Nu+1)*(Nv+1) patch control points
# each 3 or 4 floats according to header
  vertex(u=0,v=0)  vertex(1,0) ... vertex(Nu,0)
  vertex(0,1)    ... vertex(Nu,1)
  ...
  vertex(0,Nv)     ... vertex(Nu,Nv)

# ST texture coordinates if mentioned in header
  S(u=0,v=0) T(0,0) S(0,Nv) T(0,Nv)
```

```
  S(Nu,0) T(Nu,0) S(Nu,Nv) T(Nu,Nv)

# 4-component float (0..1) R G B A colors
# for each patch corner if mentioned in header
  RGBA(0,0)   RGBA(0,Nv)
  RGBA(Nu,0)  RGBA(Nu,Nv)
```

These formats represent collections of Bezier surface patches, of degrees up to 6, and with 3-D or 4-D (rational) vertices.

The header keyword has the forms `[ST]BBP` or `[C]BEZ<`*Nu*`><`*Nv*`><`*Nd*`>[_ST]` (the '`<`' and '`>`' are not part of the keyword.

The `ST` prefix on `BBP`, or `_ST` suffix on `BEZuvn`, indicates that each patch includes four pairs of floating-point texture-space coordinates, one for each corner of the patch.

The `C` prefix on `BEZuvn` indicates a colored patch, including four sets of four-component floating-point colors (red, green, blue, and alpha) in the range 0..1, one color for each corner.

Nu and *Nv*, each a single digit in the range 1..6, are the patch's polynomial degree in the u and v direction respectively.

Nd is the number of components in each patch vertex, and must be either `3` for 3-D or `4` for homogeneous coordinates, that is, rational patches.

`BBP` patches are bicubic patches with 3-D vertices, so `BBP` = `BEZ333` and `STBBP` = `BEZ333_ST`.

Any number of patches follow the header. Each patch comprises a series of patch vertices, followed by optional (s,t) texture coordinates, followed by optional (r,g,b,a) colors.

Each patch has (*Nu*`+1`)*(*Nv*`+1`) vertices in v-major order, so that if we designate a vertex by its control point indices (u,v) the order is

```
(0,0) (1,0) (2,0) ...  (Nu,0)
(0,1) (1,1) (2,1) ...  (Nu,1)
...
(0,Nv)             ...  (Nu,Nv)
```

with each vertex containing either 3 or 4 floating-point numbers as specified by the header.

If the header calls for ST coordinates, four pairs of floating-point numbers follow: the texture-space coordinates for the (0,0), (Nu,0), (0,Nv), and (Nu,Nv) corners of the patch, respectively.

If the header calls for colors, four four-component (red, green, blue, alpha) floating-point colors follow, one for each patch corner.

The series of patches ends at end-of-file, or with a closebrace if incorporated in an object reference.

4.2.4 OFF Files

The conventional suffix for `OFF` files is '`.off`'.

Syntax:

```
[C][N][4]OFF # Header keyword (optional)
NVertices  NFaces  NEdges   # NEdges not used or checked

x[0]  y[0]  z[0] # Vertices, possibly with colors
     # and/or normals if COFF or NOFF
...
x[NVertices-1]  y[NVertices-1]  z[NVertices-1]

     # Faces
     # Nv = # vertices on this face
     # v[0] ... v[Nv-1]: vertex indices
     # in range 0..NVertices-1
Nv  v[0] v[1] ... v[Nv-1]  colorspec
...
     # colorspec continues past v[Nv-1]
     # to end-of-line; may be 0 to 4 numbers
     # nothing: default
     # j integer: colormap index
     # 3 or 4 integers: RGB[A] values 0..255
# 3 or 4 floats: RGB[A] values 0..1
```

`OFF` files (name for "object file format") represent collections of planar polygons with possibly shared vertices, a convenient way to describe polyhedra. The polygons may be concave but there's no provision for polygons containing holes.

An `OFF` file may begin with the keyword `OFF`; it's

recommended but optional, as many existing files lack this keyword.

Three ASCII integers follow: *NVertices*, *NFaces*, and *NEdges*. Thse are the number of vertices, faces, and edges, respectively. Current software does not use nor check *NEdges*; it needn't be correct but must be present.

The vertex coordinates follow: dimension * *Nvertices* floating-point values. They're implicitly numbered 0 through *NVertices*-1. dimension is either 3 (default) or 4 (specified by the key character `4` directly before `OFF` in the keyword).

Following these are the face descriptions, typically written with one line per face. Each has the form

N *Vert1* *Vert2* ... *VertN* [*color*]

Here *N* is the number of vertices on this face, and *Vert1* through *VertN* are indices into the list of vertices (in the range 0..*NVertices*-1).

The optional *color* may take several forms. Line breaks are significant here: the *color* description begins after *VertN* and ends with the end of the line (or the next # comment). A *color* may be:

nothing	the default color
one integer	index into "the" colormap; see below
three or four integers	RGB and possibly alpha values in the range 0..255
three or four floating-point numbers	RGB and possibly alpha values in the range 0..1

For the one-integer case, the colormap is currently read from the file '`cmap.fmap`' in Geomview's '`data`' directory. Some better mechanism for supplying a colormap is likely someday.

The meaning of "default color" varies. If no face of the object has a color, all inherit the environment's default material color. If some but not all faces have colors, the default is gray (R,G,B,A=.666).

A `[C][N]OFF BINARY` format is accepted; See Section 4.1.6 [Binary format], page 40. It resembles the ASCII format in almost the way you'd expect, with 32-bit integers for all counters and vertex indices and 32-bit floats for vertex positions (and vertex colors or normals if `COFF/NOFF/CNOFF` format).

Exception: each face's vertex indices are followed by an integer indicating how many color components accompany it. Face color components must be floats, not integer values. Thus a colorless triangular face might be represented as

```
int int int int int
3   17  5   9   0
```

while the same face colored red might be

```
int int int int int float float float float
 3  17   5   9   4   1.0   0.0   0.0   1.0
```

4.2.5 VECT Files

The conventional suffix for `VECT` files is '`.vect`'.

Syntax:

```
[4]VECT
NPolylines   NVertices   NColors

Nv[0] ... Nv[NPolylines-1]      # number of vertices
                                        # in each polyline

Nc[0] ... Nc[NPolylines-1]      # number of colors supplied
                                        # in each polyline

Vert[0] ... Vert[NVertices-1]  # All the vertices
                                        # (3*NVertices floats)
```

```
Color[0] ... Color[NColors-1]  # All the colors
                                 # (4*NColors floats, RGBA)
```

`VECT` objects represent lists of polylines (strings of connected line segments, possibly closed). A degenerate polyline can be used to represent a point.

A `VECT` file begins with the key word `VECT` or `4VECT` and three integers: *NLines*, *NVertices*, and *NColors*. Here *NLines* is the number of polylines in the file, *NVertices* the total number of vertices, and *NColors* the number of colors as explained below.

Next come *NLines* integers

Nv[0] Nv[1] Nv[2] ... Nv[NLines-1]

giving the number of vertices in each polyline. A negative number indicates a closed polyline; 1 denotes a single-pixel point. The sum (of absolute values) of the *Nv[i]* must equal *NVertices*.

Next come *NLines* more integers Nc[i]: the number of colors in each polyline. Normally one of three values:

0	No color is specified for this polyline. It's drawn in the same color as the previous polyline.
1	A single color is specified. The entire polyline is drawn in that color.
abs(*Nv[i]*)	Each vertex has a color. Either each segment is drawn in the corresponding color, or the colors are smoothly interpolated along the line segments, depending on the implementation.

The sum of the *Nc[i]* must equal *NColors*.

Next come *NVertices* groups of 3 or 4 floating-point numbers: the coordinates of all the vertices. If the keyword is *4VECT* then there are 4 values per vertex. The first abs(*Nv[0]*) of them form the first polyline, the next abs(*Nv[1]*) form the second and so on.

Finally *NColors* groups of 4 floating-point numbers give red, green, blue and alpha (opacity) values. The first *Nc[0]* of them apply to the first polyline, and so on.

A *VECT BINARY* format is accepted; See Section 4.1.6 [Binary format], page 40. The binary format exactly follows the ASCII format, with 32-bit ints where integers appear, and 32-bit floats where real values appear.

4.2.6 INST Files

The conventional suffix for a `INST` file is '`.inst`'.

An `INST` applies a 4x4 transformation to another OOGL object. It begins with `INST` followed by these sections which may appear in any order:

`geom` *oogl-object*

specifies the OOGL object to be instantiated. See Section 4.1.7 [References], page 41, for the syntax of an *oogl-object*. The keyword `unit` is a synonym for `geom`.

```
transform   ["{"] 4x4 transform ["}"]
```

specifies a single transformation matrix. Either the matrix may appear literally as 16 numbers, or there may be a reference to a "transform" object, i.e.

```
"<" file-containing-4x4-matrix
```

or

```
":" symbol-representing-"transform"-object>
```

Another way to specify the transformation is

`transforms`
oogl-object

The *oogl-object* must be a `TLIST` object (list of transformations) object, or a `LIST` whose members are ultimately `TLIST` objects. In effect, the `transforms` keyword takes a collection of 4x4 matrices and replicates the `geom` object, making one copy for each 4x4 matrix.

If no `transform` nor `transforms` keyword appears, no transformation is applied (actually the identity is applied). This might be useful, e.g., for wrapping an appearance around an externally-supplied object.

See Section 4.1.5 [Transformation matrices], page 40, for the matrix format.

There is no INST BINARY format.

4.2.6.1 INST Examples

Here are some examples of `INST` files

```
INST
     unit < xyz.vect
     transform {
        1 0 0 0
        0 1 0 0
        0 0 1 0
        1 3 0 1
     }


{ appearance { +edge  material { edgecolor 1 1 0 } }
    INST geom < mysurface.quad }


{INST transform {: T} geom {<dodec.off}}


{ INST
     transforms
         { LIST
      { < some-matrices.prj }
      { < others.prj }
      { TLIST <still more of them> }

         }
     geom
         { # stuff replicated by all the above matrices
      ...
```

```
                }
    }
```

4.2.7 LIST Files

The conventional suffix for a `LIST` file is '`.list`'.

A list of OOGL objects

Syntax:

```
    LIST
        oogl-object
        oogl-object
        ...
```

Note that there's no explicit separation between the oogl-objects, so they should be enclosed in curly braces (`{ }`) for sanity. Likewise there's no explicit marker for the end of the list; unless appearing alone in a disk file, the whole construct should also be wrapped in braces, as in:

```
{ LIST { QUAD ... } { < xyz.quad } }
```

A `LIST` with no elements, i.e. `{ LIST }`, is valid, and is the easiest way to create an empty object. For example, to remove a symbol's definition you might write

```
{ define somesymbol  { LIST } }
```

4.2.8 TLIST Files

The conventional suffix for a `TLIST` file is '`.grp`' ("group") or or '`.prj`' ("projective" matrices).

Collection of 4x4 matrices, used in the `transforms` section of and `INST` object.

Syntax:

```
TLIST # key word

<4x4 matrix (16 floats)>
... # Any number of 4x4 matrices
```

`TLIST`s are used only within the `transforms` clause of an `INST` object. They cause the `INST`s `geom` object to be instantiated once under each of the transforms in the `TLIST`. The effect is like that of a `LIST` of `INST`s each with a single transform, and all referring to the same object, but is more efficient.

Be aware that a `TLIST` is a kind of geometry object, distinct from a `transform` object. Some contexts expect one type of object, some the other. For example in

```
INST transform { : myT } geom { ... }
```

myT must be a transform object, which might have been created with the gcl

```
(read transform { define myT 1 0 0 1 ... })
```

while in

```
    INST transforms { : myTs } geom { ... }
or  INST transforms { LIST {: myTs} {< more.prj} } geom { ... }
```

myTs must be a geometry object, defined e.g. with

```
    (read geometry { define myTs { TLIST 1 0 0 1 ... } })
```

A `TLIST BINARY` format is accepted. Binary data begins with a 32-bit integer giving the number of transformations, followed by that number of 4x4 matrices in 32-bit floating-point format. The order of matrix elements is the same as in the ASCII format.

4.2.9 GROUP Files

This format is obsolete, but is still accepted. It combined the functions of `INST` and `TLIST`, taking a series of transformations and a single Geom (`unit`) object, and replicating the object under each transformation.

```
GROUP ... < matrices > ... unit { oogl-object }
```

is still accepted and effectively translated into

```
INST
transforms { TLIST ... <matrices> ... }
unit { oogl-object }
```

4.2.10 DISCGRP Files

This format is for discrete groups, such as appear in the theory of manifolds or in symmetry patterns. This format has its own man page. See discgrp(5).

4.3 Non-geometric objects

The syntax of these objects is given in the form used in See Section 4.1.7 [References], page 41, where "quoted" items should appear literally but without quotes, square bracketed ([ ]) items are optional, and `|` separates alternative choices.

4.3.1 Transform Objects

Where a single 4x4 matrix is expected – as in the `INST` `transform` field, the camera's `camtoworld` transform and the Geomview `xform*` commands – use a transform object.

Note that a transform is distinct from a `TLIST`, which is a type of geometry. `TLIST`s can contain one or more 4x4 transformations; "transform" objects must have exactly one.

Why have both? In many places – e.g. camera positioning – it's only meaningful to have a single transform. Using a separate object type enforces this.

Syntax for a transform object is

```
<transform> ::=
  [ "{" ]             (curly brace, generally needed to make
                       the end of the object unambiguous.)

   [ "transform" ]    (optional keyword; unnecessary if the type
                       is determined by the context, which it
                       usually is.)
   [ "define" <name> ]
                      (defines a transform named <name>, setting
                       its value from the stuff which follows)

      <sixteen floating-point numbers>
                      (interpreted as a 4x4 homogeneous transform
       given row by row, intended to apply to a
                       row vector multiplied on its LEFT, so that e.g.
                       Euclidean translations appear in the bottom row)
   |
      "<" <filename>  (meaning: read transform from that file)
   |
      ":" <name>      (meaning: use variable <name>,
                        defined elsewhere; if undefined the initial
                        value is the identity transform)

 [ "}" ]              (matching curly brace)
```

The whole should be enclosed in { braces }. Braces are not essential if exactly one of the above items is present, so e.g. a 4x4 array of floats standing alone may but needn't have braces.

Some examples, in contexts where they might be used:

```
# Example 1: A gcl command to define a transform
# called "fred"

(read transform { transform  define fred
         1 0 0 0
         0 1 0 0
         0 0 1 0
        -3 0 1 1
    }
)
```

```
# Example 2:  A camera object using transform
# "fred" for camera positioning
# Given the definition above, this puts the camera at
# (-3, 0, 1), looking toward -Z.

{ camera
        halfyfield 1
        aspect 1.33
        camtoworld { : fred }
}
```

4.3.2 cameras

A camera object specifies the following properties of a camera:

position and orientation

specified by either a camera-to-world or world-to-camera transformation; this transformation does not include the projection, so it's typically just a combination of translation and rotation. Specified as a transform object, typically a 4x4 matrix.

"focus" distance

Intended to suggest a typical distance from the camera to the object of interest; used for default camera positioning (the camera is placed at (X,Y,Z) = (0,0,focus) when reset) and for adjusting field-of-view when switching between perspective and orthographic views.

window aspect ratio

True aspect ratio in the sense <Xsize>/<Ysize>. This normally should agree with the aspect ratio of the camera's window. Geomview normally adjusts the aspect ratio of its cameras to match their associated windows.

near and far clipping plane distances

Note that both must be strictly greater than zero. Very large <far>/<near> distance ratios cause Z-buffering to behave badly; part of an object may be visible even if somewhat more distant than another.

field of view

Specified in either of two forms.

'`fov`' is the field of view – in degrees if perspective, or linear distance if orthographic – in the *shorter* direction.

`halfyfield`

is half the projected Y-axis field, in world coordinates (not angle!), at unit distance from the camera. For a perspective camera, halfyfield is related to angular field:

halfyfield = tan(Y_axis_angular_field / 2)

while for an orthographic one it's simply:

halfyfield = Y_axis_linear_field / 2

This odd-seeming definition is (a) easy to calculate with and (b) well-defined in both orthographic and perspective views.

The syntax for a camera is:

```
<camera> ::=

   [ "camera" ] (optional keyword)
    [ "{" ] (opening brace, generally required)
[ "define" <name> ]

"<" <filename>
      |
":" <name>
      |
(or any number of the following,
 in any order...)

"perspective"  {"0" | "1"} (default 1)
(otherwise orthographic)

"stereo"       {"0" | "1"} (default 0)
(otherwise mono)

"worldtocam" <transform> (see transform syntax above)

"camtoworld" <transform>
(no point in specifying both
 camtoworld and worldtocam; one is
 constrained to be the inverse of  the other)

"halfyfield" <half-linear-Y-field-at-unit-distance>
(default tan 40/2 degrees)

"fov" (angular field-of-view if perspective,
 linear field-of-view otherwise.
 Measured in whichever direction is smaller,
 given the aspect ratio.  When aspect ratio
```

```
      changes -- e.g. when a window is reshaped --
      "fov" is preserved.)

     "frameaspect" <aspect-ratio> (X/Y) (default 1.333)

     "near"  <near-clipping-distance> (default 0.1)

     "far" <far-clipping-distance> (default 10.0)

     "focus" <focus-distance> (default 3.0)


          [ "}" ] (matching closebrace)
```

4.3.3 window

A window object specifies size, position, and other window-system related information about a window in a device-independent way.

The syntax for a window object is:

```
window ::=

[ "window" ] (optional keyword)
  [ "{" ] (curly brace, often required)

     (any of the following, in any order)

"size"  <xsize> <ysize>
(size of the window)

"position"  <xmin> <xmax> <ymin> <ymax>
(position & size)


"noborder"
(specifies the window should
 have no window border)

"pixelaspect"  <aspect>
    (specifies the true visual aspect ratio
     of a pixel in this window in the sense
     xsize/ysize, normally 1.0.
     For stereo hardware which stretches the
     display vertically by a factor of 2,
     ``pixelaspect 0.5'' might do.
```

```
   The value is used when computing the
   projection of a camera associated with
   this window.)

[ "}" ] (matching closebrace)
```

Window objects are used in the Geomview `window` and `ui-panel` commands to set default properties for future windows or to change those of an existing window.

5 Customization: '`.geomview`' files

When Geomview is started, it loads and executes commands in a system-wide startup file named '`.geomview`'. This file is in the '`data`' subdirectory of the Geomview distribution directory '`/u/gcg/ngrap/data`' on the Geometry Center's computer system) and contains gcl commands to configure Geomview in a way common to all users on the system.

Next, Geomview looks for the file '`~/.geomview`' ('`~`' stands for your home directory). You can use this to configure your own default Geomview behavior to suit your tastes.

After reading '`~/.geomview`', Geomview looks for a file named '`.geomview`' in the current directory. If such a file exists Geomview reads it, unless it is the same as '`~/.geomview`' (which would be the case if you are running Geomview from your home directory). You can use the current directory's '`.geomview`' to create a Geomview customization specific to a certain project.

You can use '`.geomview`' files to control all kinds of things about Geomview. They can contain any valid gcl statements. Especially useful is the `ui-panel` command which controls the initial placement of Geomview's panels. For an example see the system-wide '`.geomview`' file mentioned above. For details of gcl, See Chapter 7 [GCL], page 86.

It is a good idea to enclose all the commands you put in a '`.geomview`' file in a `progn` statement in order to cause Geomview to execute them all at once. Otherwise Geomview might execute them sequentially over the first few refresh cycles after starting up.

6 External Modules

An external module is a program that interacts with Geomview. A module communicates with Geomview through gcl and can control any apsect of Geomview that you can control through Geomview's user interface.

In many cases an external module is a specialized program that implements some mathematical algorithm that creates a geometric object that changes shape as the algorithm progresses. The module informs Geomview of the new object shape at each step, so the object appears to evolve with time in the Geomview window. In this way Geomview serves as a *display engine* for the module.

An external module may be interactive. It can respond to mouse and keyboard events that take place in a Geomview window, thus extending the capability of Geomview itself.

6.1 How External Modules Interface with Geomview

External modules appear in the *Modules* browser in Geomview's *Main* panel. To run a module, click the left mouse button on the module's entry in the browser. While the module is running, an additional line for that module will appear in red in the browser. This line begins with a number in brackets, which indicates the *instace* number of the module. (For some modules it makes sense to have more than one instance of the module running at the same time.) You can kill an external module by clicking on its red instance entry.

By default when Geomview starts, it displays all the modules that have been installed on your system.

For instructions on installing a module on your system so that it will appear in the *Modules* browser every time Geomview is run by anyone on your system, See Section 6.6 [Module Installation], page 83.

When Geomview invokes an external module, it creates pipes connected to the module's standard input and output. (Pipes are like files except they are used for communication between programs rather than for storing things on a disk.) Geomview interprets anything that the module writes to its standard output as a gcl command. Likewise, if the exernal module requests any data from Geomview, Geomview writes that data to the module's standard input. Thus all a module has to do in order to communicate with Geomview is write commands to standard output and (optionally) receive data on standard input. Note that this means that the module cannot use standard input and output for communicating with the user. If a module needs to communicate with the user it can do so either through a control panel of its own or else by responding to certain events that it finds out about from Geomview.

6.2 Example 1: Simple External Module

This section gives a very simple external module which displays an oscillating mesh. To try out this example, make a copy of the file '`example1.c`' (it is distributed with Geomview in the '`doc`' subdirectory) in your directory and compile it with the command

```
cc -o example1 example1.c -lm
```

Then put the line

```
(emodule-define "Example 1" "./example1")
```

in a file called '`.geomview`' in your current directory. Then invoke Geomview; it is important that you compile the example program, create the '`.geomview`' file, and invoke Geomview all in the same directory. You should see "Example 1" in the *Modules* browser of Geomview's *Main* panel; click on this entry in the browser to start the module. A surface should appear in your camera window and should begin oscillating. You can stop the module by clicking on the red "[1] Example 1" line in the *Modules* browser.

```
/*
 * example1.c: oscillating mesh
 *
 * This example module is distributed with the Geomview manual.
 * If you are not reading this in the manual, see the "External
 * Modules" chapter of the manual for more details.
 *
 * This module creates an oscillating mesh.
 */

#include <math.h>
#include <stdio.h>

/* F is the function that we plot
```

```
 */
float F(x,y,t)
     float x,y,t;
{
  float r = sqrt(x*x+y*y);
  return(sin(r + t)*sqrt(r));
}

main(argc, argv)
     char **argv;
{
  int xdim, ydim;
  float xmin, xmax, ymin, ymax, dx, dy, t, dt;

  xmin = ymin = -5;             /* Set x and y            */
  xmax = ymax = 5;              /*    plot ranges         */
  xdim = ydim = 24;             /* Set x and y resolution */
  dt = 0.1;                     /* Time increment is 0.1  */

  /* Geomview setup.  We begin by sending the command
   *            (geometry example { : foo})
   * to Geomview.  This tells Geomview to create a geom called
   * "example" which is an instance of the handle "foo".
   */
  printf("(geometry example { : foo })\n");
  fflush(stdout);

  /* Loop until killed.
   */
  for (t=0; ; t+=dt) {
    UpdateMesh(xmin, xmax, ymin, ymax, xdim, ydim, t);
  }
}

/* UpdateMesh sends one mesh iteration to Geomview.  This consists of
 * a command of the form
 *    (read geometry { define foo
 *       MESH
 *       ...
 *    })
 * where ... is the actual data of the mesh.  This command tells
 * Geomview to make the value of the handle "foo" be the specified
 * mesh.
 */
UpdateMesh(xmin, xmax, ymin, ymax, xdim, ydim, t)
     float xmin, xmax, ymin, ymax, t;
     int xdim, ydim;
{
  int i,j;
  float x,y, dx,dy;
```

```
    dx = (xmax-xmin)/(xdim-1);
    dy = (ymax-ymin)/(ydim-1);

    printf("(read geometry { define foo \n");
    printf("MESH\n");
    printf("%1d %1d\n", xdim, ydim);
    for (j=0, y = ymin; j<ydim; ++j, y += dy) {
      for (i=0, x = xmin; i<xdim; ++i, x += dx) {
        printf("%f %f %f\t", x, y, F(x,y,t));
      }
      printf("\n");
    }
    printf("})\n");
    fflush(stdout);
  }
```

The module begins by defining a function `F(x,y,t)` that specifies a time-varying surface. The purpose of the module is to animate this surface over time.

The main program begins by defining some variables that specify the parameters with which the function is to be plotted.

The next bit of code in the main program prints the following line to standard output

```
(geometry example { : foo })
```

This tells Geomview to create a geom called `example` which is an instance of the handle `foo`. *Handles* are a part of the OOGL file format which allow you to name a piece of geometry whose value can be specified elsewhere (and in this case updated many times); for more information on handles, See Chapter 4 [OOGL File Formats], page 38 In this case, `example` is the title by which the user will see the object in Geomview's object browser, and `foo` is the internal name of the handle that the object is a reference to.

We then do `fflush(stdout)` to ensure that Geomview receives this command immediately. In general, since pipes may be buffered, an external module should do this whenever it wants to be sure Geomview has actually received everything it has printed out.

The last thing in the main program is an infinite loop that cycles through calls to the procedure `UpdateMesh` with increasing values of `t`. `UpdateMesh` sends Geomview a command of the form

```
(read geometry { define foo
MESH
24 24
...
})
```

where ... is a long list of numbers. This command tells Geomview to make the value of the handle `foo` be the specified mesh. As soon as Geomview receives this command, the geom being displayed changes to reflect the new geometry.

The mesh is given in the format of an OOGL MESH. This begins with the keyword `MESH`. Next come two numbers that give the x and y dimensions of the mesh; in this case they are both 24. This line is followed by 24 lines, each containing 24 triples of numbers. Each of these triples is a point on the surface. Then finally there is a line with "`})`" on it that ends the "`{`" which began the `define` statement and the "`(`" that began the command. For more details on the format of MESH data, see Section 4.2.2 [MESH], page 46.

This module could be written without the use of handles by having it write out commands of the form

```
(geometry example {
MESH
24 24
...
})
```

This first time Geomview receives a command of this form it would create a geom called `example` with the given `MESH` data. Subsequent `(geometry example ...)` commands would cause Geomview to replace the geometry of the geom `example` with the new `MESH` data. If done in this way there would be no need to send the initial `(geometry example { : foo })` command as above. The handle technique is useful, however, because it can be used in more general situations where a handle represents only part of a complex geom, allowing an external module to replace only that part without having to retransmit the entire geom. For more information on handles, See Chapter 7 [GCL], page 86.

The module loops through calls to `UpdateMesh` which print out commands of the above form one after the other as fast as possible. The loop continues indefinitely; the module will terminate when the user kills it by clicking on its instance line in the *Modules* browser, or else when Geomview exits.

Sometimes when you terminate this module by clicking on its instance entry the *Modules* browser, Geomview will kill it while it is in the middle of sending a command to Geomview. Geomview will then receive only a piece of a command and will print out a cryptic but harmless error message about this. When a module has a user interface panel it can use a "Quit" button to provide a more graceful way for the user to terminate the module. See the next example.

You can run this module in a shell window without Geomview to see the commands it prints out. You will have to kill it with `ctrl-C` to get it to stop.

6.3 Example 2: Simple External Module with FORMS Control Panel

This section gives a new version of the above module — one that includes a user interface panel for controlling the velocity of the oscillation. We use the FORMS library by Mark Overmars for the control panel. The FORMS library is a public domain user interface toolkit for IRISes; for more information See Section 6.4 [Forms], page 74.

To try out this example, make a copy of the file '`example2.c`' (distributed with Geomview in the '`doc`' subdirectory) in your directory and compile it with the command

```
cc -I/u/gcg/ngrap/include -o example2 example2.c \
  -L/u/gcg/ngrap/lib/sgi -lforms -lfm_s -lgl_s -lm
```

If you are not using the Geometry Center's computer system you should replace the string '`/u/gcg/ngrap`' above with the pathname of the Geomview distribution directory on your system. (The forms library is distributed with Geomview and the `-I` and `-L` options above tell the compiler where to find it.)

Then put the line

```
(emodule-define "Example 2" "./example2")
```

in a file called '`.geomview`' in the current directory and invoke Geomview from that directory. Click on the "Example 2" entry in the *Modules* browser to invoke the module. A small control panel should appear. You can then control the velocity of the mesh oscillation by moving the slider.

```
/*
 * example2.c: oscillating mesh with FORMS control panel
 *
 * This example module is distributed with the Geomview manual.
 * If you are not reading this in the manual, see the "External
 * Modules" chapter of the manual for an explanation.
 *
 * This module creates an oscillating mesh and has a FORMS control
 * panel that lets you change the speed of the oscillation with a
 * slider.
 */

#include <math.h>
#include <stdio.h>
#include <sys/time.h>           /* for struct timeval below */

#include "forms.h"              /* for FORMS library */

FL_FORM *OurForm;
FL_OBJECT *VelocitySlider;
float dt;

/* F is the function that we plot
 */
float F(x,y,t)
     float x,y,t;
{
  float r = sqrt(x*x+y*y);
  return(sin(r + t)*sqrt(r));
}

/* SetVelocity is the slider callback procedure; FORMS calls this
 * when the user moves the slider bar.
 */
void SetVelocity(FL_OBJECT *obj, long val)
{
  dt = fl_get_slider_value(VelocitySlider);
}

/* Quit is the "Quit" button callback procedure; FORMS calls this
 * when the user clicks the "Quit" button.
 */
void Quit(FL_OBJECT *obj, long val)
{
  exit(0);
```

```
}

/* create_form_OurForm() creates the FORMS panel by calling a bunch of
 * procedures in the FORMS library.  This code was generated
 * automatically by the FORMS designer program; normally this code
 * would be in a separate file which you would not edit by hand.  For
 * simplicity of this example, however, we include this code here.
 */
create_form_OurForm()
{
  FL_OBJECT *obj;
  FL_FORM *form;
  OurForm = form = fl_bgn_form(FL_NO_BOX,380.0,120.0);
  obj = fl_add_box(FL_UP_BOX,0.0,0.0,380.0,120.0,"");
  VelocitySlider = obj = fl_add_valslider(FL_HOR_SLIDER,20.0,30.0,
                                          340.0,40.0,"Velocity");
    fl_set_object_lsize(obj,FL_LARGE_FONT);
    fl_set_object_align(obj,FL_ALIGN_TOP);
    fl_set_call_back(obj,SetVelocity,0);
  obj = fl_add_button(FL_NORMAL_BUTTON,290.0,75.0,70.0,35.0,"Quit");
    fl_set_object_lsize(obj,FL_LARGE_FONT);
    fl_set_call_back(obj,Quit,0);
  fl_end_form();
}

main(argc, argv)
     char **argv;
{
  int xdim, ydim;
  float xmin, xmax, ymin, ymax, dx, dy, t;
  int fdmask;
  static struct timeval timeout = {0, 200000};

  xmin = ymin = -5;             /* Set x and y            */
  xmax = ymax = 5;              /*    plot ranges         */
  xdim = ydim = 24;             /* Set x and y resolution */
  dt = 0.1;                     /* Time increment is 0.1  */

  /* Forms panel setup.
   */
  foreground();
  create_form_OurForm();
  fl_set_slider_bounds(VelocitySlider, 0.0, 1.0);
  fl_set_slider_value(VelocitySlider, dt);
  fl_show_form(OurForm, FL_PLACE_SIZE, TRUE, "Example 2");


  /* Geomview setup.
   */
  printf("(geometry example { : foo })\n");
```

```
  fflush(stdout);

  /* Loop until killed.
   */
  for (t=0; ; t+=dt) {
    fdmask = (1 << fileno(stdin)) | (1 << qgetfd());
    select(qgetfd()+1, &fdmask, NULL, NULL, &timeout);
    fl_check_forms();
    UpdateMesh(xmin, xmax, ymin, ymax, xdim, ydim, t);
  }
}

/* UpdateMesh sends one mesh iteration to Geomview
 */
UpdateMesh(xmin, xmax, ymin, ymax, xdim, ydim, t)
     float xmin, xmax, ymin, ymax, t;
     int xdim, ydim;
{
  int i,j;
  float x,y, dx,dy;

  dx = (xmax-xmin)/(xdim-1);
  dy = (ymax-ymin)/(ydim-1);

  printf("(read geometry { define foo \n");
  printf("MESH\n");
  printf("%1d %1d\n", xdim, ydim);
  for (j=0, y = ymin; j<ydim; ++j, y += dy) {
    for (i=0, x = xmin; i<xdim; ++i, x += dx) {
      printf("%f %f %f\t", x, y, F(x,y,t));
    }
    printf("\n");
  }
  printf("})\n");
  fflush(stdout);
}
```

The code begins by including some header files needed for the event loop and the FORMS library. It then declares global variables for holding a pointer to the slider FORMS object and the velocity `dt`. These are global because they are needed in the slider callback procedure `SetVelocity`, which forms calls every time the user moves the slider bar. `SetVelocity` sets `dt` to be the new value of the slider.

`Quit` is the callback procedure for the *Quit* button; it provides a graceful way for the user to terminate the program.

The procedure `create_panel` calls a bunch of FORMS library procedures to set up the control panel with slider and button. For more information on using FORMS to create interface panels see the FORMS documentation. In particular, FORMS comes with a graphical panel designer that lets you design your panels interactively and generates code like that in `create_panel`.

This example's main program is similar to the previous example, but includes extra code to deal with setting up and managing the FORMS panel.

To set up the panel we call the GL procedure `foreground` to cause the process to run in the foreground. By default GL programs run in the background, and for various reasons external modules that use FORMS (which is based on GL) need to run in the foreground. We then call `create_panel` to create the panel and `fl_set_slider_value` to set the initial value of the slider. The call to `fl_show_form` causes the panel to appear on the screen.

The first three lines of the main loop, starting with

```
fdmask = (1 << fileno(stdin)) | (1 << qgetfd());
```

check for and deal with events in the panel. The call to `select` imposes a delay on each pass through the main loop. This call returns either after a delay of 1/5 second or when the next GL event occurs, or when data appears on standard input, whichever comes first. The `timeout` variable specifies the amount of time to wait on this call; the first member (0 in this example) gives the number of seconds, and the second member (200000 in this example) gives the number of microseconds. Finally, `fl_check_forms()` checks for and processes any FORMS events that have happened; in this case this means calling `SetVelocity` if the user has moved the slider or calling `Quit` if the user has clicked on the *Quit* button.

The purpose of the delay in the loop is to keep the program from using excessive amounts of CPU time running around its main loop when there are no events to be processed. This is not so crucial in this example, and in fact may actually slow down the animation somewhat, but in general with external modules that have event loops it is important to do something like this because otherwise the module will needlessly take CPU cycles away from other running programs (such as Geomview!) even when it isn't doing anything.

The last line of the main loop in this example, the call to `UpdateMesh`, is the same as in the previous example.

6.4 The FORMS Library

Geomview itself is written using Mark Overmar's public domain FORMS library. FORMS is a handy and relatively simple user interface toolkit for IRISes. Many Geomview external modules, including the examples in this manual, use FORMS to create and manage control panels.

We distribute a version of the FORMS library with Geomview because it is necessary in order to compile Geomview and many of our modules. If you use FORMS to write Geomview modules (or anything else, for that matter) you may use this copy. The header file '`forms.h`' is in the '`include`' subdirectory, and the library file '`libforms.a`' is in the '`lib/sgi`' subdirectory (these are subdirectories of the Geomview distribution directory, '`/u/gcg/ngrap`' on the Geometry Center's system). In particular, you can link the example modules in this manual using this copy.

FORMS is available via ftp on the Internet from a variety of sites, including `cs.ruu.nl` or `glaurung.physics.mcgill.ca`. It comes with source code and extensive documentation.

If you wish you may use any other interface toolkit instead of FORMS in an external module. We chose FORMS because it is free and relatively simple.

6.5 Example 3: External Module with Bi-Directional Communication

The previous two example modules simply send commands to Geomview and do not receive anything from Geomview. This section describes a module that communicates in both directions. There are two types of communication that can go from Geomview to an external module. This example shows *asynchronous* communication — the module needs to be able to respond at any moment to expressions that Geomview may emit which inform the module of some change of state within Geomview.

(The other type of communication is *synchronous*, where a module sends a request to Geomview for some piece of information and waits for a response to come back before doing anything else. The main gcl command for requesting information of this type is `write`. This module does not do any synchronous communication.)

In ansynchronous communication, Geomview sends expressions that are essentially echoes of gcl commands. The external module sends Geomview a command expressing interest in a certain command, and then every time Geomview executes that command, the module receives a copy of it. This happens regardless of who sent the command to Geomview; it can be the result of the user doing something with a Geomview panel, or it may have come from another module or from a file that Geomview reads. This is how a module can find out about and act on things that happen in Geomview.

This example uses the OOGL lisp library to parse and act on the expressions that Geomview writes to the module's standard input. This library is actually part of Geomview itself — we wrote the library in the process of implementing gcl. It is also convenient to use it in external modules that must understand a of subset of gcl — specifically, those commands that the module has expressed interest in.

This example shows how a module can receive user pick events, i.e. when the user clicks the right mouse button with the cursor over a geom in a Geomview camera window. When this happens Geomview generates an internal call to a procedure called `pick`; the arguments to the procedure give information about the pick, such as what object was picked, the coordinates of the picked point, etc. If an external module has expressed interest in calls to `pick`, then whenever `pick` is called Geomview will echo the call to the module's standard input. The module can then do whatever it wants with the pick information.

This module is the same as the *Nose* module that comes with Geomview. Its purpose is to illustrate picking. Whenever you pick on a geom by clicking the right mouse button on it, the module draws a little box at the spot where you clicked. Usually the box is yellow. If you pick a vertex, the box is colored magenta. If you pick a point on an edge of an object, the module will also highlight the edge by drawing cyan boxes at its endpoints and drawing a yellow line along the edge.

Note that in order for this module to actually do anything you must have a geom loaded into Geomview and you must click the right mouse button with the cursor over a part of the geom.

```
/*
 * example3.c: external module with bi-directional communication
 *
 * This example module is distributed with the Geomview manual.
 * If you are not reading this in the manual, see the "External
 * Modules" chapter of the manual for an explanation.
 *
 * This module is the same as the "Nose" program that is distributed
 * with Geomview.  It illustrates how a module can find out about
 * and respond to user pick events in Geomview.  It draws a little box
 * at the point where a pick occurrs.  The box is yellow if it is not
 * at a vertex, and magenta if it is on a vertex.  If it is on an edge,
 * the program also marks the edge.
 *
```

```
 * To compile:
 *
 *   cc -I/u/gcg/ngrap/include -g -o example3 example3.c \
 *      -L/u/gcg/ngrap/lib/sgi -loogl -lm
 *
 * If you are not on the Geometry Center's system you should replace
 * "/u/gcg/ngrap" above with the pathname of the Geomview distribution
 * directory on your system.
 */

#include <stdio.h>
#include "lisp.h"                 /* We use the OOGL lisp library */
#include "pickfunc.h"             /* for PICKFUNC below */
#include "3d.h"                   /* for 3d geometry library */

/* boxstring gives the OOGL data to define the little box that
 * we draw at the pick point.  NOTE:  It is very important to
 * have a newline at the end of the OFF object in this string.
 */
char boxstring[] = "\
INST\n\
transform\n\
.04 0 0 0\n\
0 .04 0 0\n\
0 0 .04 0\n\
0 0 0 1\n\
geom\n\
OFF\n\
8 6 12\n\
\n\
-.5 -.5 -.5     # 0   \n\
.5 -.5 -.5      # 1   \n\
.5  .5 -.5      # 2   \n\
-.5  .5 -.5     # 3   \n\
-.5 -.5  .5     # 4   \n\
.5 -.5  .5      # 5   \n\
.5  .5  .5      # 6   \n\
-.5  .5  .5     # 7   \n\
\n\
4 0 1 2 3\n\
4 4 5 6 7\n\
4 2 3 7 6\n\
4 0 1 5 4\n\
4 0 4 7 3\n\
4 1 2 6 5\n";

progn()
{
  printf("(progn\n");
}
```

```
endprogn()
{
  printf(")\n");
  fflush(stdout);
}

Initialize()
{
  extern LObject *Lpick();  /* This is defined by PICKFUNC below but must */
       /* be used in the following LDefun() call */
  LInit();
  LDefun("pick", Lpick, NULL);

  progn(); {
    /* Define handle "littlebox" for use later
     */
    printf("(read geometry { define littlebox { %s }})\n", boxstring);

    /* Express interest in pick events; see Geomview manual for explanation.
     */
    printf("(interest (pick world * * * * nil nil nil nil nil))\n");

    /* Define "pick" object, initially the empty list (= null object).
     * We replace this later upon receiving a pick event.
     */
    printf("(geometry \"pick\" { LIST } )\n");

    /* Make the "pick" object be non-pickable.
     */
    printf("(pickable \"pick\" no)\n");

    /* Turn off normalization, so that our pick object will appear in the
     * right place.
     */
    printf("(normalization \"pick\" none)\n");

    /* Don't draw the pick object's bounding box.
     */
    printf("(bbox-draw \"pick\" off)\n");

  } endprogn();
}

/* The following is a macro call that defines a procedure called
 * Lpick().  The reason for doing this in a macro is that that macro
 * encapsulates a lot of necessary stuff that would be the same for
 * this procedure in any program.  If you write a Geomview module that
 * wants to know about user pick events you can just copy this macro
 * call and change the body to suit your needs; the body is the last
```

```
 * argument to the macro and is delimited by curly braces.
 *
 * The first argument to the macro is the name of the procedure to
 * be defined, "Lpick".
 *
 * The next two arguments are numbers which specify the sizes that
 * certain arrays inside the body of the procedure should have.
 * These arrays are used for storing the face and path information
 * of the picked object.  In this module we don't care about this
 * information so we declare them to have length 1, the minimum
 * allowed.
 *
 * The last argument is a block of code to be executed when the module
 * receives a pick event.  In this body you can refer to certain local
 * variables that hold information about the pick.  For details see
 * Example 3 in the Extenal Modules chapter of the Geomview manual.
 */
PICKFUNC(Lpick, 1, 1,
{
  handle_pick(pn>0, &point, vn>0, &vertex, en>0, edge);
})

handle_pick(picked, p, vert, v, edge, e)
     int picked;                /* was something actually picked?     */
     int vert;                  /* was the pick near a vertex?        */
     int edge;                  /* was the pick near an edge?         */
     HPoint3 *p;                /* coords of pick point               */
     HPoint3 *v;                /* coords of picked vertex            */
     HPoint3 e[2];              /* coords of endpoints of picked edge */
{
  Normalize(&e[0]);             /* Normalize makes 4th coord 1.0 */
  Normalize(&e[1]);
  Normalize(p);
  progn(); {
    if (!picked) {
      printf("(geometry \"pick\" { LIST } )\n");
    } else {
      /*
       * Put the box in place, and color it magenta if it's on a vertex,
       * yellow if not.
       */
      printf("(xform-set pick { 1 0 0 0  0 1 0 0  0 0 1 0  %g %g %g 1 })\n",
             p->x, p->y, p->z);
      printf("(geometry \"pick\"\n");
      if (vert) printf("{ appearance { material { diffuse 1 0 1 } }\n");
      else printf("{ appearance { material { diffuse 1 1 0 } }\n");
      printf("  { LIST { :littlebox }\n");

      /*
       * If it's on an edge and not a vertex, mark the edge
```

```
       * with cyan boxes at the endpoins and a black line
       * along the edge.
       */
      if (edge && !vert) {
        e[0].x -= p->x; e[0].y -= p->y; e[0].z -= p->z;
        e[1].x -= p->x; e[1].y -= p->y; e[1].z -= p->z;
        printf("{ appearance { material { diffuse 0 1 1 } }\n\
  LIST\n\
   { INST transform 1 0 0 0 0 1 0 0 0 0 1 0 %f %f %f 1 geom :littlebox }\n\
   { INST transform 1 0 0 0 0 1 0 0 0 0 1 0 %f %f %f 1 geom :littlebox }\n\
   { VECT\n\
          1 2 1\n\
          2\n\
          1\n\
          %f %f %f\n\
          %f %f %f\n\
          1 1 0 1\n\
   }\n\
  }\n",
               e[0].x, e[0].y, e[0].z,
               e[1].x, e[1].y, e[1].z,
               e[0].x, e[0].y, e[0].z,
               e[1].x, e[1].y, e[1].z);
      }
      printf("    }\n  }\n)\n");
    }

  } endprogn();

}

Normalize(HPoint3 *p)
{
  if (p->w != 0) {
    p->x /= p->w;
    p->y /= p->w;
    p->z /= p->w;
    p->w = 1;
  }
}

main()
{
  Lake *lake;
  LObject *lit, *val;
  extern char *getenv();

  Initialize();

  lake = LakeDefine(stdin, stdout, NULL);
```

```
    while (!feof(stdin)) {

      /* Parse next lisp expression from stdin.
       */
      lit = LSexpr(lake);

      /* Evaluate that expression; this is where Lpick() gets called.
       */
      val = LEval(lit);

      /* Free the two expressions from above.
       */
      LFree(lit);
      LFree(val);
    }
  }
```

The code begins by defining procedures `progn()` and `endprogn()` which begin and end a Geomview `progn` group. The purpose of the Geomview `progn` command is to group commands together and cause Geomview to execute them all at once, without refreshing any graphics windows until the end. It is a good idea to group blocks of commands that a module sends to Geomview like this so that the user sees their cumulative effect all at once.

Procedure `Initialize()` does various things needed at program startup time. It initializes the lisp library by calling `LInit()`. Any program that uses the lisp library should call this once before calling any other lisp library functions. It then calls `LDefun` to tell the library about our `pick` procedure, which is defined further down with a call to the `DEFPICKFUNC` macro. Then it sends a bunch of setup commands to Geomview, grouped in a `progn` block. This includes defining a handle called `littlebox` that stores the geometry of the little box. Next it sends the command

```
(interest (pick world * * * * nil nil nil nil nil))
```

which tells Geomview to notify us when a pick event happens.

The syntax of this `interest` statement merits some explanation. In general `interest` takes one argument which is a (parenthesized) expression representing a Geomview function call. It specifies a type of call that the module is interested in knowing about. The arguments can be any particular argument values, or the special symbols `*` or `nil`. For example, the first argument in the `pick` expression above is `world`. This means that the module is interested in calls to `pick` where the first argument, which specifies the coordinate system, is `world`. A `*` is like a wild-card; it means that the module is interested in calls where the corresponding argument has any value. The word `nil` is like `*`, except that the argument's value is not reported to the module. This is useful for cutting down on the amount of data that must be transmitted in cases where there are arguments that the module doesn't care about.

The second, third, fourth, and fifth arguments to the `pick` command give the name, pick point coordinates, vertex coordinates, and edge coordinates of a pick event. We specify these by `*`'s above. The remaining five arguments to the `pick` command give other information about the pick event that we do not care about in this module, so we specify these with `nil`'s. For the details of the arguments to `pick`, See Chapter 7 [GCL], page 86.

The `geometry` statement defines a geom called `pick` that is initially an empty list, specified as `{ LIST }` ; this is the best way of specifying a null geom. The module will replace this with something useful by sending Geomview another `geometry` command when the user picks something. Next we arrange for the `pick` object to be non-pickable, and turn normalization off for it so that Geomview will display it in the size and location where we put it, rather than resizing and relocating it to fit into the unit cube.

The next function in the file, `Lpick`, is defined with a strange looking call to a macro called `PICKFUNC`, defined in the header file '`pickfunc.h`'. This is the function for handling pick events. The reason we provide a macro for this is that that macro encapsulates a lot of necessary stuff that would be the same for the pick-handling function in any program. If you write a Geomview module that wants to know about user pick events you can just copy this macro call and change it to suit yours needs.

In general the syntax for `PICKFUNC` is

`PICKFUNC(`*name*`, `*maxfaceverts*`, `*maxpathlen*`, `*block*`)`

where *name* is the name of the procedure to be defined, in this case `Lpick`. The next two arguments, *maxfaceverts* and *maxpathlen*, give the sizes to be used for declaring two local variable arrays in the body of the procedure. These arrays are for storing information about the picked face and the picked primitive's path. In this module we don't care about this information (it corresponds to some of the things masked out by the `nil`'s in the `interest` call above) so we specify 1, the minimum allowable, for both of these. The last argument, *block*, is a block of code to be executed when a pick event occurs. The *block* should be delimited by curly braces. The code in your *block* should not include any `return` statements.

`PICKFUNC` declares certain local variables in the body of the procedure. When the module receives a `(pick ...)` statement from Geomview, the procedure assigns values to these variables based on the information in the `pick` call. (Variables corresponding to `nil`'s in the `(interest (pick ...))` are not given values.) These variables are:

`char *coordsys;`

A string specifying the coordinate system in which coordinates are given. In this example, this will always be `world` because of the `interest` call above.

`char *id;` A string specifying the name of the picked geom.

`HPoint3 point; int pn;`

`point` is an `HPoint3` structure giving the coordinates of the picked point. `HPoint3` is a homogeneous point coordinate representation eqivalent to an array of 4 floats. `pn` tells how many coordinates have been written into this array; it will always be either 0 or 4. A value of zero means no point was picked, i.e. the user clicked the right mouse button while the cursor was not pointing at a geom.

`HPoint3 vertex; int vn;`

`vertex` is an `HPoint3` structure giving the coordinates of the picked vertex, if the pick point was near a vertex. `vn` tells how many coordinates have been written into this array; it will always be either 0 or 4. A value of zero means the pick point was not near a vertex.

`HPoint3 edge[2]; int en;`

`edge` is an array of two `HPoint3` structures giving the coordinates of the endpoints of the picked edge, if the pick point was near an edge. `en` tells how many coordinates have been written into this array; it will always be either 0 or 8. A value of zero means the pick point was not near an edge.

In this example module, the remaining variables will never be given values because their values in the `interest` statement were specified as `nil`.

`HPoint3 face[`*maxfaceverts*`]; int fn;`

`face` is an array of *maxfaceverts* `HPoint3`'s; *maxfaceverts* is the value specified in the `PICKFUNC` call. `face` gives the coordinates of the vertices of the picked face. `fn` tells how many coordinates have been written into this array; it will always be a multiple of 4 and will be at most 4**maxfaceverts*. A value of zero means the pick point was not near a face.

`HPoint3 ppath[`*maxpathlen*`; int ppn;`

`ppath` is an array of *maxpathlen* `int`'s; *maxpathlen* is the value specified in the `PICKFUNC` call. `ppath` gives the path through the OOGL heirarchy to the picked primitive. `pn` tells how many integers have been written into this array; it will be at most *maxpathlen*. A path of 3,1,2, for example, means that the picked primitive is "subobject number 2 of subobject number 1 of object 3 in the world".

`int vi;` `vi` gives the index of the picked vertex in the picked primitive, if the pick point was near a vertex.

`int ei[2]; int ein`

The `ei` array gives the indices of the endpoints of the picked edge, if the pick point was near a vertex. `ein` tells how many integers were written into this array. It will always be either 0 or 2; a value of 0 means the pick point was not near an edge.

`int fi;` `fi` gives the index of the picked face in the picked primitive, if the pick point was near a face.

The `handle_pick` procedure actually does the work of dealing with the pick event. It begins by normalizing the homogeneous coordinates passed in as arguments so that we can assume the fourth coordinate is 1. It then sends gcl commands to define the `pick` object to be whatever is appropriate for the kind of pick recieved. See see Chapter 4 [OOGL File Formats], page 38, and see Chapter 7 [GCL], page 86, for an explanation of the format of the data in these commands.

The main program, at the bottom of the file, first calls `Initialize()`. Next, the call to `LakeDefine` defines the `Lake` that the lisp library will use. A `Lake` is a structure that the lisp library uses internally as a type of communiation vehicle. (It is like a unix stream but more general, hence the name.) This call to `LakeDefine` defines a `Lake` structure for doing I/O with `stdin` and `stdout`. The third argument to `LakeDefine` should be `NULL` for external modules (it is used by Geomview). Finally, the program enters its main loop which parses and evaluates expressions from standard input.

6.6 Module Installation

This section tells how to install an external module so you can invoke it within Geomview. There are two ways to install a module: you can install a *private* module so that the module is available to you whenever you run Geomview, or you can install a *system* module so that the module is available to all users on your system whenever they run Geomview.

6.6.1 Private Module Installation

The `emodule-define` command arranges for a module to appear in Geomview's *Modules* browser. `emodule-define` takes two string arguments; the first is the name that will appear in the *Modules* browser. The second is the shell command for running the module; it may include arguments. Geomview executes this command in a subshell when you click on the module's entry in the browser. For example

```
(emodule-define "Foo" "/u/home/modules/foo -x")
```

adds a line labeled "Foo" to the *Modules* browser which causes the command "/u/home/modules/foo -x" to be executed when selected.

You may put `emodule-define` commands in your '`~/.geomview`' file to arrange for certain modules to be available every time you run Geomview; See Chapter 5 [Customization], page 63. You can also execute `emodule-define` commands from the *Commands* panel if you want to add a module to an already running copy of Geomview.

There are several other gcl commands for controlling the entries in the *Modules* browser; for details, See Chapter 7 [GCL], page 86.

6.6.2 System Module Installation

To install a module so that it is available to all Geomview users do the following

1 Create a file called '`.geomview-`*module*' where '*module*' is the name of the module. This file should contain a single line which is an `emodule-define` command for that module:

```
(emodule-define "New Module" "newmodule")
```

The first argument, `"New Module"` above, is the string that will appear in the *Modules* browser. The second string, `"newmodule"` above, is the shell command for invoking the module. It may include arguments, and you may assume that the module is on the $path searched by the shell.

2 Put a copy of the '`.geomview-`*module*' and the module executable itself in Geomview's '`modules/sgi`' directory. This is a subdirectory of the Geomview distribution directory (on the Geometry Center's system the pathname is '`/u/gcg/ngrap/modules/sgi`'.

After these steps, the new module should appear, in alphabetical position, in the *Modules* browser of Geomview's *Main* panel next time Geomview is run. The reason this works is that when Geomview is invoked it processes all the '`.geomview-*`' files in its '`modules`' directory. It also remembers the pathname of this directory and prepends that path to the $path of the shell in which it invokes such a module.

7 gcl: the Geomview Command Language

Gcl has the syntax of lisp – i.e. an expression of the form (f a b ...) means pass the values of a, b, ... to the function f. Gcl is very limited and is by no means an implementation of lisp. It is simply a language for expressing commands to be executed in the order given, rather than a programming language. It does not support variable or function definition.

Gcl is the language that Geomview understands for files that it loads as well as for communication with other programs. If you want to execute a gcl command interactively, you can bring up the *Command* panel which lets you type in a command; Geomview executes the command when you hit the return key. Output from such commands is printed to standard output. Alternately, you can invoke Geomview as `geomview -c -` which causes it to read gcl commands from standard input.

Gcl functions return a value, and you can nest function calls in ways which use this returned value. For example

```
(f (g a b))
```

evaluates `(g a b)` and then evaluates `(f x)` where `x` is the result returned by `(g a b)`. Geomview maintains these return values internally but does not normally print them out. If you want to print out a return value pass it to the `echo` function. For example the `geomview-version` function returns a string representing the version of Geomview that is running, and

```
(echo (geomview-version))
```

prints out this string.

Many functions simply return `t` for success or `nil` for failure; this is the case if the documentation for the function does not indicate otherwise. These are the lisp symbols for true and false, respectively. (They correspond to the C variables `Lt` and `Lnil` which you are likely to see if you look at the source code for Geomview or some of the external modules.)

In the descriptions of the commands below several references are made to "OOGL" formats. OOGL is the data description language that Geomview uses for describing geometry, cameras, appearances, and other basic objects. For details of the OOGL formats, See Chapter 4 [OOGL File

Formats], page 38. (Or eqivalently, see the oogl(5) manual page, distributed with Geomview in the file man/cat5/oogl.5.

The gcl commands and argument types are listed below. Most of the documentation in this section of the manual is available within Geomview via the `?` and `??` commands. The command (`?` *command*) causes Geomview to print out a one-line summary of the syntax of *command*, and (`??` *command*) prints out an explanation of what *command* does. You can include the wild-card character `*` in *command* to print information for a group of commands matching a pattern. For example, `(?? *emodule*)` will print all information about all commands containing the string `emodule`. `(? *)` will print a short list of all commands.

7.1 Conventions Used In Describing Argument Types

The following symbols are used to describe argument types in the documentation for gcl functions.

appearance	is an OOGL appearance specification.
cam-id	is an *id* that refers to a camera.
camera	is an OOGL camera specification.
geom-id	is an *id* that refers to a geometry.
geometry	is an OOGL geometry specification.
id	is a string which names a geometry or camera. Besides those you create, valid ones are:

`World, world, worldgeom, g0`	the collection of all geom's
`target`	selected target object (cam or geom)
`center`	selected center-of-motion object
`targetcam`	last selected target camera
`targetgeom`	last selected target geom
`focus`	camera where cursor is (or most recently was)
`allgeoms`	all geom objects
`allcams`	all cameras

`default, defaultcam, prototype`
future cameras inherit default's settings

The following *ids* are used to name coordinate systems, e.g. in `pick` and `write` commands:

`World, world, worldgeom, g0`
the world, within which all other geoms live.

`universe` the universe, in which the World, lights and cameras live. Cameras' world2cam transforms might better be called universe2cam, etc.

`self` "this Geomview object". Transform from an object to `self` is the identity; writing its geometry gives the object itself with no enclosing transform; picked points appear in the object's coordinates.

`primitive`
(for `pick` only) Picked points appear in the coordinate system of the lowest-level OOGL primitive.

A name is also an acceptable id. Given names are made unique by appending numbers if necessary (i.e. "foo<2>"). Every geom is also named g[n] and every camera is also named c[n] ("g0" is always the worldgeom): this name is used as a prefix to keyboard commands and can also be used as a gcl id. Numbers are reused after an object is deleted. Both names are shown in the Object browser.

statement represents a function call. Function calls have the form `(func arg1 arg2 ... )`, where `func` is the name of the function and `arg1`, `arg2`, ... are the arguments.

transform is an OOGL 4x4 transformation matrix.

window is an OOGL winddow specification.

7.2 Gcl Reference Guide

`!` `!` is a synonym for `shell`

`?` `?` is a synonym for `help`

`??` `??` is a synonym for `morehelp`

`|` `|` is a synonym for `emodule-run`

`(<` *expr1 expr2*`)`
Returns t if *expr1* is less than *expr2*. *expr1* and *expr2* should be either both integers or floats, or both strings.

`(=` *expr1 expr2*`)`
Returns t if *expr1* is equal to *expr2*. *expr1* and *expr2* should be either both integers or floats, or both strings.

`(>` *expr1 expr2*`)`

Returns t if *expr1* is greater than *expr2. expr1* and *expr2* should be either both integers or floats, or both strings.

`(all geometry)`

returns a list of names of all geometry objects. Use e.g. "(echo (all geometry))" to print such a list.

`(all camera)`

returns a list of names of all cameras.

`(all emodule defined)`

returns a list of all defined external modules.

`(all emodule running)`

returns a list of all running external modules.

`(ap-override [on|off])`

Selects whether appearance controls should override objects' own settings. On by default. With no arguments, returns current setting.

`(backcolor` *cam-id r g b*`)`

Set the background color of *cam-id*; *r g b* are numbers between 0 and 1.

`(bbox-color` *geom-id r g b*`)`

Set the bounding-box color of *geom-id*; *r g b* are between 0 and 1.

`(bbox-draw` *geom-id* `[yes|no])`

Say whether *geom-id*'s bounding-box should be drawn; `yes` if omitted.

`(camera` *cam-id* `[`*camera*`])`

Specify data for *cam-id*; *camera* is a string giving an OOGL camera specification. If no camera *cam-id* exists, it is created; in this case, the second argument is optional, and if omitted, a default camera is used. See also: new-camera.

`(camera-draw` *cam-id* `[yes|no])`

Say whether or not cameras should be drawn in *cam-id*; `yes` if omitted.

`(camera-prop` *geometry* `[projective])`

Specify the object to be shown when drawing other cameras. By default, this object is drawn with its origin at the camera, and with the camera looking toward the object's -Z axis. With the "projective" keyword, the camera's viewing projection is also applied to the object; this places the object's Z=-1 and Z=+1 at near and far clipping planes, with the viewing area -1<=X,Y<=+1. Example: (camera-prop < cube projective)

`(camera-reset` *cam-id*`)`

Reset *cam-id* to its default value.

`(car LIST)`

returns the first element of LIST.

`(cdr LIST)`

returns the list obtained by removing the first element of LIST.

`(clock)` Returns the current time, in seconds, as shown by this stream's clock. See also set-clock and sleep-until.

`(command INFILE [OUTFILE])`

Read commands from INFILE; send corresponding responses (e.g. anything written to filename '`-`') to OUTFILE, stdout by default.

`(copy [`*id*`] [name])`

Copies an object or camera. If *id* is not specified, it is assumed to be targetgeom. If name is not specified, it is assumed to be the same as the name of *id*.

`(cull-backface [on|off])`

Select whether back-facing polygons should be displayed. Initially on: all polygons are displayed. When off, polygons whose vertices are arranged clockwise on the screen are hidden. Useful for simulating two-sided surface coloring.

`(cursor` *cam-id* `{on|off} [pbmfile xorigin yorigin])`

Turns the given window's graphics cursor on or off. Optionally sets the 16x16 pixel cursor glyph from the given file, which must be in binary (P4) PBM format. Can only be applied to actual windows, not e.g. `allcams` or `default`. Sorry.

`(cursor-still [INT])`

Sets the number of microseconds for which the cursor must not move to register as holding still. If INT is not specified, the value will be reset to the default.

`(cursor-twitch [INT])`

Sets the distance which the cursor must not move (in x or y) to register as holding still. If INT is not specified, the value will be reset to the default.

`(delete` *id*`)`

Delete object or camera *id*.

`(dimension [`N`])`

Sets or reads the space dimension for N-dimensional viewing. (Since calculations are done using homogeneous coordinates, this means matrices are (N+1)x(N+1).) With no arguments, returns the current dimension, or 0 if N-dimensional viewing has not been enabled.

`(dice` *geom-id* N`)`

Dice any Bezier patches within *geom-id* into NxN meshes; default 10.

`(draw` *cam-id*`)`

Draw the view in *cam-id*, if it needs redrawing. See also `redraw`.

`(echo ...)`

Write the given data to the special file '`-`'. Strings are written literally; lisp expressions are evaluated and their values written. If received from an external program, `echo` sends to the program's input. Otherwise writes to Geomview's own standard output (typically the terminal).

`(emodule-clear)`

Clears the Geomview application (external module) browser.

`(emodule-define` *name shell-command* `...)`

Define an external module called *name*, which then appears in the external-module browser. The *shell-command* string is a UNIX shell command which invokes the module. See emodule-run for discussion of external modules.

`(emodule-defined` *modulename*`)`

If an external module named *modulename* is known, returns the name of the program invoked when it's run as a quoted string; otherwise returns `nil`. `(echo (emodule-defined` *modulename*`))` prints the string.

`(emodule-isrunning` *name*`)`

Returns Lt if the emodule *name* is running, or Lnil if it is not running. *name* is searched for in the names as they appear in the browser and in the shell commands used to execute the external modules (not including arguements).

`(emodule-path)`

Returns the current search path for external modules. Note: to actually see the value returned by this function you should wrap it in a call to echo: (echo (emodule-path)). See also set-emodule-path.

`(emodule-run` *shell-command args* `...)`

Runs the given *shell-command* (a string containing a UNIX shell command) as an external module. The module's standard output is taken as gcl commands; responses (written to filename '`-`' are sent to the module's standard input. The shell command is interpreted by /bin/sh, so e.g. I/O redirection may be used; a program which prompts the user for input from the terminal could be run with: `(emodule-run yourprogram <&2)`. If not already set, the environment variable $MACHTYPE is set to the name of the machine type. Input and output connections to Geomview are dropped when the shell command terminates. Clicking on a running program's module-browser entry sends the signal SIGHUP to the program. For this to work, programs should avoid running in the background; those using FORMS or GL should call foreground() before the first FORMS or winopen() call. See also emodule-define, emodule-start.

`(emodule-sort)`

Sorts the modules in the *Modules* browser alphabetically.

`(emodule-start` *name*`)`

Starts the external module *name*, defined by emodule-define. Equivalent to clicking on the corresponding module-browser entry.

`(emodule-transmit` *name* `LIST)`

Places LIST into external module *name*'s standard input. *name* is searched for in the names of the modules as they appear in the External Modules browser and then in the shell commands used to execute the external modules. Does nothing if modname is not running.

`(escale` *geom-id factor*`)`

Same as scale but multiplies by exp(scale). Obsolete.

`(event-mode` *modestring*`)`

Set the mouse event (motion) mode; *modestring* should be one of the strings that appears in the motion mode browser (including the keyboard shortcut, e.g. "[r] Rotate").

`(evert` *geom-id* `[yes|no])`

Set the normal eversion state of *geom-id*. If the second argument is omitted, toggle the eversion state.

`(exit)` Terminates Geomview.

`(ezoom` *geom-id factor*`)`

Same as zoom but multiplies by exp(zoom). Obsolete.

`(freeze` *cam-id*`)`

Freeze *cam-id*; drawing in this camera's window is turned off until it is explicitly redrawn with `(redraw` *cam-id*`)`, after which time drawing resumes as normal.

`(geometry` *geom-id* `[`*geometry*`])`

Specify the geometry for *geom-id*. *geometry* is a string giving an OOGL geometry specification. If no object called *geom-id* exists, it is created; in this case the *geometry* argument is optional, and if omitted, the new object *geom-id* is given an empty geometry.

`(geomview-version)`

Returns a string representing the version of Geomview that is running.

`(hdefine geometry|camera|transform|window` *name value*`)`

Sets the value of a handle of a given type. (hdefine *type name value*) is generally equivalent to `(read` *type* `{ define` *name value* `})` except that the assignment is done when hdefine is executed, (possibly not at all if inside a conditional statement), while `(read` ... `define` ...`)` performs assignment as soon as the text is read.

`(help [command])`

Command may include "*"s as wildcards; see also `morehelp`. One-line command help; lists names only if multiple commands match. `?` is a synonym for `help`.

`(hmodel` *cam-id* `{virtual|projective|conformal})`

Set the model used to display geometry in this camera; see also `space`.

`(hsphere-draw` *cam-id* `[yes|no])`

Say whether to draw the sphere at infinity in hyperbolic space. If the second argument is omitted, `yes` is assumed.

`(if` *test expr1* `[`*expr2*`])`

Evaluates *test*; if *test* returns a non-nil value, returns the value of *expr1*. If *test* returns nil, returns the value of *expr2* if *expr2* is present, otherwise returns nil.

`(inhibit-warning` *string*`)`

Inhibit warning inhbits Geomview from displaying a particular warning message determined by *string*. At present there are no warning messages that this applies to, so this command is rather useless.

`(interest (`*command* `[`*args*`]))`

Allows you to express interest in a command. When Geomview executes that command in the future it will echo it to the communication pool from which the interest command came. *command* can be any command. Args specify restrictions on the values of the arguments; if *args* are present in the interest command, Geomview will only echo calls to the command in which the arguments match those given in the interest command. Two special argument values may appear in the argument list. `*` matches any value. `nil` matches any value but supresses the reporting of that value; its value is reported as `nil`.

The purpose of the interest command is to allow external modules to find out about things happening inside Geomview. For example, a module interested in knowing when a geom called "foo" is deleted could say `(interest (delete foo))` and would receive the string `(delete foo)` when foo is deleted.

Picking is a special case of this. For most modules interested in pick events the command `(interest (pick world))` is sufficient. This causes Geomview to send a string of the form `(pick world ...)` every time a pick event (right mouse double click). See the `pick` command for details.

`(lines-closer` *cam-id* `DIST)`

Draw lines (including edges) closer to the camera than polygons by DIST / 10^5 of the Z-buffer range. DIST = 3.0 by default. If DIST is too small, a line lying on a surface may be dotted or invisible, depending on the viewpoint. If DIST is too large, lines may appear in front of surfaces that they actually lie behind. Good values for DIST vary with the scene, viewpoint, and distance between near and far clipping planes. This feature is a kludge, but can be helpful.

`(load filename [command|geometry|camera])`

Loads the given file into Geomview. The optional second argument specifies the type of data it contains, which may be `command` (gcl commands), `geometry` (OOGL geometric data), or `camera` (OOGL camera definition). If omitted, attempts to guess about the file's contents. Loading geometric data creates a new visible object; loading a camera opens a new window; loading a gcl file executes the commands in the file.

`(load-path)`

Returns the current search path for command, geometry, etc. files. Note: to actually see the value returned by this function you should wrap it in a call to echo: (echo (load-path)). See also set-load-path.

`(look` [*geom-id*] [*cam-id*]`)`

Rotates the named camera to point toward the center of the bounding box of the named object (or the origin in hyperbolic or sphereical space). In Euclidean space, moves the camera forward or backward until the object appears as large as possible while still being entirely visible. Equivalent to

```
progn (
  (look-toward [geom-id] [cam-id] {center | origin})
  [(look-encompass [geom-id] [cam-id])]
)
```

If *geom-id* is not specified, it is assumed to be World. If *cam-id* is not specified, it is assumed to be targetcam.

`(look-encompass` [*geom-id*] [*cam-id*]`)`

Moves *cam-id* backwards or forwards until its field of view surrounds *geom-id*. This routine works only in Euclidean space. If *geom-id* is not specified, it is assumed to be the world. If *cam-id* is not specified, it is assumed to be the targetcam. See also (look-encompass-size).

`(look-encompass-size` [*view-fraction clip-ratio near-margin far-margin*]`)`

Sets/returns parameters used by (look-encompass). *view-fraction* is the portion of the camera window filled by the object, *clip-ratio* is the max allowed ratio of near-to-far clipping planes. The near clipping plane is 1/*near-margin* times closer than the near edge of the object, and the far clipping plane is *far-margin* times further away. Returns the list of current values. Defaults: .75 100 0.1 4.0

`(look-recenter` [*geom-id*] [*cam-id*]`)`

Translates and rotates the camera so that it is looking in the -z direction (in *geom-id*'s coordinate system) at the center of *geom-id*'s bounding box (or the origin of the coordinate system in non-Eudlidean space). In Euclidean space, the camera is also moved as close as possible to the object while allowing the entire object to be visible. Also makes sure that the y-axes of *geom-id* and *cam-id* are parallel.

`(look-toward [`*geom-id*`] [`*cam-id*`] [origin | center])`

Rotates the named camera to point toward the origin of the object's coordinate system, or the center of the object's bounding box (in non-Euclidean space, the origin will be used automatically). Default *geom-id* is the world, default camera is targetcam, default location to point towards is the center of the bounding box.

`(merge {window|camera}` *cam-id* `{` *window* `or` *camera* ... `} )`

Modify the given window or camera, changing just those properties specified in the last argument. E.g. `(merge camera "Camera" { far 20 })` sets Camera's far clipping plane to 20 while leaving other attributes untouched.

`(merge-ap` *geom-id appearance*`)`

Merge in some appearance characteristics to *geom-id*. Appearance parameters include surface and line color, shading style, line width, and lighting.

`merge-base-ap`

is a synonym for merge-baseap.

`(merge-baseap` *appearance*`)`

Merge in some appearance characteristics to the base default appearance (applied to every geom before its own apperance). Lighting is typically included in the base appearance.

`(morehelp` *command*`)`

command may include `"*"` wildcards. Prints more info than `(help` *command*`)`. `??` is a synonym for `morehelp`

`(name-object` *id name*`)`

Assign a new *name* (a string) to *id*. A number is appended if that name is in use (for example, "foo" -> "foo<2>"). The new name, possibly with number appended, may be used as object's id thereafter.

`(new-alien name [`*geometry*`])`

Create a new alien (geom not in the world) with the given name (a string). *geometry* is a string giving an OOGL geometry specification. If *geometry* is omitted, the new alien is given an empty geometry. If an object with that name already exists, the new alien is given a unique name. The light beams that are used to move around the lights are an example of aliens. They're drawn but are not controllable the way ordinary objects are: they don't appear in the object browser and the user can't move them with the normal motion modes.

`(new-camera name [`*camera*`])`

Create a new camera with the given name (a string). If a camera with that name already exists, the new object is given a unique name. If *camera* is omitted a default camera is used.

`(new-center [id])`

Stop id, then set id's transform to the identity. Default id is target. Also, if the id is a camera, calls (look-recenter World id). The main function of the call to (look-recenter) is to place the camera so that it is pointing parallel to the z axis toward the center of the world.

`(new-geometry name [`*geometry*`])`

Create a new geom with the given name (a string). *geometry* is a string giving an OOGL geometry specification. If *geometry* is omitted, the new object is given an empty geometry. If an object with that name already exists, the new object is given a unique name.

`(new-reset)`

Equivalent to (progn (new-center ALLGEOMS)(new-center ALLCAMS))

`(ND-axes` *cam-id* `[`*clustername* `[Xindex Yindex Zindex]])`

Sets or reads the set of axes in the space of the N-dimensional virtual camera *clustername* which determine the 3-D subspace seen by camera *cam-id*. Axes are specified by their indices, from 0 to N-1 for an N-dimensional space.

`(ND-color` *cam-id* `[ (((x0 x1 x2 ... xn) v r g b a v r g b a ... )`

((x0 ... xn) v r g b a v r g b a ...) ...)])

`(ND-xform` *geom-id* `[`*ntransform idim odim* `... ])`

Sets or returns the N-D transform of the given *geom*.

`(NeXT)` Returns t if running on a NeXT, nil if not

`(normalization` *geom-id* `{each|none|all|keep})`

Set the normalization status of *geom-id*.

`none`	suppresses all normalization.
`each`	normalizes the object's bounding box to fit into the unit sphere, with the center of its bounding box translated to the origin. The box is scaled such that its long diagonal, sqrt((xmax-xmin)^2 + (ymax-ymin)^2 + (zmax-zmin)^2), is 2.
`all`	resembles `each`, except when an object is changing (e.g. when its geometry is being changed by an external program). Then, `each` tightly fits the bounding box around the object whenever it changes and normalizes accordingly, while `all` normalizes the union of all variants of the object and normalizes accordingly.
`keep`	leaves the current normalization transform unchanged when the object changes.

It may be useful to apply `each` or `all` normalization apply to the first version of a changing object to bring it in view, then switch to `keep`.

`(pick COORDSYS GEOMID G V E F P VI EI FI)`

The pick command is executed internally in response to pick events (right mouse double click).

`COORDSYS`	coordinate system in which coordinates of the following arguments are specified. This can be: `world` world coord sys `self` coord sys of the picked geom (GEOMID) `primitive` coord sys of the actual primitive within the picked geom where the pick occurred.
`GEOMID`	id of picked geom
`G`	picked point (actual intersection of pick ray with object)
`V`	picked vertex, if any
`E`	picked edge, if any
`F`	picked face
`P`	path to picked primitive [0 or more]
`VI`	index of picked vertex in primitive
`EI`	list of indices of endpoints of picked edge, if any
`FI`	index of picked face

External modules can find out about pick events by registering interest in calls to `pick` via the `interest` command.

`(pickable` *geom-id* `{yes|no})`

Say whether or not *geom-id* is included in the pool of objects that could be returned from the pick command.

`(position objectID otherID)`

Set the transform of objectID to that of otherID.

`(position-at objectID otherID [center | origin])`

Translate objectID to the center of the bounding box or the origin of the coordinate system of otherID (parallel translation). Default is center.

`(position-toward objectID otherID [center | origin])`

Rotate objectID so that the center of the bounding box or the origin of the coordinate system of the otherID lies on the positive z-axis of the first object. Default is the center of the bounding box.

`(progn STATEMENT [ ... ])`

evaluates each STATEMENT in order and returns the value of the last one. Use progn to group a collection of commands together, forcing them to be treated as a single command.

`quit` is a synonym for `exit`.

`(quote EXPR)`

returns the symbolic lisp expression EXPR without evaluating it.

`(rawevent dev val x y t)`

Enter the specified raw event into the event queue. The arguments directly specify the members of the event structure used internally by Geomview. This is the lowest level event handler and is not intended for general use.

`(rawpick` *cam-id* `X Y)`

Process a pick event in camera *cam-id* at location (X,Y) given in integer pixel coordinates. This is a low-level procedure not intended for external use.

`(read {geometry|camera|transform|command}` {*geometry* `or` *camera* `or` ...}`)`

Read and interpret the text in ... as containing the given type of data. Useful for defining objects using OOGL reference syntax, e.g.

```
(geometry  thing { INST  transform : T    geom : fred })
(read  geometry  { define fred QUAD 1 0 0  0 1 0  0 0 1  1 0 0 })
(read  transform { define T <myfile})
```

`(real-id` *id*`)`

Returns a string canonically identifying the given *id*, or `nil` if the object does not exist. Examples:

```
(if (real-id fred) (delete fred))
```

deletes "fred" if it exists but reports no error if it doesn't, and

```
(if (= (real-id targetgeom) (real-id World)) () (delete targetgeom))
```

deletes "targetgeom" if it is different from the World.

`(redraw` *cam-id*`)`

States that the view in *cam-id* should be redrawn on the next pass through the main loop or the next invocation of `draw`.

`(regtable)`

shows the internal interest table; for debugging only.

`(rehash-emodule-path)`

Rebuilds the application (external module) browser by reading all .geomview-* files in all directories on the emodule-path. Primarily intended for internal use; any applications defined by (emodule-define ...) commands outside of the .geomview-* files on the emodule-path will be lost. Does not sort the entries in the brower; see (emodule-sort) for that.

`(replace-geometry` *geom-id* `PART-SPECIFICATION` *geometry*`)`

Replace a part of the geometry for *geom-id*.

`(rib-display [frame|tiff] FILEPREFIX)`

> Set Renderman display to framebuffer (popup screen window) or a TIFF format disk file. FILEPREFIX is used to construct names of the form "prefixNNNN.suffix". (i.e. foo0000.rib) The number is incremented on every call to `rib-snapshot` and reset to 0000 when `rib-display` is called. TIFF files are given the same prefix and number as the RIB file (i.e. foo0004.rib generates foo0004.tiff). The default FILEPREFIX is "geom" and the default format is TIFF. (Note that Geomview just generates a RIB file, which must then be rendered.)

`(rib-snapshot` *cam-id* `[`*filename*`])`

> Write Renderman snapshot (in RIB format) of *cam-id* to *filename*. If no filename specified, see `rib-display` for explanation of the filename used.

`(scale` *geom-id factor* `[`*y-factor z-factor*`])`

> Scale *geom-id*, multiplying its size by *factor*. The factors should be positive numbers. If *y-factor* and *z-factor* are present and non-zero, the object is scaled by *factor* in x, by *y-factor* in y, and by *z-factor* in z. If only *factor* is present, the object is scaled by *factor* in x, y, and z. Scaling only really makes sense in Euclidean space. Mouse-driven scaling in other spaces is not allowed; the scale command may be issued in other spaces but should be used with caution because it may cause the data to extend beyond the limits of the space.

`(scene` *cam-id* `[`*geometry*`])`

> Make *cam-id* look at *geometry* instead of at the universe.

`(set-clock` *time*`)`

> Adjusts the clock for this command stream to read *time* (in seconds) as of the moment the command is received. See also sleep-until, clock.

`(set-conformal-refine` *cmx* `[`*n* `[`*showedges*`]])`

> Sets the parameters for the refinement algorithm used in drawing in the conformal model. *cmx* is the cosine of the maximum angle an edge can bend before it is refined. Its value should be between -1 and 1; the default is 0.95; decreasing its value will cause less refinement. *n* is the maximum number of iterations of refining; the default is 6. *showedges*, which should be `no` or `yes`, determines whether interior edges in the refinement are drawn.

`(set-emodule-path (`*path1* ... *pathn*`))`

> Sets the search path for external modules. The *path*i should be pathnames of directories containing, for each module, the module's executable file and a .geomview-<modulename> file which contains an (emodule-define ...) command for that module. This command implicitly calls (rehash-emodule-path) to rebuild the application brower from the new path setting.

`(set-load-path (PATH1 ... PATHN))`

Sets search path for command, geometry, etc. files. The PATHi are strings giving the pathnames of directories to be searched.

`(set-motionscale X)`

Set the motion scale factor to X (default value 0.5). These commands scale their motion by an amount which depends on the distance from the frame to the center and on the size of the frame. Specifically, they scale by dist + scaleof(frame) * motionscale where dist is the distance from the center to the frame and motionscale is the motion scale factor set by this function. Scaleof(frame) measures the size of the frame object.

`(setenv name string)`

sets the environment variable "name" to the value "string"; the name is visible to Geomview (as in pathnames containing $name) and to processes it creates, e.g. external modules.

`(sgi)` Returns t if running on an sgi machine, nil if not

`(shell` *shell-command*`)`

Execute the given UNIX *shell-command* using /bin/sh. Geomview waits for it to complete and will be unresponsive until it does.

`(sleep_for TIME)`

Suspend reading commands from this stream for TIME seconds. Commands already read will still be executed; "sleep-for" inside "progn" won't delay execution of the rest of the progn's contents.

`(sleep-until TIME)`

Suspend reading commands from this stream until TIME (in seconds). Commands already read will still be executed; "sleep-until" inside "progn" won't delay execution of the rest of the progn's contents. Time is measured according to this stream's clock, as set by "set-clock"; if never set, the first sleep-until sets it to 0 (so initially (sleep-until TIME) is the same as (sleep-for TIME)). Returns the number of seconds until TIME.

`(snapshot` *cam-id filename*`)`

Save a snapshot of *cam-id* in IRIS rgb image format in file *filename* (a string). The window is popped above all other windows and redrawn before taking the snapshot.

`(soft-shader` *cam-id* `{on|off|toggle})`

Select whether to use software or hardware shading in that camera.

`(space {euclidean|hyperbolic|spherical})`

Set the space associated with the world.

`(stereowin` *cam-id* `[`*layout*`] [`*gapsize*`])`

Configure *cam-id* as a stereo window. *layout* should be one of

`no` entire window is a single pane, stereo disabled

`horizontal`
split left/right: left is stereo eye#0, right is #1.

`vertical` split top/bottom: bottom is eye#0, top is #1.

`colored` panes overlap, red is stereo eye#0, cyan is #1.

A gap of *gapsize* pixels is left between subwindows; if omitted, subwindows are adjacent. If both *layout* and *gapsize* are omitted, e.g. `(stereowin` *cam-id*`)`, returns current settings as a `(stereowin` ...`)` command list. This command doesn't set stereo projection; use `merge camera` or `camera` to set the stereyes transforms, and `merge window` or `window` to set the pixel aspect ratio & window position if needed.

`(time-interests` *deltatime initial prefix* `[`*suffix*`])`

Indicates that all interest-related messages, when separated by at least *deltatime* seconds of real time, should be preceded by the string *prefix* and followed by *suffix*; the first message is preceded by *initial*. All three are printf format strings, whose argument is the current clock time (in seconds) on that stream. A *deltatime* of zero timestamps every message. Typical usages:

```
(time-interests .1 "(set-clock %g)" "(sleep-until %g)")
(time-interests .1 "(set-clock %g)"
"(sleep-until %g) (progn (set-clock %g)" ")")
(time-interests .1 "(set-clock %g)"
"(if (> 0 (sleep-until %g)) (" "))"
```

`(transform` *object-id center-id frame-id* `[rotate|translate|translate-scaled] x y z)`

Apply a motion (rotation or translation) to object *object-id*; that is, construct and concatenate a transformation matrix with *object-id*'s transform The 3 *id*'s involved are the object that moves, the center of motion, and the frame of reference in which to apply the motion. The center is easiest understood for rotations: if *center-id* is the same as *object-id* then it will spin around its own axes; otherwise the moving object will orbit the center object. Normally *frame-id*, in whose coordinate system the (mouse) motions are interpreted, is `focus`, the current camera. Translations can be scaled proportional to the distance between the target and the center. Support for spherical and hyperbolic as well as Euclidean space is built-in: use the "space" command to change spaces. With type "rotate" x, y, and z are floats specifying angles in radians. For types "translate" and "translate-scaled" x, y, and z are floats specifying distances in the coordinate system of the center object.

`(transform-incr` *object-id center-id frame-id* `[rotate|translate|translate-scaled]` *x y z* `[`*dt*`])`

Apply continuing motion: construct a transformation matrix and concatenate it with the current transform of *object-id* every refresh (sets *object-id*'s incremental transform). Same syntax as transform. If optional *dt* argument is present, the object is moved at each time step such that its average motion equals one instance of the motion per *dt* seconds. E.g. `(transform-incr World World World rotate 360 0 0 10.0)` rotates the World about its X axis at 360 degrees every 10 seconds.

`(transform-set` *object-id center-id frame-id* `[rotate|translate|translate-scaled]` *x y z*`)`

Set *objectid*'s transform to the constructed transform. Same syntax as transform.

`(ui-center` *id*`)`

Set the center for user interface (i.e. mouse) controlled motions to object *id*.

`ui-emotion-program`

is an obsolete command. Use its new eqivalent `emodule-define` instead.

`ui-emotion-run`

is an obsolete command. Use its new eqivalent `emodule-start` instead.

`(ui-panel` *panelname* `{on|off} [` *window* `] )`

Do or don't display the given user-interface panel. Case is ignored in panel names. Current *panelnames* are:

geomview	main panel
tools	motion controls
appearance	appearance controls
cameras	camera controls
lighting	lighting controls
obscure	obscure controls
materials	material properties controls
command	command entry box
credits	Geomview credits

By default, the *Main* and *Tools* panels appear when Geomview starts. If the optional Window is supplied, a `position` clause (e.g. `(ui-panel obscure on { position xmin xmax ymin ymax })` sets the panel's default position. (Only xmin and ymin values are actually used.) A present but empty Window, e.g. `(ui-panel obscure on {})`" causes interactive positioning.

`(ui-target` *id* `[yes|no])`

Set the target of user actions (the selected line of the target object browser) to *id*. The second argument specifies whether to make *id* the current object regardless of its type. If `no`, then *id* becomes the current object of its type (geom or camera). The default is `yes`. This command may result in a change of motion modes based on target choice.

`(uninterest (`*command* `[`*args*`]))`

Undoes the effect of an `interest` command. (*command* [*args*]) must be identical to those used in the `interest` command.

`(update [timestep_in_seconds])`

Apply each incremental motion once. Uses timestep if it's present and nonzero; otherwise motions are proportional to elapsed real time.

`(update-draw` *cam-id* `[`*timestep*`])`

Apply each incremental motion once and then draw *cam-id*. Applies *timestep* seconds' worth of motion, or uses elapsed real time if *timestep* is absent or zero.

`(window` *cam-id window*`)`

Specify attributes for the window of *cam-id*, e.g. its size or initial position, in the OOGL Window syntax. The special *cam-id* `default` specifies properties of future windows (created by `camera` or `new-camera`).

`(winenter` *cam-id*`)`

Tell Geomview that the mouse cursor is in the window of *cam-id*. This function is for development purposes and is not intended for general use.

`(write {command,geometry,camera,transform,window}` *filename* `[`*id*`|(`*id* ...`)] [self|world|universe|other`

write description of *id* in given format to *filename*. Last parameter chooses coordinate system for geometry & transform:

`self`	just the object, no transformation or appearance (geometry only)
`world`	the object as positioned within the World.
`universe`	object's position in universal coordinates; includes Worldtransform

`other` *id*`: the object transformed to` *id*`'s coordinate system.`

A filename of '`-`' is a special case: data are written to the stream from which the 'write' command was read. For external modules, the data are sent to the module's standard input. For commands not read from an external program, '`-`' means Geomview's standard output. (See also the `command` command.)

The *id* can either be a single id or a parenthesized list of ids, like `g0` or `(g2 g1 dodec.off)`.

`(write-sexpr` *filename lispobject*`)`

Writes the given *lispobject* to *filename*. This function is intended for internal debugging use only.

`(xform` *id transform*`)`

Concatenate *transform* with the current transform of the object (apply *transform* to object *id*).

`(xform-incr` *id transform*`)`

Apply continual motion: concatenate *transform* with the current transform of the object every refresh (set object ID's incremental transform to *transform*).

`(xform-set` *id transform*`)`

Overwrite the current object transform with *transform* (set object ID's transform to *transform*).

`(zoom` *cam-id factor*`)`

Zoom *cam-id*, multiplying its field of view by *factor*. *factor* should be a positive number.

8 Non-Euclidean Geometry

Geomview supports hyperbolic and spherical geometry as well as Euclidean geometry. The three buttons at the bottom of the *Main* panel are for setting the current geometry type.

In each of the three geometries, three models are supported: *Virtual*, *Projective*, and *Conformal*. You can change the current model with the *Model* browser on the *Camera* panel. Each Geomview camera has its own model setting.

The default model is all three spaces is *Virtual*. This corresponds to the camera being in the same space as, and moving under the same set of transformations as, the geometry itself.

In Euclidean space *Virtual* is the most useful model. The other models were implemented for hyperbolic and spherical spaces and just happen to work in Eucldiean space as well: *Projective* is the same as *Virtual* but by default displays the unit sphere, and *Conformal* displays everything inverted in the unit sphere.

In hyperbolic space, the *Projective* model setting gives a view of the projective ball model of hyperbolic 3-space imbedded in Euclidean space. The camera is initially outside the unit ball. In this model, the camera moves by Euclidean motions and geometry moves by hyperbolic motions. *Conformal* model is similar but shows the conformal ball model instead.

In spherical space, the *Projective* model gives a view of half of the 3-sphere imbedded in Euclidean 3-space. Spherical motions give rise to projective transformations in the *Projective* model, and to Möbius transformations in the *Conformal* model. In both of these models the camera moves by Euclidean motions.

In *Projective* and *Conformal* models, the unit sphere is drawn by default. You can turn it off and on at will using the *Draw Sphere* button in the *Camera* panel. In the *Conformal* model, polygons and edges are subdivided as necessary to make them look curved. The parameters which determine this subdivision can be set with the `set-conformal-refine` gcl command.

There are several sample hyperbolic space objects in the '`data/geom/hyperbolic`' subdirectory of the Geomview directory ('`/u/gcg/ngrap/data/geom/hyperbolic`' on the Geometry Center's system). Likewise, the subdirectory '`data/geom/spherical`' contains several sample spherical space objects.

9 Mathematica Graphics in Geomview or RenderMan

Geomview comes with some Mathematica packages that let you use use Geomview to display Mathematica graphics. Mathematica is a commercial mathematical software system available from Wolfram Research, Inc.

There are two ways to do this.

1. Use Mathematica to write a graphics object to a file in OOGL format or in RIB format.
2. Use Geomview as the default display for all 3D graphics output in Mathematica.

You can also use these packages to save Mathematica graphics in RenderMan (RIB) format.

Since the format of Mathematica graphics objects is different from the OOGL formats, both of these methods involve translating Mathematica graphics to OOGL format. Geomview is distributed with a Mathematica package which does this translation. Before doing either of the above you must install this package.

9.1 Using Mathematica to generate OOGL files

The package '`OOGL.m`' allows Mathematica to write graphics objects in OOGL format. To use it, give the command `<< OOGL.m` to Mathematica to load the package. The `WriteOOGL[`*file*`,`*graphics*`]` command writes an OOGL description of the 3D graphics object *graphics* to the file named *file*.

This package also provides the `Geomview` command which sends a 3D graphics object to Geomview. The first time you use this command it starts up a copy of Geomview. Later calls send the graphics to the same Geomview. There are two ways to use the `Geomview` command.

`Geomview[`*graphics*`]`

> Sends the 3D graphics object *graphics* to Geomview as a geom named `Mathematica`. Subsequent usage of `Geomview[`*graphics*`]` replaces the `Mathematica` object in Geomview with the new *graphics*.

`Geomview[`*name*`,`*graphics*`]`

> Sends the 3D graphics object *graphics* to Geomview as a geom named *name*. You can use multiple calls of this form with different names to cause Geomview to display several Mathematica objects at once and allow independent control over them.

```
% math
Mathematica 2.0 for SGI Iris
Copyright 1988-91 Wolfram Research, Inc.
 -- GL graphics initialized --

In[1] := <<OOGL.m

In[2] := Plot3D[Sin[x + Sin[y]], {x,-2,2},{y,-2,2}]

Out[2] := -Graphics3D-
```

This displays graphics in the usual Mathematica way here.

```
In[3] := WriteOOGL["math.oogl", %2]

Out[3] := -Graphics3D-
```

This displays nothing new but writes the file '`math.oogl`'. You can now load that file into Geomview on any computer. Alternately, you can use the `Geomview` command to start up a copy of Geomview from within Mathematica.

```
In[5] := Geomview[%2]

Out[5] := -Graphics3D-
```

9.2 Using Geomview as Mathematica's Default 3D Display

The package '`Geomview.m`' arranges for Geomview to be the default display program for 3D graphics in Mathematica. To load it, give the command `<< Geomview.m` to Mathematica. Thereafter, whenever you display 3D graphics with `Plot3D` or `Show`, Mathematica will send the graphics to Geomview.

Loading '`Geomview.m`' implicitly loads '`OOGL.m`' as well, so you can use the `Geomview` and `WriteOOGL` as described above after loading '`Geomview.m`'. You do not have to separately load '`OOGL.m`'.

```
% math
Mathematica 2.0 for SGI Iris
Copyright 1988-91 Wolfram Research, Inc.
 -- GL graphics initialized --

In[1] := <<Geomview.m

In[2] := Plot3D[x^2 + y^2, {x, -2, 2}, {y, -2, 2}]

Out[2] := -SurfaceGraphics-
```

This invokes geomivew and loads the graphics object into it.

```
In[3] := Plot3D[{x*y + 6, RGBColor[0,x,y]}, {x,0,1}, {y,0,1}]

Out[3] := -SurfaceGraphics-
```

This replaces the previous Geomview object by the new object.

```
In[4] := Geomview[{%2,%3}]

Out[4] := {-SurfaceGraphics-, -SurfaceGraphics-}
```

This displays both objects at once. You also can have more than one Mathematica object at a time on display in Geomview, and have separate control over them, by using the `Geomview` command with a name, See Section 9.1 [OOGL.m], page 105.

```
In[5] := Graphics3D[ {RGBColor[1,0,0], Line[{ {2,2,2},{1,1,1} }] }]

Out[5] := -Graphics3D-

In[6] := Geomview["myline", %5]
```

This addes the `Line` specified in `In[5]` to the existing Geomview display. It can be controlled independently of the "Mathematica" object, which is currently the list of two plots.

```
In[7] := <<GL.m
```

If you're on an SGI, loading `GL.m` returns Mathematica to its usual 3D graphics display. To do this on a NeXT you should load `PSDirect.m` if you are using Mathematica in a notebook, or `NeXT.m` if you invoked Mathematica from a shell. The following plot will appear in a normal static Mathematica window.

```
In[8] := ParametricPlot3D[{Sin[x],Sin[y],Sin[x]*Cos[y]}, {x,0,Pi},{y,0,Pi}]

Out[8] := -Graphics3D-
```

We can return to Geomview graphics at any time by reloading '`Geomview.m`'.

```
In[9] := <<Geomview.m

In[10] := Show[%8]

Out[10] := -Graphics3D-

In[11] := ParametricPlot3D[
   {(2*(Cos[u] + u*Sin[u])*Sin[v])/(1 + u^2*Sin[v]^2),
   (2*(Sin[u] - u*Cos[u])*Sin[v])/(1 + u^2*Sin[v]^2),
   Log[Tan[v/2]] + (2*Cos[v])/(1 + u^2*Sin[v]^2)},
  {u,-4,4},{v,.01,Pi-.01}]

Out[11] := -Graphics3D-
```

This last plot is Kuen's surface, a surface of constant negative curvature. Parametrization from Alfred Gray's *Modern Differential Geometry of Curves and Surfaces* textbook.

9.3 Using Mathematica to generate RenderMan files

In addition to the `WriteOOGL` and `Geomview` commands described above, the package '`OOGL.m`' also defines the command `WriteRIB` which writes a 3D graphics object to a RenderMan RIB file:

`WriteRIB[`*file*`,` *graphics*`]` writes *graphics* to file *file*. RenderMan is a commercial rendering system available from Pixar, Inc., which can produce extremely high quality images.

```
In[1] := <<OOGL.m

In[2] := <<Graphics/Polyhedra.m

In[3] := Graphics3D[Cube[]]

Out[3] := -Graphics3D-

In[4] := WriteRIB["cube.rib", %3]

Out[4] := -Graphics3D-
```

This generates the file '`math.rib`'. This is a ready-to-render RIB file of the given geometry, using a default camera position, lighting, and the "plastic" shader. In a shell window, type `render cube.rib` to generate the image file '`mma.tiff`'. Of course, you need to have RenderMan installed for this to work. A shortcut to render from inside Mathematica is `WriteRIB["!render", foo]`.

`WriteRIB` works by first converting the Mathematica graphics object to OOGL format using `WriteOOGL` and then calls an external program '`oogl2rib`' to convert OOGL to RIB format. The oogl2rib program takes several options which you can specify in a string as an optional third argument to `WriteRIB`. The default option string is `" -n mma.tiff "`, which indicates that the RIB file should generate a rendered TIFF file named '`mma.tiff`'. A particularly useful option is `-g`, which tells oogl2rib to convert only the geometry into a RIB fragment. You can insert that fragment into a full RIB file of your own making with camera positions and shaders of your choice, to harness the full power of RenderMan.

The full usage of oogl2rib is:

`oogl2rib [-n` *name*`] [-B` *r*,*g*,*b*`] [-w` *width*`] [-h` *height*`] [-fgb] [`*infile*`] [`*outfile*`]`

By default it reads from stdin and writes to stdout. Either *infile* or *outfile* may be '`-`', which means use stdin/stdout. The options are:

`-n` *name*	Use *name* for the name of the rendered TIFF file (default "geom.tiff") or framebuffer window (default "geom.rib").
`-B` *r*,*g*,*b*	Use background color (*r*,*g*,*b*). Each component ranges from 0 to 1. Default: none.

`-w` *width* `-h` *height*	Rendered frame will be *width* by *height* pixels.
`-f`	RIB file renders to on-screen framebuffer instead of TIFF file.
`-g`	Output only the geometry in RIB format.
`-b`	Output only a Quick Renderman clip object. Ignores -nBwhf.

9.4 Using Geomview and Mathematica on Different Computers

It is possible to use Geomview to display graphics generated by Mathematica running on a different computer. If each computer is either an SGI or a NeXT and they are networked together, you can tell Mathematica to use a remote host for Geomview graphics. If you want to use Mathematica on a computer that is not networked with your Geomview computer, or on any kind of computer other than an SGI or a NeXT (for example a PC or a Mac), you can write out *chunk* files in Mathematica which you transfer to the Geomview computer and then translate to OOGL format.

9.4.1 Using a Networked Geomview Host

The `Geomview` command looks at the `DISPLAY` or `REMOTEHOST` environment variables to try to determine if you are logged in from another computer. If either of these indicates that you are, `Geomview` will attempt to run Geomview on that computer. In order for this to work, your network must be configured such that the Mathematica computer can successfully `rsh` to the Geomview computer without giving a password.

You can also explicitly set the `DisplayHost` option to the `Geomview` command to a string which is the desired hostname, for example:

```
In[1] := << OOGL.m

In[2] := Plot3D[Sin[x + Sin[y]], {x,-2,2},{y,-2,2}]

Out[2] := -Graphics3D-

In[3] := Geomview[%3, DisplayHost->"riemann"]
```

This displays the graphics `%3` on the remote host named `riemann`.

`Geomview` recognizes the string `"local"` as a value for `$DisplayHost`; it forces the graphics to be displayed on the local machine.

In addition to knowing the name of the machine you want to run Geomview on, the `Geomview` needs to know the type of that machine (SGI or NeXT). By default, `Geomview` assumes that it is the same kind of computer as the one you are running Mathematica on. The `MachType` option lets you explicitly specify the type of the `DisplayHost` computer; it should be one of the strings `"sgi"` or `"next"`.

You can use `SetOptions` to change the default `DisplayHost` and `MachType`. For example,

```
In[4] := SetOptions[Geomview, DisplayHost->"riemann", MachType->"sgi"]
```

arranges for `Geomview` to run Geomview on an SGI workstation named `riemann`.

9.4.2 Transporting Mathematica Files to Geomview by Hand

The auxilliary function `WriteChunk` is for those who can only use Mathematica on a non-Unix machine (Mac, PC) or a Unix machine that is not on a network with an SGI or NeXT. `WriteChunk[`*file*`,` *graphics*`]` generates a file named *file* which contains the graphics object *graphics* in the format accepted by '`math2oogl`'.

You can transfer that file to a computer that has Geomview installed on it and then use the programs '`math2oogl`', '`oogl2rib`', and '`geomview`' directly from the shell. These programs are distributed in the '`bin/sgi`' (on SGIs) or '`bin/next`' (on NeXTs) subdirectory of the Geomview directory, and may have been installed so that they are on your `path`.

```
In[1]:= <<OOGL.m

In[2]:= Plot3D[ Sin[x + Sin[y]], x,-2,2, y,-2,2 ]

Out[2]= -SurfaceGraphics-

In[3]:= WriteChunk["mychunk",%2]
```

This writes the file '`mychunk`' which contains a description of the graphics object. You can then transfer this file to an SGI or NeXT and type

```
math2oogl < mychunk > mma.oogl
```

to convert it to the OOGL file '`mma.oogl`' which you can then view using Geomview. This is the equivalent of the `WriteOOGL` command.

For a result equivalent to the `Geomview` or `Show` commands, type

```
math2oogl -togeomview Mathematica geomview < mychunk
```

The `WriteRIB` command can be emulated from the shell as

```
math2oogl < mychunk | oogl2rib -n mma.tiff
```

9.5 Details of the Mathematica->Geomview Package

The '`OOGL.m`' package uses the external program '`math2oogl`' to convert `Graphics3D` objects to OOGL format, because a compiled external program is able to do this conversion many times faster than Mathematica.

The converter will sometimes handle colored SurfaceGraphics objects correctly that Mathematica does not handle correctly, which means that Geomview[object] sometimes works where Show[object] will give errors.

The converter supports the `Polygon`, `Line`, and `Point` graphics primitives, `RGBColor` `Graphics3D` directives, and `SurfaceGraphics` objects with or without `RGBColor` directives, and lists of any combination of these. It silently ignores all other directives.

The Mathematica to RenderMan conversion is actually a two-step process: Mathematica->OOGL (math2oogl), and OOGL->RenderMan (oogl2rib). The math2oogl program has only been tested on SGIs and NeXTs, but could theoretically compile on any machine. The oogl2rib program depends on the OOGL (Object Oriented Graphics Language) libraries, which now only exist on SGI and NeXT machines.

In the `WriteOOGL` and `WriteRIB` commands, filename can either be a string containing a filename, an `OutputStream` object, or a string starting with a `!` to send the output to a command. Object can be a `Graphics3D` object, a `SurfaceGraphics` object, or a list of these.

The packages work best with Mathematica 2.0 or better. With version 1.2, the Geomview display is always on the local host.

9.6 Installing the Mathematica Packages

If Geomview is properly installed on your system according to the instructions in See Chapter 10 [Installation], page 115, then the Mathematica-to-Geomview packages should work as described here; there should be no need for additional installation procedures. In practice, however, it is sometimes necessary to taylor the installation of the Mathematica packages and/or of Geomview itself to suit the needs of a particular system. This section contains details about how the installation works; if the Mathematica-to-Geomview connection does not seem to work for you after following the Geomview installation procedure, consult this section to see what might need to be fixed.

In this section, the phrase *Geomview installation* refers any of the procedures in See Chapter 10 [Installation], page 115. The way the Mathematica packages work and are installed is the same regardless of whether you have one of the binary distributions or the source distribution.

1. The relevant mathematica files are '`OOGL.m`', '`Geomview.m`', and '`BezierPlot.m`'; Mathematica must be able to find these files. They are distributed in the '`$GEOMROOT/mathematica`' subdirectory of the binary distributions, and in the '`$GEOMROOT/src/bin/geomutil/math2oogl`' subdirectory of the source distribution. These files need to be in a directory that is on Mathematica's search path. You can look at the value of the `$Path` variable in a Mathematica session on your system to see a list of the directories on Mathematica's search path.

   The Geomview installation procedure puts copies of the Mathematica packages into a directory that you specify (`MMAPACKAGEDIR`). This should ensure that Mathematica can find them. Alternately, you could arrange to append the pathname of the Mathmematica package subdirectory of the Geomview distribution to the `$Path` variable each time you run Mathematica.
2. The package '`OOGL.m`' needs to be able to invoke the programs '`geomview`', '`math2oogl`', and '`oogl2rib`'. The Geomview installation procedure installs these programs into a directory that you specify for executables (`BINDIR`). Ideally, this directory should be on your shell's `$path`. More specifically, it should be on the `$path` of the shell in which Mathematica runs; the directory '`/usr/local/bin`' is usually a good choice. You can see the list of directories on this path by giving the command `!echo $path` in Mathematica.

If for some reason you can't arrange for '`geomview`', '`math2oogl`', and '`oogl2rib`' to be in a directory on the shell's `$path`, you can modify '`OOGL.m`' to cause it to look for them using absolute pathnames. To do this, change the definitions of the variables `$GeomviewPath` and `$GeomRoot`, which are defined near the top of the file. Change `$GeomviewPath` to the absolute pathname of the '`geomview`' shell script on your system. Change `$GeomRoot` to the absolute pathname of the '`$GEOMROOT`' directory on your system. If you do this, you should also make sure there are copies of '`geomview`', '`math2oogl`', and '`oogl2rib`' in the '`$GEOMROOT/bin/sgi`' (on an SGI) or '`$GEOMROOT/bin/next`' (on a NeXT) directory.

3. The '`geomview`' shell script, which '`OOGL.m`' uses to invoke Geomview, needs to be able to find the geomview executable file (which is called '`gvx`' on the SGI and '`Geomview.app/Geomview`' on the NeXT). The Geomview installation procedure should have been taken care of this, but if your Mathematica session doesn't seem to be able to invoke Geomview, it's worth double-checking that the settings in the '`geomview`' script are correct.

10 Installation

What you do to install Geomview depends on which kind of computer you have (SGI or NeXT) and on whether you have the source distribution or the binary distribution.

In general, if you don't care about looking at Geomview's source code, you should get the binary distribution. Its installation is much easier and quicker than that for the source code.

10.1 Installing the SGI Binary Distribution

If you have just obtained a copy of the SGI binary distribution (file '`geomview-sgi.tar.Z`'), you should be able to run Geomview and make use of most of its features immediately after unpacking it by `cd`'ing to the directory that it is in and typing `geomview`.

In order to fully install Geomview so that you can run it from any directory and use all of its features, follow the steps in this section. In particular, you must go through this installation procedure in order to use Geomview to display Mathematica graphics.

Geomview is distributed in a directory that contains various files and subdirectories that Geomview needs at run-time, such as data files and external modules. It also contains other things distributed with Geomview, such as documentation and (in the soure-code distribution) source-code. We refer to the root directory of this tree as the '`$GEOMROOT`' directory. This is the directory called '`Geomview`' that is created when you unpack the distribution file.

To install Geomview on your system, arrange for the '`$GEOMROOT`' directory to be in a permanent place. Then, in a shell window, `cd` to that directory and type `install`. This runs a shell script which does the installation after asking you several questions about where you want to install the various components of Geomview.

After running the `install` script you should now be able to run Geomview from any directory on your system. (You may need to give the `rehash` command in any shells on your computer that were started up before you did the installation.)

The '`install`' script puts copies of the files in '`$GEOMROOT/bin/sgi`' and '`$GEOMROOT/man`' into the directories you specified for executables and man pages, respectively. Once you have done the installation you can cut down one the disk space required by Geomview by removing some files from these directories, since copies have been installed elsewhere. You should first test that your installed Geomview works properly because once you remove these files from their distribution directories you will not be able to do the installation again.

In particular, the files you can remove are

'`$GEOMROOT/bin/sgi`':
: Remove all files from here except '`gvx`', which is the geomview executable file. DO NOT REMOVE '`gvx`'. It is not installed elsewhere.

'`$GEOMROOT/man`':
: You can remove all the files in this directory.

10.1.1 Details of the SGI Binary Installation

The `install` script should be self-explanatory; just run it and answer the questions. This section gives some details for system administrators and other users who may want to know more about the installation.

The installation is actually done by `make`; the `install` script queries the user for the settings of the following `make` variables and then invokes `make install`.

`GEOMROOT`: the absolute pathname of the Geomview root directory. The `geomview` shell script, which is what users invoke to run Geomview, uses this to set various environment variables that Geomview needs. It is very important that this be an *absolute* pathname — i.e. it should start with a '/'.

`BINDIR`: a directory where executable files are installed. The `geomview` shell script goes here, as well as various other auxiliary programs that can be used in conjunction with `geomview`. This should be a directory that is on users' '`$path`'. These auxiliary programs are distributed in the '`$GEOMROOT/bin/sgi`' directory; if you specify this directory for `BINDIR`, they are left in that directory.

`MANDIR`: a directory where Unix manual pages are installed. These are distributed in the '`$GEOMROOT/man`' subdirectory; if you specify this directory for `MANDIR`, they are left in that directory.

`MMAPACKAGEDIR`:
: a directory where Mathematica packages are installed. This should be a directory that Mathematica searches for packages that it loads; you can see what directories your Mathematica searches by looking at the value of the `$Path` variable in a Mathematica session. The installation process will install some packages there which al-

low you to use Geomview to display Mathematica graphics. These packages are distributed in the '`$GEOMROOT/mathematica`' subdirectory; if you specify this directory for `MMAPACKAGEDIR`, or if you specify the empty string for `MMAPACKAGEDIR`, the packages are left in that directory. For more details about the way these Mathematica packages connect to Geomview, see Section 9.6 [Package Installation], page 113.

10.2 Installing the NeXT Binary Distribution

1. If you have just obtained a copy of the NeXTStep binary distribution (file '`geomview-next.tar`'), you can unpack it by double-clicking on it in the Workspace. This will open up a File Viewer panel showing, among other things, a NeXT Installer package called '`Geomview.pkg`'.
2. The first thing you should do is double-click on '`Geomview.pkg`' to invoke the NeXT Installer. You will be asked where you want to install it; typically it should go in '`/LocalApps`' or in '`~/Apps`' in your home directory. You should now be able to run Geomview and make use of most of its features by double-clicking on the installed '`Geomview.app`' icon.
3. There are some aspects of the installation, however, that the NeXT Installer can't handle. In order to fully install Geomview so that you can use all of its features, you should run the '`install`' script in the '`Geomview.app`' directory. In particular, you must go through this installation procedure in order to use Geomview to display Mathematica graphics.

   To run the '`install`' script you can open '`Geomview.app`' in the Workspace by selecting it and picking `File->Open as Folder` from the Workspace menu. This will pop up a File Viewer panel showing the contents of '`Geomview.app`'. Scroll down to the file named '`install`', and double-click on it. This will open a terminal window and run the script in that window. Alternately, you can open a terminal window yourself, cd to '`Geomview.app`', and run '`install`' there.

   The '`install`' script does the installation after asking you several questions about where you want to install the various components of Geomview. After running the `install` script, Geomview is completely installed. If in the future you move '`Geomview.app`' to some other location you should run '`install`' again.
4. This step is optional. Geomview's example data files are in the '`Geomview.app/data`' directory. If you are on a network with both SGI workstations and NeXTStep workstations, and you want to install Geomview to run on both, you can save disk space by having the two installations share a common data directory. To do this, decide on a location for the data directory and copy it there if it isn't there already (a good choice would be to leave it in the '`$GEOMROOT`' directory in your SGI Geomview installation). Then edit the file '`Geomview.app/CONFIG.gv`' to change the setting of the variable `GEOMVIEW_DATA` to point to this directory (there are comments in the file telling you what to do). You can then remove the data directory from '`Geomview.app`'.

To run geomview, double-click on '`Geomview.app`' from the workspace, or type `open Geomview.app` from the appropriate directory, or type `geomview` from a shell window.

More Geomview documentation is in the '`Geomview.app/doc`' subdirectory. In particular, a copy of the manual is there.

The '`install`' script puts copies of the files in '`Geomview.app/bin/next`' and '`Geomview.app/man`' into the directories you specified for executables and man pages, respectively. Once you have done the installation you can cut down one the disk space required by Geomview by removing all the files in these directories, since copies have been installed elsewhere. You should first test that your installed Geomview works properly because once you remove these files from their distribution directories you will not be able to do the installation again.

10.2.1 Details of the NeXTStep Binary Installation

Other than the installation of the '`Geomview.app`' directory, the installation details of the NeXTStep binary distribution are the same as for the SGI distribution, see Section 10.1.1 [SGI Binary Detail], page 116. Note that the directory referred to in the SGI distribution as '`$GEOMROOT`' is the '`Geomview.app`' directory in the NeXTStep distribution.

10.3 Compiling and Installing the Source Code Distribution

The main reason to get the source code distribution is to look at and/or work with the source code. If you are only concered with *using* Geomview it is better to get the binary distribution. It takes anywhere from 15 minutes to 1.5 hours to compile the entire source distribution, depending on what kind of computer you have.

Let '`$GEOMROOT`' denote the full pathname of the Geomview source code directory; this is the directory called '`Geomview`' that is created when you unpack the distribution. This directory contains the Geomview source code as well as various other files and subdirectories that Geomview needs when it runs.

Before doing any compilation you should edit the file '`$GEOMROOT/makefiles/mk.site.default`'. This file defines some `make` variables which specify your local configuration. This includes the pathnames of the directories into which Geomview will be installed, and possibly some other settings as well. There are comments in the file telling you what to do. This file is included by every Makefile in the source tree, so the settings you specify here are used throughout the source.

If you will be compiling for both SGI and NeXT, you can do both in the same directory tree. By default the Makefiles are set up to put the objects files, libraries, and executables in directories which depend on the type of computer, so the two architectures will not interfere with each other. The Makefiles use a variable called `CPU` to determine the type of machine. Before doing any compilation you must arrange for this variable to have a value. There are two ways you can do this.

1. If you will always be compiling Geomview on the same type of computer (SGI or NeXT), edit the file '`$GEOMROOT/makefiles/Makedefs.global`' to set the `CPU` variable to either `iris4` or `NeXT`. The comments near the top of that file will tell you where to do this.
2. If you will be compiling on both types of computers you can set a shell environment variable named `CPU` to either `iris4` or `NeXT`, and the Makefiles will inherit the value from the environment. The script '`$GEOMROOT/config`' determines which kind of computer you are on and sets this variable accordingly. To use this script, type `source config` in the (assuming a C-shell type shell) in the '`$GEOMROOT`' directory shell in which you plan to do the compilation. Or you can set the variable directly; it should be either `NeXT` or `iris4`. You will need to do this in every shell in which you plan to do compilation.

   Alternately, you could modify your shell initialization file ('`.cshrc`' or whatever) to set `CPU` appropriately.

Note that many of the Makefiles refer to a variable called `MACHTYPE` to determine the type of machine. This is set to either `sgi` or `next`, depending on the value of `CPU`.

Once you have configured your source tree by editing the files as described above and setting the `CPU` variable, you can compile and install Geomview by typing `make install` in the '`$GEOMROOT`' directory. You can also type `make all`, or equivalently just `make`, to compile without installing, and then type `make install` later to install.

You can use these same `make` comands in any subdirectory in the tree to recompile and/or install a part of Geomview or a module.

If you want to compile fat binaries under NeXTStep 3.1, before doing any compilation edit the file '`$GEOMROOT/makefiles/mk.next`' to uncomment a particular line there. There are comments in the file telling you which line to uncomment.

If you want to modify the complier flags used during compilation, edit the file '`$GEOMROOT/makefiles/Makedefs.global`'; the `COPTS` variable specifies the flags passed to the C compiler (cc).

10.4 Obtaining Geomview

Geomview is available free via anonymous ftp from Internet host '`geom.umn.edu`', IP address 128.101.25.35. The Geomview distribution files are in the '`pub/software/geomview`' subdirectory. They are all tar archive files ('`.tar`' or '`.tar.Z`' files), so you should use binary mode in ftp for transferring them to your site.

The main files are

'`geomview-sgi.tar.Z`'

The SGI binary distribution. Contains executables for running on any Silcon Graphics IRIS workstation, plus documentation and example files.

'`geomview-next.tar`'

The NeXTStep binary distribution. This contains fat binaries which will run on either a NeXT workstation running NeXTStep 3.0 or 3.1, or a 486 PC running NeXTStep 3.1. Also contains documentation and example files. This .tar file is not compressed because it contains the distribution compressed into a NeXT Installer package, and further compression actually increases the size of the file. To unpack '`geomview-next.tar`' on a NeXT, simple double-click on it in the Workspace.

'`geomview-src.tar.Z`'

The source code distribution; contains source code so you can compile Geomview and the distributed external modules on either an SGI or on a NeXT workstation running NeXTStep 3.0 or 3.1, or a 486 PC running NeXTStep 3.1. Also contains documentaion and examples files.

Each of the above archive files contains the entire distribution: executables or source for Geomview itself, plus all distributed external modules, example data files, and documentation. These archive files are therefore rather large. If you do not have enough disk space on your workstation for the entire distribution, various pieces of the distribution are available separately in the '`pub/software/geomview/pieces`' subdirectory. See the file '`README`' in that directory for details.

After retrieving any of the distribution archive files, you can unpack it with a command like the following

```
% uncompress < geomview-sgi.tar.Z | tar xvopf -
```

This will unpack the contents of the archive file into a subdirectory named '`Geomview`'. Once unpacked, you can delete the archive file.

The following is a sample ftp session for retreiving and unpacking the SGI binary distribution. After unpacking, see the file '`README`' for more information.

```
artin% ftp geom.umn.edu
Connected to geom.umn.edu.
220 cameron FTP server (Version 5.88 Thu Jun 25 16:41:41 CDT 1992) ready.
Name (geom.umn.edu:mbp): anonymous
331 For password please enter your e-mail address or name and institution.
Password:mbp@geom.umn.edu

230 Guest login ok, access restrictions apply.
ftp> cd pub/software/geomview
250 CWD command successful.
ftp> binary
200 Type set to I.
ftp> get geomview-sgi.tar.Z
200 PORT command successful.
150 Opening BINARY mode data connection for geomview-sgi.tar.Z (5815980 bytes).
226 Transfer complete.
local: geomview-sgi.tar.Z remote: geomview-sgi.tar.Z
5815980 bytes received in 28.67 seconds (1.98e+02 Kbytes/s)
ftp> quit
221 Goodbye.
artin% ls -l
total 5680
-rw-rw-r--  1 mbp      5815980 Aug 19 16:38 geomview-sgi.tar.Z
artin% uncompress < geomview-sgi.tar.Z | tar xvopf -
x ./CHANGES, 16910 bytes, 34 tape blocks
...
artin% rm geomview-sgi.tar.Z
```

Function Index

Table of Contents