

Geomview Manual

Geomview Version 1.6.1

for Unix

December 10, 1996

(updated January 5, 2000)

Mark Phillips et al.

Copyright © 1993, The Geometry Center
Copyright © 2000, Geometry Technologies, Inc.

Geomview, the interactive 3D viewing program.

Introduction to Geomview

Geomview is an interactive program for viewing and manipulating geometric objects, originally written by staff members of the Geometry Center at the University of Minnesota, starting in 1991. 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 many kinds of Unix computers, including Linux, SGI, Sun, and HP. This manual describes Geomview version 1.6.1.

Geomview is free software, available under the terms of the GNU General Public License; See [\[Copying\]](#), page [\[undefined\]](#) for details.

Geomview and this manual are available for download from '<http://www.geomview.org>' or '<ftp://ftp.geomview.org>'. Permission is granted to make copies of this manual.

If you have questions or comments about Geomview or this manual, consider joining in the '[geomview-users](#)' mailing list, which is a forum in which users of Geomview communicate to answer each others' questions and to share news about what they are doing with Geomview. The Geomview authors participate in this list and sometimes post answers to questions there. To join the list, send a note to geomview-users-request@geomview.org.

If you find a bug in Geomview, please report it to the Geomview team by sending a note to software@geomview.org.

Distribution

Geomview is *free software*; this means that everyone is free to use it and free to redistribute it on certain conditions. Geomview is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of Geomview that they might get from you. The precise conditions are found in the GNU General Public License that comes with Geomview and also appears following this section.

One way to get a copy of Geomview is from someone else who has it. You need not ask for our permission to do so, or tell any one else; just copy it. If you have access to the Internet, you can get the latest distribution version of Geomview by anonymous FTP from <ftp.geomview.org>, or through your web browser at www.geomview.org.

You may also receive Geomview when you buy a computer. Computer manufacturers are free to distribute copies on the same terms that apply to everyone else. These terms require them to give you the full sources, including whatever changes they may have made, and to permit you to redistribute the Geomview received from them under the usual terms of the General Public License. In other words, the program must be free for you when you get it, not just free for the manufacturer.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
 - a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
 - b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
 - c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions

for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore,

by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a

version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

one line to give the program's name and an idea of what it does.

Copyright (C) 19yy *name of author*

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy *name of author*
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'. This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright
interest in the program ‘Gnomovision’
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

History of Geomview's Development

Geomview was originally written at the Geometry Center at the University of Minnesota in Minneapolis. The Geometry Center was a research and education center funded by the National Science Foundation, with a mission to promote research and communication of mathematics. Much of the work there involved the use of computers to help visualize mathematical concepts.

The project that eventually led to Geomview began in the summer of 1988 with the work of Pat Hanrahan on a viewing program called MinneView. Shortly thereafter Charlie Gunn began developing OOGL (Object Oriented Graphics Language) in conjunction with MinneView. Many people contributed to OOGL and MinneView, including Stuart Levy, Mark Meurer, Tamara Munzner, Steve Anderson, Mario Lopez, Todd Kaplan.

In 1991 the staff of the Geometry Center began work on a new improved version of OOGL, and a new and improved viewing program, which they called Geomview. At that time essentially the only game in town for interactive 3D graphics was Silicon Graphics (SGI), so Geomview was developed initially on SGI workstations, using IRIS GL. The first version was finished in January of 1992. It immediately became very popular among visitors to the Geometry Center, and through the Center's ftp archive (this was before the web) people at other institutions began using it too.

In addition to SGI workstations the Geometry Center had quite a few NeXT stations, so soon after Geomview was running on SGIs the staff developed a version for NeXTStep as well. By this time there were several thousand people using it around the world.

A few years later the staff ported Geomview to X windows and OpenGL, and eventually, with the demise of NeXT, the NeXT version fell by the wayside.

In its mission to foster communication among researchers and educators, the Geometry Center developed a web site, www.geom.umn.edu, in late 1993. It was one of the first 300 web sites in existence. A part of the web site was of course devoted to Geomview, and helped to spread the word about its existence.

The Geometry Center closed its "brick and mortar" facilities in August of 1998 (NSF cut its funding), but the web site continued to exist, and Geomview continued to be very popular around the world. In December of 1999 some of the former Geometry Center staff set up www.geomview.org as a permanent home on the web for Geomview.

Geomview's original authors, as well as a number of other volunteers around the world, are still actively involved in using and developing Geomview.

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 and Tim Rowley did the port to X windows. Many other Geometry Center staff members, as well as several people elsewhere, also contributed.

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.

How to Pronounce 'Geomview'

The word 'Geomview' is a combination of the first syllable of the word 'geometry', and the word 'view'. The authors pronounce it with the accent on the first syllable

GE-om-view

Some people put the accent on the second syllable, where it falls in the word 'geometry', but the original authors, who invented the name, prefer the accent-on-first-syllable pronunciation.

Let Us Hear From You

The developers of Geomview would like to find out how you are using Geomview. We use this information in deciding what features to focus on, and in finding ways to continue to support its development. If you find Geomview useful, please send us a letter telling us what you are doing with it. We may include a link to your work on the geomview.org web site (but we'll ask you about this before doing so).

Please send the letter via email to register@geomview.org.

If you are interested in contributing to the development of Geomview, there are several things you can do:

1. **volunteer programming work**

If you are a programmer and make an improvement to Geomview, contact the Geomview team by emailing software@geomview.org. In general, if you intend to work on Geomview very much please contact us so that we can coordinate your work with other development work.

2. **volunteer documentation work**

Geomview also needs updated documentation; if you use Geomview a lot and are familiar with it, you can help by working on revised documentation. For information on this, email software@geomview.org.

3. **contract with Geometry Technologies**

Geometry Technologies, Inc. is a consulting firm that provides contract technical support and custom programming services in the area of 3D graphics. This includes a wide range of services related to 3D graphics, included but not limited to applications involving Geomview. To the extent that resources allow, Geometry Technologies supports the development of Geomview; in particular it hosts the www.geomview.org web site, and its staff make ongoing improvements to Geomview itself. If you are in a position to pay for technical support or custom programming work, contracting with Geometry Technologies indirectly supports Geomview. You can also contract with with Geometry Technologies to have particular features that you want added to Geomview. Geometry Technologies web site at www.geomtech.com, or email info@geomtech.com.

4. **make a donation**

If you want to donate money directly to support Geomview, you can do so online with a credit card at <https://secure.geomtech.com/geomview>. You can also send a check via regular mail to

Geometry Technologies, Inc.
77 Mid Oaks Ln.
Roseville, MN 55113
USA

Please include the name under which you wish the donation to be credited, either your own or the third-party if it is a gift on behalf of someone else, and an e-mail address to which we can acknowledge receipt. Geometry Technologies will not release this information to anyone else (although they may use it to contact you). We will send a physical receipt by normal mail for any donations of US\$100 or more, provided

you include a return address. Geometry Technologies is a private corporation, so for individuals within the U.S., donations are not tax-deductible.

Thank you.

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 is the product of an 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 3-D graphics output from Mathematica and Maple.

2 Tutorial

This chapter leads you through some of the basics of using Geomview. Work through this chapter in front of a computer 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 (Enter here means hit the "Enter" key):

```
geomview tetra dodec Enter
```

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.

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 *Targets* 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)

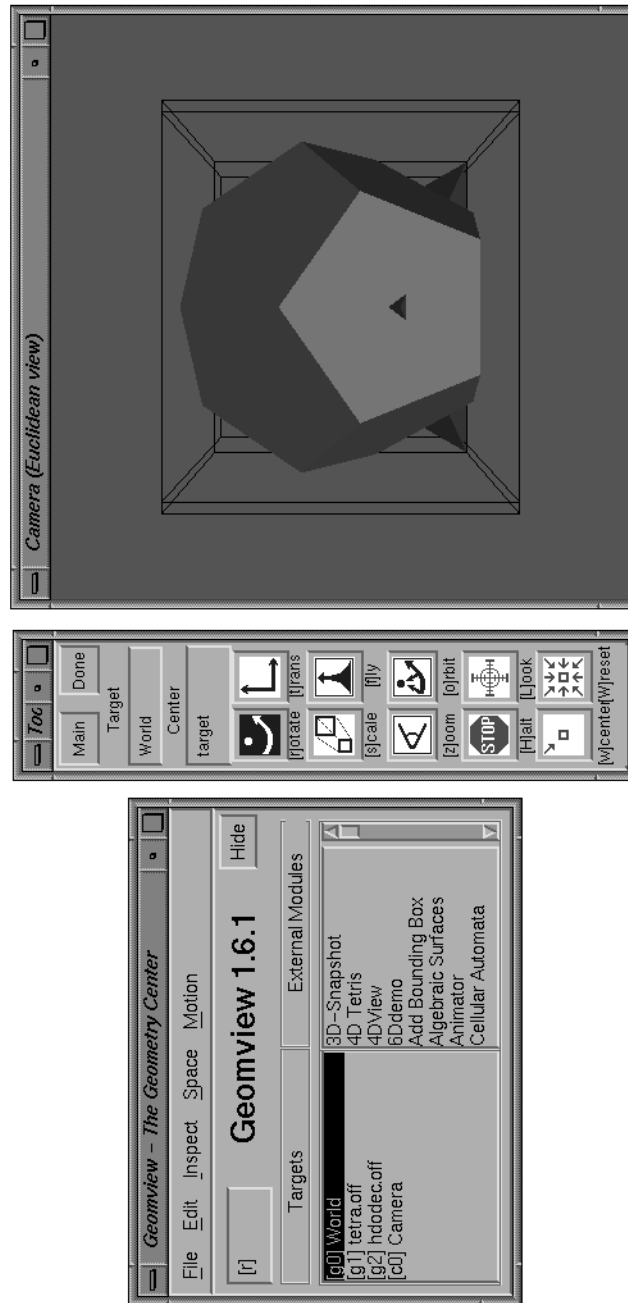


Figure 1: Initial Geomview display

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 *Targets* 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.

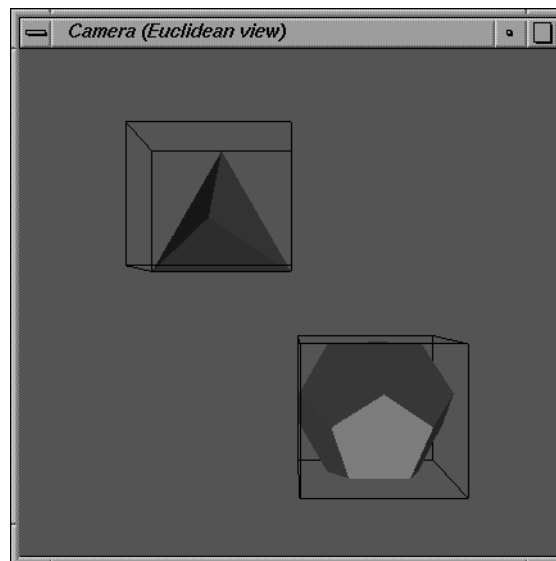


Figure 2: Looking at the world

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 *Targets* browser in the *Main* panel. Double-clicking the right mouse button on an object is another way to make it the target object.

Go to the *Inspect* menu at the top of the *Main* panel and select *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. Note the 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.

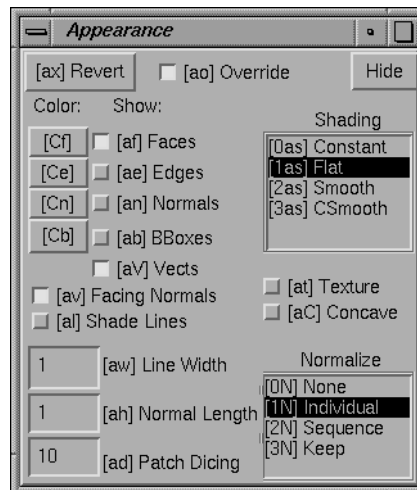


Figure 3: The Appearance Panel

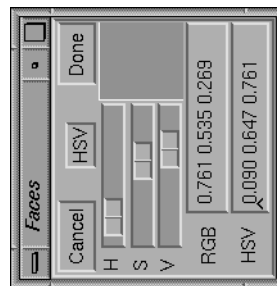


Figure 4: Color Chooser Panel

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

Note the three sliders, *H*, *S*, and *V*, controlling the color's hue, saturation, and value (lightness). Clicking the *HSV* button gives a different set of sliders, one each for red, green, and blue. Numerical values for both RGB and HSV color systems can be seen or edited at the bottom of the panel. The dodecahedron's 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 *Done* button.

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 *Targets* 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 *File* menu in the *Main* panel and choose *Open*. The *Files* panel will appear.

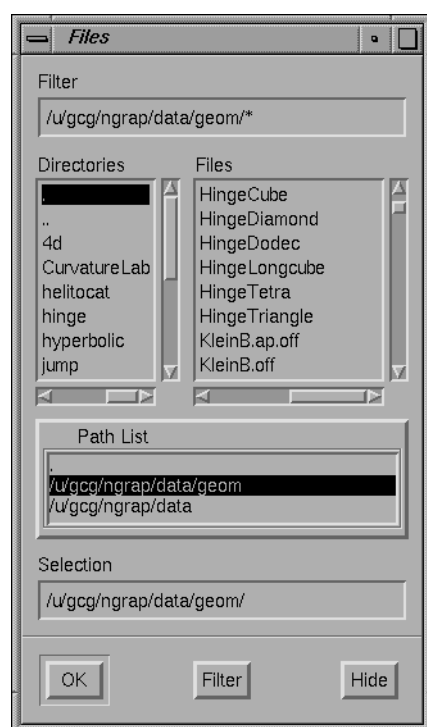


Figure 5: The Files Panel

Below the middle 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, 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.

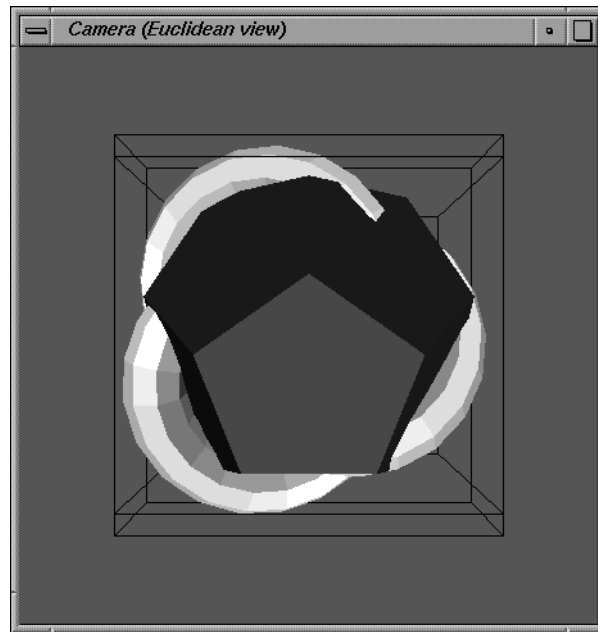


Figure 6: Trefoil and Dodecahedron

For a tutorial on how to create your own objects to load into Geomview, see file 'doc/oogl`tour`' distributed with Geomview. The things in that file will be incorporated into a future version of this manual.

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` `(Enter)` in a shell window (`(Enter)` 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 `(undefined)` [Command Line Options], page `(undefined)`.

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 `(undefined)` [GCL], page `(undefined)`.
- '`-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. Multiple `-c` options are allowed.
- '`-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.
- '`-M objectname`'
 Display (possibly dynamically changing) geometry sent from the programs `geomstuff` or `togeomview`. 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 ...’

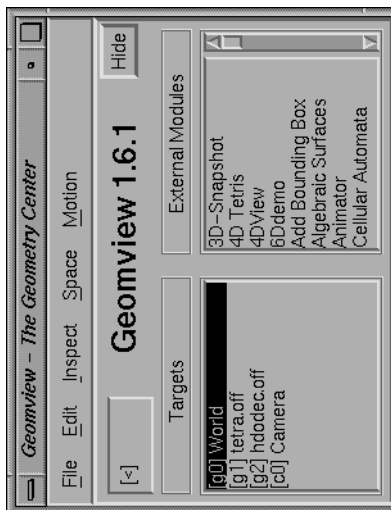
Like -e but allows you to pass arguments to the external module. "-" signals the end of the argument list; the "-" may be omitted if it would be the last argument on the Geomview command line.

‘-run shell-command args ...’

Like -start but takes the pathname of executable of the external module instead of the module’s name. The pathnames of all known module directories are appended to the UNIX search path when invoking *shell-command*.

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.



The Main Panel

Geomview can display an arbitrary number of objects simultaneously. The *Targets* 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 *Targets* 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 *New Camera* entry of the *Main* panel's *File* menu. Geomview treats cameras in much the 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 [\[Cameras\]](#), page [\[undefined\]](#).)

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 *Targets* 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 *Targets* 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 *Targets* 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 [\[Mouse Motions\]](#), page [\[undefined\]](#).

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 [\[Modules\]](#), page [\[undefined\]](#).

The menu bar at the top of the main panel offers menus for common operations.

To create new windows, load new objects, save objects or other information, or quit from geomview, see the *File* menu.

To copy or delete objects, see the *Edit* menu.

You can invoke any panel from the *Inspect* menu.

The *Space* menu lets you choose whether geomview operates in Euclidean, Hyperbolic, or Spherical mode. Euclidean mode is selected by default. For details about using *Hyperbolic* and *Spherical* spaces, See (undefined) [Non-Euclidean Geometry], page (undefined).

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 **Enter** key afterwards.

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

Many keyboard shortcuts can be preceded by a numeric parameter. For example, typing **ae** toggles the state of drawing edges, while **1ae** always enables edge drawing.

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

For a list of all keyboard shortcuts, press the ? key.

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 top of the panel is a standard Motif file browser. Below this is a list of directories on geomview's standard search path; click on one of these to browse files in that directory.

To select a file, double-click on its name in the browser at upper right, or click on its name and press the **Enter** key, or type the file's name into the text box at the bottom of the browser and press **Enter**.

If the selected file contains OOGL geometric data, it will be added to the geomview *Targets* browser. If it contains GCL commands instead, the file will be interpreted. See (undefined) [GCL], page (undefined).

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 directory. This contains a wide variety of sample objects.



The Files Panel

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.

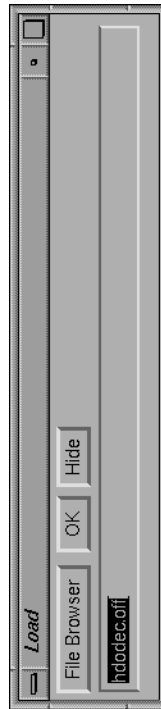
You can change the list of directories shown the *Files* panel’s directory browser by using the `set-load-path` command; see (undefined) [GCL], page (undefined).

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 (or a GCL command surrounded by parentheses). After typing the name of the file to load, type Enter; Geomview will load the file as if you had loaded it with the *Add* button in the *Files* panel. If, after bringing up the small load panel with <, you decide you want to use the larger *Files* panel after all, press the *File Browser* button.

geometry loading commands:

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



The Load Panel

3.5 Using the Mouse to Manipulate Objects

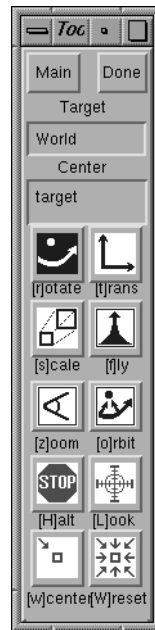
Geomview lets you manipulate objects with the mouse. There are six different mouse motion modes: *Rotate*, *Translate*, *Cam Fly*, *Cam Zoom*, *Geom Scale*, and *Cam Orbit*. The tools panel has a button for each of these modes; to switch modes, click on the corresponding button. You can also select these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

This section describes basic mouse interaction. For details, see [\[Commands\]](#), page [\(undefined\)](#).

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 *Targets* 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 listed in the *Targets* 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 [\[Point of Interest\]](#), page [\(undefined\)](#).



The Tools Panel

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 *Main* panel's *Motion* menu, described below.

Generally, the left mouse button controls motion in the screen plane, while the middle mouse controls motion along or around the forward direction.

Pressing the shift key while dragging with left or middle mouse buttons in most motion modes gives slow-speed motions, useful for fine adjustment.

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 acts as 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 pivot 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. Its 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 *Tools* panel also sports a *Main* button, to invoke the main panel in case it was dismissed or buried, and a *Done* button to close the *Tools* panel.

The *Main* panel's *Motion Style* menu has special controls affecting how mouse motions are interpreted.

[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's view. 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.

3.5.1 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 *Targets* 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 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 a 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

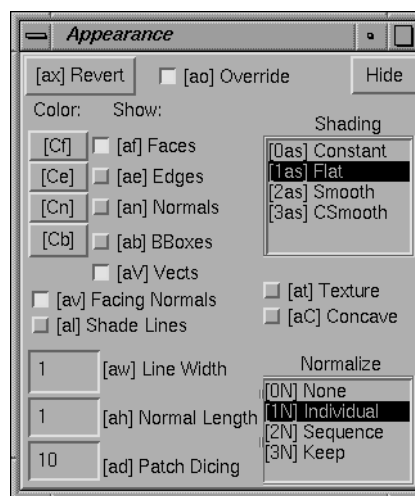
values from the base appearance. This means that by default all objects are drawn using the base appearance.

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.

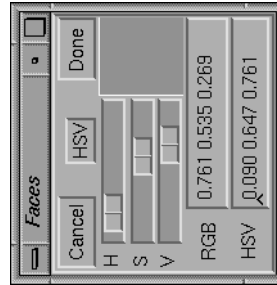


The Appearance Panel

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.

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

- | | |
|----------------|--|
| <i>Faces</i> | This button specifies whether faces are drawn. |
| <i>Edges</i> | This button specifies whether edges are drawn. |
| <i>BBox</i> | This button specifies whether the bounding box is drawn. |
| <i>Vects</i> | This button specifies whether VECT objects are drawn. VECTs are a type of OOG 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. |
| <i>Normals</i> | This button specifies whether surface normal vectors are drawn. |



Color Chooser Panel

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.

This panel offers two sets of sliders: H(ue) S(aturation) V(alue), or R(ed) G(reen) B(lue), each in the range 0 through 1. The square shows the current color, which is given numerically in both HSV and RGB systems in the corresponding text boxes.

In the HSV color system, hue H runs from red at 0, green at .333, blue at .667, and back to red at 1.0. Saturation gives the fraction of white mixed into the color, from 0 for pure gray to 1 for pure color. Value gives the brightness, from 0 for black to 1 for full brightness.

Pressing the *RGb* or *HSV* button at top center switches the sliders to the other color system. You can adjust colors either via the sliders, or by typing in either the RGB or HSV text boxes.

Click *OK* to accept the color that you have chosen, or *Cancel* to retain the previous color setting.

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. Using *Facing Normals* can give strange flickering dark or light shading effects, though, near the horizon of a fairly smooth faceted object. Press this button to use the normals given with the object.

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

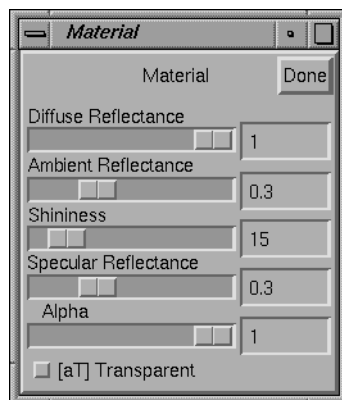
Geomview draws Bezier patches by first converting them to meshes. This parameter specifies the resolution of the mesh: if *Patch Dicing* is n , then an n by n mesh is used to draw each Bezier patch. if *Patch Dicing* is 1, the resolution reverts to a built-in default value.

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.



The Materials Panel

Transparent

This button determines whether transparency is enabled. Geomview itself does not fully support transparency yet and on some machines it does not work at all. (The missing piece is implementation of a depth-sorting algorithm in the rendering engine; not difficult, but just not done yet.) Use RenderMan if you want real transparency: when transparency is enabled, a RenderMan snapshot will contain the alpha information.

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.

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

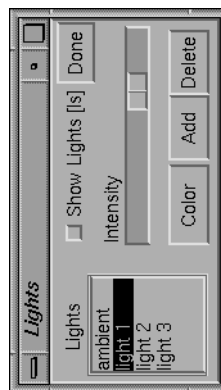
This slider controls the specular reflectance of a surface. This has to do with how directly the surface reflects light rays. Higher values give brighter specular highlights.

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.



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.

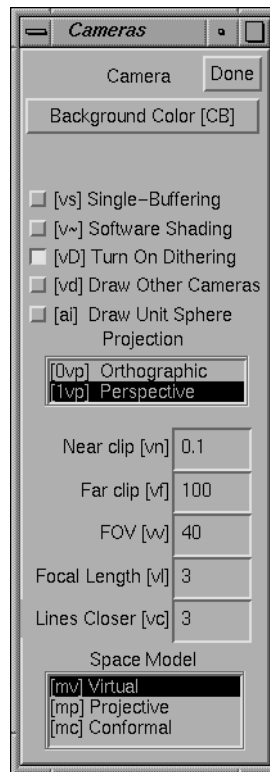
Geomview's *Appearance*, *Materials*, and *Lighting* panels are constructed to allow you to easily do most of the appearance related things that you might want to do. The appearance hierarchy that Geomview supports internally, however, is very complex and there are certain operations that you cannot 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 [\[Appearances\]](#), page [\(undefined\)](#).

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 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.

To create new camera windows, use the **v+** keyboard shortcut, or see the *File* menu on the *Main* panel.



The Cameras Panel

Single-Buffering

Normally, geomview windows are *double-buffered*: geomview draws the next picture into a hidden window, then switches buffers to make it visible all at once. On many systems, the memory for the hidden buffer comes from stealing half the bits in each screen pixel, reducing the color resolution. When single-buffering is enabled, the window flickers as each scene is being drawn, but you may get smoother images with reduced grainy dithering artifacts. Single-buffering is possible if Geomview is compiled with GL or OpenGL, but not with plain-X graphics.

Dither

Many displays offer less than the 24 bits per pixel (8 bits each of red, green, and blue) conventionally needed to show smooth gradations of color. When trying to show a color not accurately available on the display, Geomview normally *dithers*, choosing pixel colors sometimes brighter, sometimes darker than the desired value, so that the average color over an area is a better approximation to the true color than a single pixel could be. Effectively this loses spatial resolution to gain color resolution. This isn't always desirable, though. Turning *Dither* off gives less grainy, but less accurately colored, images.

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. Focal length also affects interpretation of mouse-based translational motions. Speed of forward motion (in translate, fly and orbit modes) is proportional to focal length; and objects lying at the focal distance from the camera translate laterally at the same rate as the mouse cursor. Finally, in N-D projection mode, cameras are displaced back by the focal distance from the 3-D projection of the world origin.

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 [\[Non-Euclidean Geometry\]](#), page [\(undefined\)](#).

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.

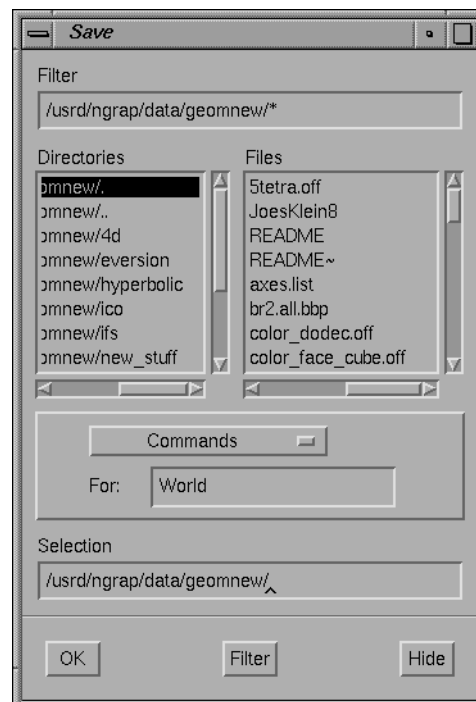
Draw Sphere

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 `(Enter)` 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 writes 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 transformation 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 a window on the screen.

SGI snapshot (camera)

Write an SGI raster file. A bell rings when the snapshot is complete. Only available on SGI systems.

PPM Screen snapshot (camera)

Take a snapshot of the given window and save it as a PPM image. If you specify a string beginning with a vertical bar (|) as the file name, it’s interpreted as a Bourne shell command to which the PPM data should be piped, as in ‘| `pnmtotiff > snap.tiff`’ or ‘| `convert -geometry 50% ppm:- snap.gif`’.

PPM screen snapshots are only available with GL and Open GL, not plain X graphics. The window should be entirely on the screen. Geomview will ensure that no other windows cover it while the snapshot is taken.

PPM software snapshot (camera)

Writes a snapshot of that window’s current view, as a PPM image, to the given file. The file name may be a Bourne shell command preceded by a vertical

bar (|), as with the PPM screen snapshot. The software snapshot, though, is produced by using a built-in software renderer (related to the X-windows renderer). It doesn't matter whether the window is visible or not, and doesn't depend on GL or OpenGL. It also doesn't support some features, such as texture mapping.

Postscript snapshot (camera)

Writes a Postscript snapshot of the camera's view. It's made by breaking up the scene into lines and polygons, sorting by depth, and generating Postscript lines and polygons for each one. Advantages over pixel-based snapshot images: resolution is very high, so edges look sharp even on high-resolution printers, or comparable-resolution images are typically much more compact. Disadvantages: depth-sorting gives good results on some scenes, but can be wildly wrong as a hidden-surface removal algorithm for other scenes. Also, Postscript doesn't offer smoothly interpolated shading, only flat shading for each facet.

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

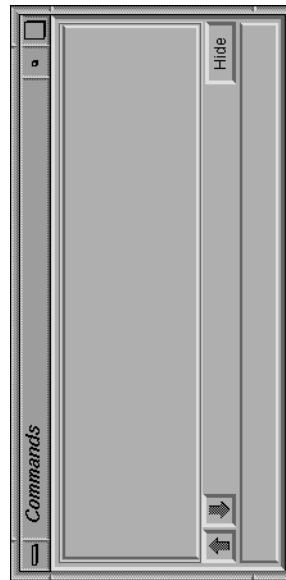
Writes a gcl file containing commands which record the state of all the Geomview panels. Loading this file later will restore the positions of all the panels.

3.9 The Commands Panel

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

[Move this.] 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.



The Commands Panel

- 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".

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. You don't need to press the **Enter** or **SPACE** keys.

Some keyboard shortcuts consist of more than one key. In these cases just type the keys one after the other, with no **Enter** 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 the space bar.

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 **0.5vn**. 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, **g4af** means toggle the face drawing of object *g4*.

Simply typing a selection prefix, like **g4**, doesn't yet select an object; that only happens when a command, like **ae**, follows the prefix. To select an object as the target without doing anything else to it, use the **p** command. So **g3p** selects object *g3*.

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 *Targets* 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	<i>at</i>	texture mapping
	<i>av</i>	eVert normals: always face viewer
	<i>#aw</i>	Line Width (pixels)
	<i>aC</i>	handle concave polygons
	<i>#vc</i>	edges Closer than faces (try 5-100)
Color	<i>Cf</i>	faces
	<i>Ce</i>	edges
	<i>Cn</i>	normals
	<i>Cb</i>	bounding boxes
	<i>CB</i>	background
Motions	<i>r</i>	rotate
	<i>t</i>	translate
	<i>z</i>	zoom FOV
	<i>f</i>	fly
	<i>o</i>	orbit
	<i>s</i>	scale
	<i>w</i>	recenter target
	<i>W</i>	recenter all
	<i>h</i>	halt
	<i>H</i>	halt all
	<i>@</i>	select center of motion (e.g. <i>g 3 @</i>)
	<i>L</i>	Look At object
Viewing	<i>0vp</i>	Orthographic view
	<i>1vp</i>	Perspective view
	<i>vd</i>	Draw other views' cameras
	<i>#vv</i>	field of View
	<i>#vn</i>	near clip distance
	<i>#vf</i>	far clip distance

	<i>v+</i>	add new camera
	<i>vx</i>	cursor on/off
	<i>vb</i>	backfacing poly cull on/off
	<i>#vl</i>	focal length
	<i>v~</i>	Software shading on/off
Panels		
	<i>Pm</i>	Main
	<i>Pa</i>	Appearance
	<i>Pl</i>	Lighting
	<i>Po</i>	Obscure
	<i>Pt</i>	Tools
	<i>Pc</i>	Cameras
	<i>PC</i>	Commands
	<i>Pf</i>	Files
	<i>Ps</i>	Save
	<i>P-</i>	read commands from tty
	<i>PA</i>	Credits ("about")
Lights		
	<i>ls</i>	show lights
	<i>le</i>	edit lights
Space		
	<i>me</i>	Euclidean
	<i>mh</i>	Hyperbolic
	<i>ms</i>	Spherical
Model		
	<i>mv</i>	Virtual
	<i>mp</i>	Projective
	<i>mc</i>	Conformal
Other		
	<i>ON</i>	normalizaton: none
	<i>1N</i>	normalization: each
	<i>2N all</i>	normalization: all

<i>ui</i>	motion: Inertia
<i>uc</i>	motion: Constrain to axis
<i>uo</i>	motion: object's Own coordinates
<i><</i>	
<i>Pf</i>	load geometry/command file
<i>dd</i>	delete target object
<i>></i>	
<i>Ps</i>	save state to file
<i>TV</i>	NTSC mode toggle
<i>p</i>	pick as target object (e.g. <i>g 3 p</i>) 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 [\(undefined\)](#) [Binary format], page [\(undefined\)](#), 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.)

optionally followed by

's t'

'or'

's t u'

(two or three texture-coordinate values).

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

Letters in the object's header key word must appear in a specific order; that's the reverse of the order in which the data is given for each vertex. So a 'CN4OFF' object's vertices contain first the 4-component space position, then the 3-component normal, finally the 4-component color. You can't change the data order by changing the header key word; an 'NCOFF' is just not recognized.

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`, `OFF` and `COMMENT` file formats. Details are given in the individual file format descriptions. See [\[QUAD\]](#), page [\[OFF\]](#), page [\[COMMENT\]](#), and See [\[COMMENT\]](#), page [\[COMMENT\]](#).

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` or `material { *diffuse 1 1 .25 }`, selects "override" status for that attribute.

```

appearance {
  +face           # (Do) draw faces of polygons.  On by default.
  -edge           # (Don't) draw edges of polygons
  +vect           # (Do) draw VECTs.  On by default.
  -transparent    # (Disable) transparency. enabling transparency
                  # does NOT result in a correct Geomview picture,
                  # but alpha values are used in RenderMan snapshots.
  -normal         # (Do) draw surface-normal vectors
  normscale 1     # ... with length 1.0 in object coordinates

  +evert          # do evert polygon normals where needed so as

```

```

# to always face the camera

-texturing      # (Disable) texture mapping
-backcull       # (Don't) discard clockwise-oriented faces
-concave        # (Don't) expect and handle concave polygons
-shadelines     # (Don't) shade lines as if they were lighted cylinders
# These four are only effective where the graphics system
# supports them, namely on GL and Open GL.

-keepcolor      # Normally, when N-D positional coloring is enabled as
# with geomview's (ND-color ...) command, all
# objects' colors are affected. But, objects with the
# "+keepcolor" attribute are immune to N-D coloring.

shading smooth  # or 'shading constant' or 'shading flat' or
# or 'shading csmooth'.
# smooth = Gouraud shading, flat = faceted,
# csmooth = smoothly interpolated but unlighted.

linewidth 1     # lines, points, and edges are 1 pixel wide.

patchdice 10 10 # subdivide Bezier patches this finely in u and v

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.

    backdiffuse .7 .5 0 # back-face color for two-sided surfaces
# If defined, this field determines the diffuse
# color for the back side of a surface.
# It's implemented by the software shader, and
# by hardware shading on GL systems which support

```

```

                                # two-sided lighting, and under Open GL.

alpha    1.0    # opacity; 0 = transparent (invisible), 1 = opaque.
               # Ignored when transparency is disabled.

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 listed
                     # are added to those already in the scene.

                     # Now a collection of sample 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 (well, universe) 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

texture {
    clamp st          # or ‘‘s’’ or ‘‘t’’ or ‘‘none’’
    file lump.tiff    # file supplying texture-map image

```

```

        alphafile mask.pgm.Z      # file supplying transparency-mask image
        apply blend               # or 'modulate' or 'decal'
        transform 1 0 0 0        # surface (s,t,0,1) * tfm -> texture coords
                                0 1 0 0
                                0 0 1 0
                                .5 0 0 1

        background 1 0 0 1      # relevant for 'apply blend'
    }
}                                # 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. Further, appearance controls with "override" status (e.g. `*+face` or `material { *diffuse 1 1 0 }`) win over those without it.

Geomview's appearance controls use the "override" feature so as to be effective even if user-supplied objects contain their own appearance settings. However, if a user-supplied object contains an appearance field with override status set, that property will be immune to Geomview's controls.

4.1.9 Texture Mapping

Some platforms support texture-mapped objects. (On those which don't, attempts to use texture mapping are silently ignored.) A texture is specified as part of an appearance structure, as in See `<undefined>` [Appearances], page `<undefined>`. Briefly, one provides a texture image, which is considered to lie in a square in (s,t) parameter space in the range $0 \leq s \leq 1$, $0 \leq t \leq 1$. Then one provides a geometric primitive, with each vertex tagged with (s,t) texture coordinates. If texturing is enabled, the appropriate portion of the texture image is pasted onto each face of the textured object.

There is (currently) no provision for inheritance of part of a texture structure; if the `texture` keyword is mentioned in an appearance, it supplants any other texture specification.

The appearance attribute `texturing` controls whether textures are used; there's no performance penalty for having texture `{ ... }` fields defined when texturing is off.

The available fields are:

```

clamp none -or- s -or- t -or- st
    Determines the meaning of texture coordinates outside the range 0..1.
    With clamp none, the default, coordinates are interpreted
    modulo 1, so  $(s,t) = (1.25,0)$ ,  $(.25,0)$ , and  $(-.75,0)$  all refer to
    the same point in texture space. With s or t or
    st, either or both of s- or t-coordinates less than 0 or
    greater than 1 are clamped to 1 or 0, respectively.

```

file filename

alphafilename

Specifies image file(s) containing the texture.

The file file's image specifies color or lightness information; the alphafilename if present, specifies a transparency ("alpha") mask; where the mask is zero, pixels are simply not drawn.

Several image file formats are available; the file type must be indicated by the last few characters of the file name:

- .ppm or .ppm.Z or .ppm.gz 24-bit 3-color image in PPM format
- .pgm or .pgm.Z or .pgm.gz 8-bit grayscale image in PGM format
- .sgi or .sgi.Z or .sgi.gz 8-bit, 24-bit, or 32-bit SGI image
- .tiff 8-bit or 24-bit TIFF image
- .gif GIF image

(Though 4-channel TIFF images are possible, and could represent both color and transparency information in one image, that's not supported in geomview at present.)

For this feature to work, some programs must be available in geomview's search path:

- zcat for .Z files
- gzip for .gz files
- tifftopnm for .tiff files
- giftopnm for .gif files

If an alphafilename image is supplied, it must be the same size as the file image.

apply modulate -or- blend -or- decal

Indicates how the texture image is applied to the surface.

Here the "surface color" means the color that surface would have in the absence of texture mapping.

With modulate, the default, the texture color (or lightness, if textured by a gray-scale image) is multiplied by the surface color.

With blend, texture blends between the background color and the surface color. The file parameter must specify a gray-scale image. Where the texture image is 0, the surface color is unaffected; where it's 1, the surface is painted in the color given by background; and color is interpolated for intermediate values.

With decal, the file parameter must specify a 3-color image. If an alphafilename parameter is present, its value interpolates between the surface color (where alpha=0) and the texture color (where alpha=1). Lighting does not affect the texture color in decal mode; effectively the texture is constant-shaded.

```
background  R  G  B  A
```

Specifies a 4-component color, with R, G, B, and A floating-point numbers normally in the range 0..1, used when apply blend is selected.

transform transformation-matrix

Expects a list of 16 numbers, or one of the other ways of representing a transformation (: `handlename` or `< filename`).

The 4x4 transformation matrix is applied to texture coordinates, in the sense of a 4-component row vector $(s, t, 0, 1)$ multiplied on the left of the matrix, to produce new coordinates (s', t') which actually index the texture.

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 [\[Binary format\]](#), page [\[undefined\]](#). 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

```
[U] [C] [N] [Z] [4] [u] [v] [n] MESH # Key word
[Ndim] # Space dimension, present only if nMESH
Nu Nv # Mesh grid 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 [U] [C] [N] [Z] [4] [u] [v] [n] MESH. The optional prefix characters mean:

- 'U' Each vertex includes a 3-component texture space parameter. The first two components are the usual S and T texture parameters for that vertex; the third should be specified as zero.
- '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' 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.
- 'n' Specifies a mesh whose vertices live in a higher dimensional space. The dimension follows the "MESH" keyword. Each vertex then has *Ndim* components.

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 [\[Binary format\]](#), page [\[undefined\]](#). 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 BEZu_vn, indicates that each patch includes four pairs of floating-point texture-space coordinates, one for each corner of the patch.

The C prefix on BEZu_vn 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, N_v) ... (N_u , N_v)

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), (N_u ,0), (0, N_v), and (N_u , N_v) 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:

```
[ST] [C] [N] [4] [n] OFF # Header keyword
[Ndim] # Space dimension of vertices, present only if nOFF
NVertices NFaces NEdges # NEdges not used or checked

x[0] y[0] z[0] # Vertices, possibly with normals,
# colors, and/or texture coordinates, in that order,
# if the prefixes N, C, ST
# are present.
# If 4OFF, each vertex has 4 components,
# including a final homogeneous component.
# If nOFF, each vertex has Ndim components.
# If 4nOFF, each vertex has Ndim+1 components.
...
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
# 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*. These 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 **[ST][C][N][n]OFF BINARY** format is accepted; See <undefined> [Binary format], page <undefined>. 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 texture coordinates or vertex colors or normals if **COFF/NOFF/CNOFF/STCNOFF/etc.** 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 [\(undefined\) \[Binary format\]](#), page [\(undefined\)](#). 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 SKEL Files

SKEL files represent collections of points and polylines, with shared vertices. The conventional suffix for **SKEL** files is `‘.skel’`.

Syntax:

```
[4] [n] SKEL
[NDim]                # Vertex dimension, present only if nSKEL
NVertices  NPolylines

x[0]  y[0]  z[0]      # Vertices
      # (if nSKEL, each vertex has NDim components)
...
x[NVertices-1]  y[NVertices-1]  z[NVertices-1]

                                # Polylines
                                # Nv = # vertices on this polyline (1 = point)
                                # v[0] ... v[Nv-1]: vertex indices
Nv  v[0] v[1] ... v[Nv-1]  [colorspec]
...
                                # colorspec continues past v[Nv-1]
                                # to end-of-line; may be nothing, or 3 or 4 numbers.
                                # nothing: default color
# 3 or 4 floats: RGB[A] values 0..1
```

The syntax resembles that of **OFF** files, with a table of vertices followed by a sequence of polyline descriptions, each referring to vertices by index in the table. Each polyline has an optional color.

For **nSKEL** objects, each vertex has *NDim* components. For **4nSKEL** objects, each vertex has *NDim+1* components; the final component is the homogeneous divisor.

No **BINARY** format is implemented as yet for **SKEL** objects.

4.2.7 SPHERE Files

The conventional suffix for **SPHERE** files is `‘.sph’`.

```
SPHERE
Radius
Xcenter Ycenter Zcenter
```

Sphere objects are drawn using rational Bezier patches, which are diced into meshes; their smoothness, and the time taken to draw them, depends on the setting of the dicing level, 10x10 by default. From Geomview, the Appearance panel, the `<N>ad` keyboard command, or a `dice nu nv` Appearance attribute sets this.

4.2.8 INST Files

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

There is no INST BINARY format.

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 [\[References\]](#), page [\(undefined\)](#), 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). You could use this for, e.g., wrapping an appearance around an externally-supplied object, though a single-membered LIST would do this more efficiently.

See [\[Transformation matrices\]](#), page [\(undefined\)](#), for the matrix format.

Two more INST fields are accepted: `location` and `origin`.

`location [global or camera or ndc or screen or local]`

Normally an INST specifies a position relative to its parent object; the `location` field allows putting an object elsewhere.

- `location global` attaches the object to the global (a.k.a. universe) coordinate system – the same as that in which `geomview`'s World objects, alien geometry, and cameras are placed.
- `location camera` places the object relative to the camera. (Thus if there are multiple views, it may appear in a different spatial position in each view.) The center of the camera's view is along its negative Z axis; positive X is rightward, positive Y upward. Normally the units of camera space are the same as global coordinates. When a camera is reset, the global origin is at (0,0,-3.0).
- `location ndc` places the object relative to the normalized unit cube into which the camera's projection (perspective or orthographic) maps the visible world. X, Y, and Z are each in the range from -1 to +1, with Z = -1 the near and Z = +1 the far clipping plane, and X and Y increasing rightward and upward respectively. Thus something like

```

INST  transform 1 0 0 0 0 1 0 0 0 0 1 0  -.9 -.9 -.999 1
      location ndc
      geom < label.vect

```

pastes `label.vect` onto the lower left corner of each window, and in front of nearly everything else, assuming `label.vect`'s contents lie in the positive quadrant of the X-Y plane. It's tempting to use -1 rather than -.999 as the Z component of the position, but that may put the object just nearer than the near clipping plane and make it (partially) invisible, due to floating-point error.

- `location screen` places the object in screen coordinates. The range of Z is still -1 through +1 as for `ndc` coordinates; X and Y are measured in pixels, and range from (0,0) at the *lower left* corner of the window, increasing rightward and upward.

`location local` is the default; the object is positioned relative to its parent.

```
origin [global or camera or ndc or screen or local] x y z
```

The `origin` field translates the contents of the INST to place the origin at the specified point of the given coordinate system. Unlike `location`, it doesn't change the orientation, only the choice of origin. Both `location` and `origin` can be used together.

So for example

```

{ INST
  location screen
  origin ndc 0 0 -.99
  geom { < xyz.vect }
  transform { 100 0 0 0 0 100 0 0 0 0 -.009 0 0 0 0 1 }
}

```

places `xyz.vect`'s origin in the center of the window, just beyond the near clipping plane. The unit-length X and Y edges are scaled to be just 100 screen units – pixels – long, regardless of the size of the window.

4.2.8.1 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
    ...
    }
}

```

This one resembles the `origin` example in the section above, but makes the X and Y edges be 1/4 the size of the window (1/4, not 1/2, since the range of ndc X and Y coordinates is -1 to +1).

```

{ INST
  location ndc
  geom { < xyz.vect }
  transform { .5 0 0 0 0 .5 0 0 0 0 -.009 0 0 0 -.99 1 }
}

```

4.2.9 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 } }

```

4.2.10 TLIST Files

The conventional suffix for a TLIST file is `.grp` ("group") 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`

TLISTs are used only within the **transforms** clause of an **INST** object. They cause the **INSTs** **geom** object to be instantiated once under each of the transforms in the **TLIST**. The effect is like that of a **LIST** of **INSTs** 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.

4.2.11 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 { oogI-object }
```

is still accepted and effectively translated into

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

4.2.12 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)`.

4.2.13 COMMENT Objects

The **COMMENT** object is a mechanism for encoding arbitrary data within an **OOGL** object. It can be used to keep track of data or pass data back and forth between external modules.

Syntax:

```
COMMENT                                # key word
```

```
name type    # individual name and type specifier
```

```
{ ... }           # arbitrary data
```

The data, which must be enclosed by curly braces, can include anything except unbalanced curly braces. The *type* field can be used to identify data of interest to a particular program through naming conventions.

COMMENT objects are intended to be associated with other objects through inclusion in a **LIST** object. (See [\[LIST\]](#), page [\[undefined\]](#).) The `"#"` OOGL comment syntax does not suffice for data exchange since these comments are stripped when an OOGL object is read in to Geomview. The **COMMENT** object is preserved when loaded into Geomview and is written out intact.

Here is an example associating a WorldWide Web URL with a piece of geometry:

```
{ LIST
  { < Tetrahedron}
  {COMMENT GCHomepage HREF { http://www.geomview.org/ }}
}
```

A binary **COMMENT** format is accepted. Its format is not consistent with the other OOGL binary formats. See [\[Binary format\]](#), page [\[undefined\]](#). The **name** and **type** are followed by

```
N Byte1 Byte2 ... ByteN
```

instead of data enclosed in curly braces.

4.3 Non-geometric objects

The syntax of these objects is given in the form used in See [\[References\]](#), page [\[undefined\]](#), 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. **TLISTs** 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 $(X,Y,Z) = (0,0,\text{focus})$ when reset) and for adjusting field-of-view when switching between perspective and orthographic views.

window aspect ratio

True aspect ratio in the sense $\langle Xsize \rangle / \langle Ysize \rangle$. 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 $\langle \text{far} \rangle / \langle \text{near} \rangle$ 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, halfy-field is related to angular field:

$\text{halfyfield} = \tan(Y_axis_angular_field / 2)$

while for an orthographic one it's simply:

$\text{halfyfield} = Y_axis_linear_field / 2$

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 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 [\[GCL\]](#), page [\(undefined\)](#).

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 aspect 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 *instance* 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 [\[Module Installation\]](#), page [\[Module Installation\]](#).

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 external 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 [\[OOGL File Formats\]](#), page [\(undefined\)](#). 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 [\(undefined\)](#) [\[MESH\]](#), page [\(undefined\)](#).

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 [\(undefined\)](#) [\[GCL\]](#), page [\(undefined\)](#).

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 [\[Forms\]](#), page [\[undefined\]](#).

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
```

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. 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 asynchronous 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 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 occurs. 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
 *
 * 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 External 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 endpoints 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 = LExpr(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 (undefined) [GCL], page (undefined).

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 your 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 equivalent 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 `<undefined>` [OOGL File Formats], page `<undefined>`, and see `<undefined>` [GCL], page `<undefined>`, 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 communication 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 Example 4: Simple Tcl/Tk Module Demonstrating Picking

It's not necessary to write a Geomview module in C. The only requirement of an external module is that it send GCL commands to its standard output and expect responses (if any) on its standard input. An external module can be written in C, perl, tcl/tk, or pretty much anything.

As an example, assuming you have Tcl/Tk version 4.0 or later, here's an external module with a simple GUI which demonstrates interaction with geomview. This manual doesn't discuss the Tcl/Tk language; see the good book on the subject by its originator John Ousterhout, published by Addison-Wesley, titled *Tcl and the Tk Toolkit*.

The '#' on the script's first line causes the system to interpret the script using the Tcl/Tk 'wish' program; you might have to change its first line if that's in some location other than /usr/local/bin/wish4.0. Or, you could define it as a module using

```
(emodule-define "Pick Demo" "wish pickdemo.tcl")
```

in which case 'wish' could be anywhere on the UNIX search path.

```
#!/usr/local/bin/wish4.0
```

```
# We use "fileevent" below to have "readsomething" be called whenever
# data is available from standard input, i.e. when geomview has sent us
# something. It promises to include a trailing newline, so we can use
# "gets" to read the geomview response, then parse its nested parentheses
# into tcl-friendly {} braces.
```

```
proc readsomething {} {
    if {[gets stdin line] < 0} {
        puts stderr "EOF on input, exiting..."
        exit
    }
    regsub -all {\(\} $line "\{" line
    regsub -all {\)} $line "\}" line
    # Strip outermost set of braces
    set stuff [lindex $line 0]
    # Invoke handler for whichever command we got. Could add others here,
    # if we asked geomview for other kinds of data as well.
    switch [lindex $stuff 0] {
        pick      {handlepick $stuff}
        rawevent  {handlekey $stuff}
    }
}
```

```
# Fields of a "pick" response, from geomview manual:
#      (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

# Report when user picked something.
#
proc handlepick {pick} {
    global nameof selvert seledge order
    set obj [lindex $pick 2]
    set xyzw [lindex $pick 3]
    set fv [lindex $pick 6]
    set vi [lindex $pick 8]
    set ei [lindex $pick 9]
    set fi [lindex $pick 10]

    # Report result, converting 4-component homogeneous point into 3-space point.■
    set w [lindex $xyzw 3]
    set x [expr [lindex $xyzw 0]/$w]
    set y [expr [lindex $xyzw 1]/$w]
    set z [expr [lindex $xyzw 2]/$w]
    set s "$x $y $z "
    if {$vi >= 0} {
        set s "$s vertex #$vi"
    }
    if {$ei != {}} {
        set s "$s edge [lindex $ei 0]-[lindex $ei 1]"
    }
    if {$fi != -1} {
        set s "$s face #$fi ([expr [llength $fv]/3]-gon)"
    }
    msg $s
}

# Having asked for notification of these raw events, we report when
# the user pressed these keys in the geomview graphics windows.

proc handlekey {event} {

```

```

    global lastincr
    switch [lindex $event 1] {
        32 {msg "Pressed space bar"}
        8 {msg "Pressed backspace key"}
    }
}

#
# Display a message on the control panel, and on the terminal where geomview
# was started. We use 'puts stderr ...' rather than simply 'puts ...',
# since Geomview interprets anything we send to standard output
# as a GCL command!
#
proc msg {str} {
    global msgtext
    puts stderr $str
    set msgtext $str
    update
}

# Load object from file
proc loadobject {fname} {
    if {$fname != ""} {
        puts "(geometry thing < $fname)"
        # Be sure to flush output to ensure geomview receives this now!
        flush stdout
    }
}

# Build simple "user interface"

# The message area could be a simple label rather than an entry box,
# but we want to be able to use X selection to copy text from it.
# The default mouse bindings do that automatically.

entry .msg -textvariable msgtext -width 45
pack .msg

frame .f

    label .f.l -text "File to load:"
    pack .f.l -side left

    entry .f.ent -textvariable fname
    pack .f.ent -side left -expand true -fill x
    bind .f.ent <Return> { loadobject $fname }

```

```

pack .f

# End UI definition.

# Call "readsomething" when data arrives from geomview.

fileevent stdin readable {readsomething}

# Geomview initialization

puts {
    (interest (pick primitive))
    (interest (rawevent 32)) # Be notified when user presses space
    (interest (rawevent 8)) # or backspace keys.
    (geometry thing < hdecode.off)
    (normalization world none)
}
# Flush to ensure geomview receives this.
flush stdout

wm title . {Sample external module}

msg "Click right mouse in graphics window"

```

6.7 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.

6.7.1 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 [\[Customization\]](#),

page [\[GCL\]](#). You can also execute **emodule-define** commands from the *Commands* panel 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 [\[GCL\]](#), page [\[GCL\]](#).

6.7.2 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 Bourne 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/<CPU>’` directory, where `‘<CPU>’` is your system type.

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. To execute a gcl command interactively, you can bring up the *Commands* panel which lets you type in a command; Geomview executes the command when you hit the `Enter` 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. 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 `(un-
defined)` [OOGL File Formats], page `(un-
defined)`. (Or equivalently, 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 window specification.

7.2 Gcl Reference Guide

`!` is a synonym for `shell`

`(< 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.

`(? [command])`

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

`(?? command) command may include * wildcards`

Prints more info than `(? command)`. `??` is a synonym for `morehelp`.

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

`(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.

`(background-image CAM-ID [FILENAME])`

Use the given image as the background of camera `CAM-ID` (which must be a real camera, not `default` or `allcams`). Centers the image on the window area. Works only with GL and OpenGL graphics. Use `""` for filename to remove background. With no filename argument, returns name of that window's current background image, or `""`. Any file type acceptable as a texture is allowed, e.g. `.ppm.gz`, `.sgi`, etc.

(bbox-color GEOM-ID R G B)

Set the bounding-box color of GEOM-ID; R G B are numbers 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 object } [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 $-1 \leq \{X,Y\} \leq +1$. 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 target-geom. If name is not specified, it is assumed to be the same as the name of ID.

(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.

(dice GEOM-ID N)

Dice any Bezier patches within GEOM-ID into NxN meshes; default 10. See also the appearance attribute **dice**, which makes this command obsolete.

(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.

(dither CAM-ID {on|off|toggle})

Turn dithering on or off in that camera.

(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 the given external-module name is known, returns the name of the program invoked when it's run as a quoted string; otherwise returns nil. (**echo (emodule-defined name)**) 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 arguments).

(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 geomview 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 application 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-keys {on|off})

Turn keyboard events on or off to enable/disable keyboard shortcuts.

(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).

(event-pick {on|off})

Turn picking on or off.

(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 the `read ... define` performs assignment as soon as the text is read.

`(help [command])`

Command may include `*` as wildcards; see also `??` One-line command help; lists names only if multiple commands match.

`(hmodel CAMID {virtual|projective|conformal})`

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

`(hsphere-draw CAMID [yes|no])`

Say whether to draw a unit sphere: the sphere at infinity in hyperbolic space, and a reference sphere in Euclidean and spherical spaces. 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 inhibits 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.

`(input-translator "#prefix_string" "Bourne-shell-command")`

Defines an external translation program for special input types. When asked to read a file which begins with the specified string, geomview invokes that program with standard input coming from the given file. The program is expected to emit OOGL geometric data to its standard output. In this implementation, only prefixes beginning with `#` are recognized. Useful as in

`(input-translator "#VRML" "vrm12oogl")`

`(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 suppresses 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` (`geomview` 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 command file executes those commands.

`(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 [objectID] [cameraID])`

Rotates the named camera to point toward the center of the bounding box of the named object (or the origin in hyperbolic or spherical 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 [objectID] [cameraID] {center | origin}) [(look-encompass [objectID] [cameraID])])`. If `objectID` is not specified, it is assumed to be `World`. If `cameraID` is not specified, it is assumed to be `targetcam`.

`(look-encompass [objectID] [cameraID])`

Moves `cameraID` backwards or forwards until its field of view surrounds `objectID`. This routine works only in Euclidean space. If `objectID` is not specified,

it is assumed to be the world. If cameraID 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 [objectID] [cameraID])

Translates and rotates the camera so that it is looking in the -z direction (in objectID's coordinate system) at the center of objectID's bounding box (or the origin of the coordinate system in non-Euclidean 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 objectID and cameraID are parallel.

(look-toward [objectID] [cameraID] [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 objectID 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 appearance). Lighting is typically included in the base appearance.

(morehelp command)

command may include * wildcards. Prints more info than (help command).

(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.

(ND-axes CAMID [CLUSTERNAME [Xindex Yindex Zindex]])

In our model for N-D viewing (enabled by (dimension)), objects in N-space are viewed by N-dimensional *camera clusters*. Each real camera window belongs to

some cluster, and shows & manipulates a 3-D axis-aligned projected subspace of the N-space seen by its cluster. Moving one camera in a cluster affects its siblings.

The ND-axes command configures all this. It specifies a camera's cluster membership, and the set of N-space axes which become the 3-D camera's X, Y, and Z axes. Axes are specified by their indices, from 0 to N-1 for an N-dimensional space. Cluster CLUSTERNAME is implicitly created if not previously known. To read a camera's configuration, use `(echo (ND-axes CAMID))`.

`(ND-color CAMID`

`[(([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 ...) ...)])` Specifies a function, applied to each N-D vertex, which determines the colors of N-dimensional objects as shown in camera CAMID. Each coloring function is defined by a vector (in ID's coordinate system) `[x0 x1 ... xn]` and by a sequence of value (v)/color(r g b a) tuples, ordered by increasing v. The inner product $v = P \cdot [x]$ is linearly interpolated in this table to give a color. If ID is omitted, the (xi) vector is assumed in universe coordinates. The ND-color command specifies a list of such functions; each vertex is colored by their sum (so e.g. green intensity could indicate projection along one axis while red indicated another. An empty list, as in `(ND-color CAMID ())`, suppresses coloring. With no second argument, `(ND-color CAMID)` returns that camera's color-function list. Even when coloring is enabled, objects tagged with the `keepcolor` appearance attribute are shown in their natural colors.

`(ND-xform OBJID [ntransform { idim odim ... }])`

Sets or returns the N-D transform of the given object. In dimension N, this is an $(N+1) \times (N+1)$ matrix. Note that all cameras in a camera-cluster have the same N-D transform.

`(ND-xform-get ID [from-ID])`

Returns the N-D transform of the given object in the coordinate system of from-ID (default `universe`), in the sense `<point-in-ID-coords> * Transform = <point-in-from-ID-coords>`

`(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))

(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, $\sqrt{(x_{\max}-x_{\min})^2 + (y_{\max}-y_{\min})^2 + (z_{\max}-z_{\min})^2}$, 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.

- (pick-invisible [yes|no])
Selects whether picks should be sensitive to objects whose appearance makes them invisible; default yes. With no arguments, returns current status.
- (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 CAMID X Y)
Process a pick event in camera CAMID 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 registry table

(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 browser; 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 [FACTORY FACTORZ])

Scale GEOM-ID, multiplying its size by FACTOR. The factors should be positive numbers. If FACTORY and FACTORZ are present and non-zero, the object is scaled by FACTOR in x, by FACTORY in y, and by FACTORZ 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 browser from the new path setting. The special directory name + is replaced by the existing path, so e.g. (set-emodule-path (mydir +)) prepends mydir to the path.

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

Sets search path for command, geometry, etc. files. The PATH_i are strings giving the pathnames of directories to be searched. The special directory name + is replaced by the existing path, so e.g. (set-load-path (mydir +)) prepends mydir to the path.

(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 $\text{dist} + \text{scaleof}(\text{frame}) * \text{motionscale}$ 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. A synonym is !.

(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 [FORMAT [XSIZE [YSIZE]]])

Save a snapshot of CAM-ID in the FILENAME (a string). The FORMAT argument is optional; it may be **ppmscreen**, **sgi**, **ps**, or **ppm**. A **ppmscreen** snapshot is created by reading the image directly from the given window; the window is popped above other windows and redrawn first, then its contents are written

as a PPM format image. With **ps**, dumps a Postscript picture representing the view from that window; hidden-surface removal might be incorrect. With **ppm**, dumps a PPM-format image produced by geomview's internal software renderer; this may be of arbitrary size. If the **FILENAME** argument begins with the vertical bar **|**, it's interpreted as a **/bin/sh** command to which the PPM or PS data should be piped. Optional **XSIZE** and **YSIZE** values are relevant only for **ppm** format, and render to a window of that size (or scaled to that size, with aspect fixed, if only **XSIZE** is given)

(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 [no|horizontal|vertical|colored] [gapsize])

Configure CAM-ID as a stereo window. **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 stereeyes 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 usage:

(time-interests .1 (set-clock %g) (sleep-until %g)) or

(time-interests .1 (set-clock %g) "(sleep-until %g) (progn (set-clock %g) " ")")

or

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

(transform objectID centerID frameID

[rotate|translate|translate-scaled|scale] x y z [dt] [smooth])

Apply a motion (rotation, translation, scaling) to object **objectID**; that is, construct and concatenate a transformation matrix with **objectID**'s transform. The 3 IDs 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 **centerID** is the same as **objectID** then it will spin around its own axes; otherwise the moving object will orbit the center object. Normally **frameID**, 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. The optional **dt** field allows a simple form of animation; if present, the object moves by just that amount during approximately *dt* seconds, then stops. If present and followed by the **smooth** keyword, the motion is animated with a $3t^2-2t^3$ function, so as to start and stop smoothly. If absent, the motion is applied immediately.

(transform-incr objectID centerID frameID

[rotate|translate|translate-scaled|scale] x y z [dt])

Apply continuing motion: construct a transformation matrix and concatenate it with the current transform of objectID every refresh (sets objectID'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 6.28318 0 0 10.0) rotates the World about its X axis at 1 turn (2pi radians) per 10 seconds.

(transform-set objectID centerID frameID

[rotate|translate|translate-scaled|scale] 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 equivalent **emodule-define** instead.

ui-emotion-run is an obsolete command.

Use its new equivalent **emodule_start** instead.

(ui-freeze [on|off])

Toggle updating user interface panels. Off by default.

(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 **geomview** 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_in_seconds])

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|otherID])**

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 World-transform **other ID**: the object transformed to otherID'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-comments FILENAME GEOMID PICKPATH)

write OOGL COMMENT objects in the GEOMID hierarchy at the level of the pick path to FILENAME. Specifically, COMMENTS at level (a b c ... f g) will match pick paths of the form (a b c ... f *) where * includes any value of g, and also any values of possible further indices h,i,j, etc. The pick path (returned in the **pick** command) is a list of integer counters specifying a subpart of a hierarchical OOGL object. Descent into a complex object (LIST or INST) adds a new integer to the path. Traversal of simple objects increments the

counter at the current level. Individual COMMENTS are enclosed by curly braces, and the entire string of zero, one, or more COMMENTS (written in the order in which they are encountered during hierarchy traversal) is enclosed by parentheses.

Note that arbitrary data can only be passed through the OOGL libraries as full-fledged OOGL COMMENT objects, which can be attached to other OOGL objects via the LIST type as described above. Ordinary comments in OOGL files (i.e. everything after '#' on a line) are ignored at when the file is loaded and cannot be returned.

(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 in 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 Euclidean 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. 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 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 geomview 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 `[OOGL.m]`, page `[OOGL.m]`.

```
In[5] := Graphics3D[ {RGBColor[1,0,0], Line[{ {2,2,2},{1,1,1} }]} ]
```

```
Out[5] := -Graphics3D-
```

```
In[6] := Geomview["myline", %5]
```

This adds 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. 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.ri`'b. 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 you want to use Mathematica on a computer that is not networked with your Geomview computer, you can write out *chunk* files in Mathematica which you transfer to the Geomview computer and then translate to OOGL format for displaying in Geomview.

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, `Geomview` needs to know the type of that machine (the setting of the CPU variable that corresponds to the machine; See [\(undefined\) \[Source Code Installation\]](#), page [\(undefined\)](#)). 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" or "x11".

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 computer that `Geomview` isn't installed on. `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/<CPU>' 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 a system with `Geomview` 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).

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 <undefined> [Installation], page <undefined>, 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 tailor 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 to any of the procedures in See <undefined> [Installation], page <undefined>. 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 Mathematica 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 `'oogl2ri'`. 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 `'oogl2ri'` 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 `'oogl2ri'` in the `'$GEOMROOT/bin/<CPU>'`.

3. The `'geomview'` shell script, which `'OOGL.m'` uses to invoke Geomview, needs to be able to find the geomview executable file (called `'gvx'`). 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 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 one of the binary distribution. The binary installation is much easier and quicker than compiling and installing the source code.

10.1 Installing the Unix Binary Distribution

If you have just obtained a copy of the binary distribution for a Unix system (Linux, SGI, Solaris, HP, etc), 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 source-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/<CPU>' and '\$GEOMROOT/man' into the directories you specified for executables and man pages, respectively. Once you have done the installation you can cut down on 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/<MACHTYPE>':

(where '<MACHTYPE>' is the type of system you are on, e.g. 'linux', 'sgi', 'hpux', etc). 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 Unix 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/<MACHTYPE>'` 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 `$Path` variable in a Mathematica session. The installation process will install some packages there which allow you to use Geomview to display Mathematica graphics. These packages are distributed in the `'$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 `<undefined>` [Package Installation], page `<undefined>`.

10.2 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 concerned with *using* Geomview it is better to get the binary distribution. It takes anywhere from a few minutes to an hour or more 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 multiple systems, you can do them all 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 edit the file `'$GEOMROOT/makefiles/Makedefs.global'` to set the `CPU` variable to one of the values `'linux'`, `'FreeBSD'`, `'sgi'`, `'hpux'`, `'hpux-gcc'`, `'solaris'`, `'sun4os4'` (for Suns with SunOS 4, not Solaris), `'rs6000'`, or `'alpha'`. If you're using a type of system not in this list, make up a new value for `CPU`, and write a `'mk.<CPU>'` file for it patterned after the other `'mk.*'` in the `'makefiles'` subdirectory.

2. If you will be compiling on more than one type of computer you can set a shell environment variable named `CPU` to one of the values above and the Makefiles will inherit the value from the environment.

Note that many of the Makefiles refer to a variable called `MACHTYPE`; this variable tells which type of graphics system to compile Geomview for. The `'mk.<CPU>'` files set this variable for you; in most cases its value is `'x11'`, which specifies that Geomview should be compiled for X windows.

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` commands in any subdirectory in the tree to recompile and/or install a part of Geomview or a module.

If you want to modify the compiler flags used during compilation, edit the file `'$GEOMROOT/makefiles/Makedefs.global'`; the `COPTS` variable specifies the flags passed to the C compiler (`cc`).

Getting Technical Support for Geomview

There are several ways to get support for Geomview.

1. Visit the Geomview web site, www.geomview.org. It contains the latest documentation, news about development, and FAQ (Frequently Asked Questions) list.
2. Send email to the geomview-users@geomview.org mailing list. This is a mailing list for discussing any issues related to using Geomview; to be added to the list send a note to geomview-users-request@geomview.org.
3. Contract with Geometry Technologies for support. Geometry Technologies is a contract support and programming company that emerged from the Geometry Center, where Geomview was written. For more information, send email to info@geomtech.com, or visit the Geometry Technologies web site at www.geomtech.com.

Function Index

(Index is nonexistent)

Table of Contents

Introduction to Geomview	2
Distribution	3
GNU GENERAL PUBLIC LICENSE	4
Preamble	4
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION	5
How to Apply These Terms to Your New Programs	9
History of Geomview's Development	11
Authors	11
How to Pronounce 'Geomview'	12
Let Us Hear From You	13
1 Overview	15
2 Tutorial	16
3 Interaction	22
3.1 Starting Geomview	22
3.2 Command Line Options	22
3.3 Basic Interaction: The Main Panel	23
3.4 Loading Objects Into Geomview	25
3.5 Using the Mouse to Manipulate Objects	27
3.5.1 Selecting a Point of Interest	30
3.6 Changing the Way Things Look	31
3.6.1 The Appearance Panel	32
3.6.2 The Materials Panel	34
3.6.3 The Lighting Panel	35
3.7 Cameras	36
3.8 Saving your work	39
3.9 The Commands Panel	41
3.10 Keyboard Shortcuts	42

4	OOGL File Formats	47
4.1	Conventions	47
4.1.1	Syntax Common to All OOGL File Formats	47
4.1.2	File Names	47
4.1.3	Vertices	47
4.1.4	Surface normal directions	48
4.1.5	Transformation matrices	48
4.1.6	Binary format	49
4.1.7	Embedded objects and external-object references	49
4.1.8	Appearances	50
4.1.9	Texture Mapping	53
4.2	Object File Formats	55
4.2.1	QUAD: collection of quadrilaterals	55
4.2.2	MESH: rectangularly-connected mesh	55
4.2.3	Bezier Surfaces	56
4.2.4	OFF Files	58
4.2.5	VECT Files	60
4.2.6	SKEL Files	61
4.2.7	SPHERE Files	61
4.2.8	INST Files	62
4.2.8.1	INST Examples	63
4.2.9	LIST Files	64
4.2.10	TLIST Files	64
4.2.11	GROUP Files	65
4.2.12	DISCGRP Files	65
4.2.13	COMMENT Objects	65
4.3	Non-geometric objects	66
4.3.1	Transform Objects	66
4.3.2	cameras	67
4.3.3	window	69
5	Customization: ‘.geomview’ files	71
6	External Modules	72
6.1	How External Modules Interface with Geomview	72
6.2	Example 1: Simple External Module	72
6.3	Example 2: Simple External Module with FORMS Control Panel	76
6.4	The FORMS Library	80
6.5	Example 3: External Module with Bi-Directional Communication	80
6.6	Example 4: Simple Tcl/Tk Module Demonstrating Picking	89
6.7	Module Installation	92
6.7.1	Private Module Installation	92
6.7.2	System Module Installation	93

7	gcl: the Geomview Command Language	94
7.1	Conventions Used In Describing Argument Types	94
7.2	Gcl Reference Guide	96
8	Non-Euclidean Geometry	112
9	Mathematica Graphics in Geomview or RenderMan	113
9.1	Using Mathematica to generate OOGL files	113
9.2	Using Geomview as Mathematica's Default 3D Display . . .	114
9.3	Using Mathematica to generate RenderMan files	115
9.4	Using Geomview and Mathematica on Different Computers	116
9.4.1	Using a Networked Geomview Host	116
9.4.2	Transporting Mathematica Files to Geomview by Hand	117
9.5	Details of the Mathematica->Geomview Package	118
9.6	Installing the Mathematica Packages	118
10	Installation	120
10.1	Installing the Unix Binary Distribution	120
10.1.1	Details of the Unix Binary Installation	121
10.2	Compiling and Installing the Source Code Distribution . .	121
	Getting Technical Support for Geomview	123
	Function Index	124