GCI: A network server for interactive 3D graphics

Georg Tuparev, Gerrit Vriend and Chris Sander

European Molecular Biology Laboratory, Meyerhofstrasse 1, D-6900 Heidelberg, Germany

The Graphics Command Interpreter (GCI) is an independent server module that can be interfaced to any program that needs interactive three-dimensional (3D) graphics capabilities. The principal advantage of GCI is its simplicity. Only a limited set of powerful features have been implemented, including object management, global and local transformations, rotation, translation, clipping, scaling, viewport operations, window management, menu handling and picking.

GCI and the master (client) program it serves run concurrently, communicating over a local or remote TCP/IP network. GCI sets up socket communication and provides a 3D graphics window and a terminal emulator for the master program. Communication between the two programs is via ASCII strings over standard I/O channels. The implied language for messages is very simple. GCI interprets messages from the master program and implements them as changes of graphical objects or as text messages to the user. GCI provides the user with facilities to manipulate the view of the displayed 3D objects interactively, independently of the master program, and to communicate mouse-controlled selection of menu items or 3D points as well as keyboard strings to the master program.

The program is written in C and initially implemented using the Silicon Graphics GL graphics library. As the need to link special libraries to the master program is completely avoided, GCI can very easily be interfaced to existing programs written in any language and running on any operating system capable of TCP/IP communication. The program is freely available.

Key words: molecular graphics, message passing, 3-D graphics server

GRAPHICS SERVER

The rapid development of computer graphics hardware, combined with the lack of universally accepted graphics standards or libraries, creates an almost continuous need for program conversions. Languages like FORTRAN or C are standardized enough to make the conversion of a well-written program straightforward. The conversion of graphics programs to new hardware, however, tends to be a major endeavor. If we compare, for example, the source codes for the FRODO version[1] that runs on an Evans and Sutherland PS300 system, with the version that runs on a Silicon Graphics IRIS, we can hardly believe that these two versions were once the same program.

A classical way of easing the job of converting graphics programs to new hardware is to decouple the graphics modules from the rest of the program. This is the concept of a device driver (for example, two-dimensional plotting) or server (for example, Xwindows). When new hardware or system software arrives, only the driver or server has to be converted, not the client or master programs that use it. Classical examples of graphics drivers are HPGL from Hewlett Packard, PLOT10 from Tektronix and the hierarchical data structure language for the PS300 machines from Evans and Sutherland.

As graphics servers for interactive three-dimensional (3D) graphics are virtually nonexistent, we set out to design one with the following characteristics.

(1) The server should be flexible and general enough to work with a large variety of master programs.
(2) Whenever possible, the complexity of program response to interactive graphics (event handling) should be dealt with by the server.
(3) The server should be small enough to facilitate future conversion.
(4) The server should be able to communicate with the master program through byte streams over a standard network protocol.
(5) The implied language of commands and data should be very simple.

The principal design goals were simplicity, ease of developing interfaces to existing programs and ease of conversion to new hardware. Although we developed this program in a molecular modeling environment, it is general enough to be used in any environment where interactive manipulation of 3D line and dot drawings is needed.

MESSAGE HANDLING

One of the problems encountered in the fields of molecular modeling (drug design, protein structure analysis, etc.) is the multitude of programs and file formats used for very

Address reprint requests to Dr. Sander at the European Molecular Biology Laboratory (EMBL), Meyerhofstrasse 1, D-6900 Heidelberg, Germany.
Received 22 August 1991; revised 29 October 1991; accepted 5 November 1991

similar information. Typically, significant effort is wasted by different individuals on programming format conversions and data I/O in different programs. There are two solutions to this problem. The first is to produce one large integrated program with internally consistent data structures that can do "everything." The second is to facilitate the communication between many programs by object and message standards. We are working along both lines. The program[2] WHAT IF is a multifunction molecular modeling and drug design package written in FORTRAN that is constantly being extended with new features as research progress and user needs dictate. However, in order not to have to rewrite very useful existing programs we are also designing a message handler that allows for flexible, user- and programmer-friendly communication between many programs. A general molecular graphics program has a very prominent place in such a system. The Graphics Command Interpreter server (GCI) will be presented here. Other programs in the system will be discussed elsewhere. Figure 1 shows the central role of GCI in this system.

MINIMAL INTERFACE

The main aim of the GCI project was to provide 3D graphics capabilities to program developers in a painless and simple fashion. To achieve this, GCI was designed as a separate program. Two programs have to run in order to have a working configuration: GCI performs the actual graphics operations like displaying, rotating, clipping and picking, and the client, the so-called master program, generates objects to be displayed, rotated, clipped and picked. Communication between these programs takes place via read and write operations to standard input and output channels via sockets. The principal advantage of this approach is that the code for 3D graphics display and user control over how to view the 3D "world" is independent of the particular master program. The graphics server can therefore be developed and debugged largely independently and reused without modifications in many different contexts. The principal disadvantage is somewhat slower data transfer between the master and the server.

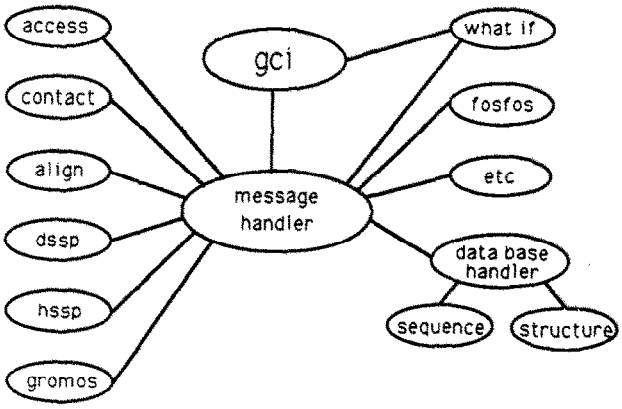

Figure 1. Function of GCI in the message handling system. The names in ellipses refer to programs that perform different manipulations on protein sequences and structures for which GCI provides 3D graphics support on request.

GRAPHICAL OBJECTS

GCI does not know anything about molecules, residues or atoms. Its graphical primitives are dots or lines. Any molecule to be displayed has to be converted into a display list by the master program. For example, the connectivity in a protein or DNA molecule has to be calculated by the master program, and a series of lines and dots representing the molecule is sent to GCI. Sphere primitives or polygons are not implemented. We have deliberately kept GCI very simple, and have used only very few GL-specific features. This was done to make any future port to another graphics language as simple as possible.

Figure 2 shows the hierarchical data structure of GCI: A world is subdivided into objects, which in turn hold items. Items can consist of lines, dots, labels (visible) and pick points (hidden). GCI administers the objects and items in this structure for the master program. Active items are identified by two pointers. The *data pointer* indicates the current item to which all lines, dots, labels and pick points sent to the graphics server will be appended. The *motion pointer* indicates the world, object or item, which can be rotated or translated by mouse action. User-controlled rotation, etc. is completely independent of the master program, but information on the current rotation–translation state of any object is communicated to the master program upon request.

COMMANDS

Table 1 lists selected commands presently implemented in GCI: flow control, window management, graphics transformations, object and item management, picking and menus and information transfer. For example, one can create or delete objects or their children, the items. Items are updated (appended to) by commands to send vectors, dots, labels or pick points. For simplicity, there is no option to delete individual dots or lines. One can, of course, achieve this by deleting an item entirely and regenerating it without the attributes one wants to delete. The last two sets of commands in Table 1 are needed for interprocess communication. For example the local transformations corresponding to the user view of objects can be passed from the 3D graphics device back to the master program.

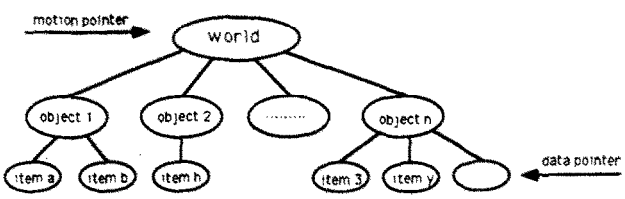

Figure 2. Sample layout of the GCI data tree. The number of objects and items is limited only by the hardware. The motion pointer *indicates the item, object or world currently manipulated by mouse action. The* data pointer *indicates the item to which new data currently can be appended. These pointers can point to any ellipse in this figure.*

Table 1.

Messages from the master program to GCI

Flow control

@Start <welcome_text>
First command in the master program. All lines sent to GCI before the @Start command will be ignored.
@ExecFile <file_name>
Executes a script file of GCI commands.
@Stop
After receiving the @Stop command both processes (master program and GCI) will be terminated. Before termination, all necessary clean-ups will be performed. Note that in the current implementation execution of the master program is initiated by GCI.
@MouseControl
Switches the standard input channel of the master program to expect mouse-driven commands.
@KeyboardControl
Switches the standard input channel of the master program to expect keyboard-driven commands.

Window management

@GraphInit
Initialization of the graphics window. Removes all graphical objects.
@TWTitle <string>
Changes the title of the text window.
@GWTitle <string>
Changes the title of the graphics window.
@<text>
Shows <text> in the text window and waits for a user response. After <RET> the entered string will be sent to the master program.
@@<text>
Shows <text> in a debug window.
@Comment <text>
Shows the <text> in the "info" line of the graphics window.
@UnComment
Removes the comment.
@Stereo
Switches the graphics window to stereo mode.
@NoStereo
Switches the graphics window to mono mode.

Color

@EnvColor <env_numb> <red> <green> <blue>
Sets the color of an environment, e.g., background, frames and characters.
@MapColor <col_numb> <red> <green> <blue>
Defines an entry in the color map for use with SetD and SetV commands.

Graphics transformations

Unless specified otherwise, graphics transformations act on the entity pointed to by the motion pointer, i.e., an item, an object or the world.
@Center <x> <y> <z>
Sets the center of the graphics viewport.
@Scale <scale>
Applies a scale factor to the world.
@Rotate <x_angle> <y_angle> <z_angle>
Rotates the world, object or item.
@Translate <x> <y> <z>
Translates the world, object or item.
@Viewport <x1> <x2> <y1> <y2> <z1> <z2>
Sets the graphics viewport, a rectangular 3D box.
@ResetRot
Resets the rotations of the world, object or item.
@ResetTrans
Resets the translations of the world, object or item.

Items and objects

@NewItem <obj_numb> <item_name>
Sets the data pointer at a new item in the graphics tree.
@DelItem <obj_numb> <item_name>
Removes the specified item from the graphics tree.
@Touch <obj_numb> <item_name>
Sets the motion pointer at the indicated position in the graphics tree.

The following commands act on the item indicated by the data pointer.
@SetD <x> <y> <z> <color_numb>
Adds a dot to the item.
@SetV <x1> <y1> <z1> <x2> <y2> <z2> <color_numb>
Adds a vector to the item.
@InitLabelList
Initializes the label list of all items.
@AddLabel <label_numb> <x> <y> <z> <"string">
Adds an entry to the label list.
@DelLabel <label_numb>
Removes an entry from the label list.

Picking and menu

@InitPickList
Initializes the pick list (list of 3-D points that can be picked by mouse action).
@SetPick <x> <y> <z> <pick_id>
Adds an entry to the pick list.
@InitMenu
Initializes the user-defined menu in the graphics window.
@Menu <menu_numb> <text>
Adds a user-defined menu entry.
@TagMenu <menu_numb>
Visibly tags a menu entry.
@UnTagMenu <menu_numb>
Removes a tag from a menu entry.

Information requests

@GetCenter
Requests the current center of the graphics viewport.
@GetScale

(Continued)

Table 1. ***(Continued)***

Requests the current scale factor.
@GetRot
Requests the absolute rotation of the currently moving item or object.
@GetTrans
Requests the absolute translation of the currently moving item or object.

Messages from GCI to the Master Program
Messages in MouseControl mode

@PickId <pick_id>
Returns the identifier of a point picked by a mouse click.
@MenuItem <menu_numb>
Returns the identifier of a menu entry picked by a mouse click.

Messages in response to information requests

@RetCenter <x> <y> <z>
Returns the center of the graphics viewport.
@RetScale <scale>
Returns the current scale factor.
@RetRot <x_angle> <y_angle> <z_angle>
Returns the absolute rotation of the item or object indicated by the motion pointer.
@RetTrans <x> <y> <z>
Returns the absolute translation of the item or object indicated by the motion pointer.

INTERPROCESS AND USER COMMUNICATION

All messages between the master and server processes are in the form of strings sent and received via standard I/O. There are two types of messages: strings to and from the user via a text window and all other strings, such as graphical commands. For the former, GCI merely emulates a text window for communication between the master program and the user (Figure 3). Every line of text sent to standard I/O by the master program will be sent to this text window, and any text typed by the user in this text window will be passed on to the master program. Strings starting with the character @ do not involve the text window and are commands, queries and replies. For example, graphical commands originating from the master program are interpreted by GCI as lines, dots and the like, and have their effect in the graphics window seen by the user. In response to informational queries from the master program GCI replies by sending back the requested information, e.g., transformation matrices indicating the location and orientation of objects. User interaction with the graphics window via the mouse also can result in strings being sent back to the master program, e.g., the location of an atom or the choice of a menu entry (Figure 3).

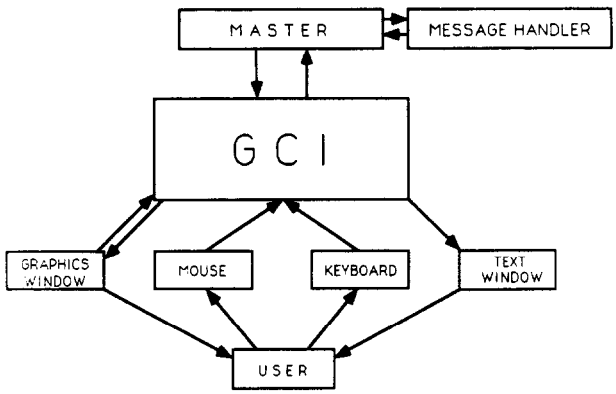

Figure 3. Communication between the master program, the GCI server and the user. Arrows indicate the direction of information flow. Communication beween the master program (the client) and GCI (the server) is via standard I/O channels. In this way, network communication can be set up by the GCI server, avoiding the need to link special network communication libraries to the master program.

As it is desirable for the master program not to have to do event handling, all mouse events not related to graphics transformations are sent to the master program over the same input channel as keyboard strings (Figure 3). A facility is provided to toggle between mouse and keyboard messages along this channel (commands MouseControl/KeyboardControl, Table 1).

SIZE

GCI occupies roughly 2.0 Mbytes of disk space, including 150 Kbytes of source code. In its present version it can deal with up to 2×10^5 vectors and 2×10^5 dots, up to 10^5 points can be pickable, and 5000 labels can be used at the same time. These limits can of course be raised. Around 5×10^4 (long) vectors can be rotated in real time on a medium size 3D IRIS graphics workstation. This performance is roughly a factor of 5 less than that of some of the more evolved molecular software packages (e.g., WHAT IF[2] or INSIGHT[3]), and could be improved by optimizing the code. One should keep in mind, however, that the primary design goal was not speed, but simplicity and ease of use.

DISCUSSION

GCI as a server that implements a simple graphics command language on a particular hardware–system platform is analogous to classical drivers, e.g., for paper and pen plotters, but it is more advanced in several aspects:

(1) The graphics window is fully 3D, including liquid-crystal stereo viewing.
(2) GCI handles two-way communication with the user via a 3D graphics window, a terminal emulator and a mouse.
(3) GCI allows manipulation of the 3D view completely independently of the master programs.

Conceptually, GCI performs some of the functions of the PS300 Evans and Sutherland graphics terminals, now superseded by modern graphics workstations. Indeed, some of the concepts in GCI were adapted from the classical text-terminal-plus-graphics-terminal mode of these machines (Figure 3). However, in contrast to the elegant but complicated PS300 function network language, both the graphics command language and the source code of GCI are much simpler to understand and can therefore be implemented on new hardware in finite time.

For the developer, the concept of a strongly decoupled graphics module as outlined here has the following key advantages:

(1) A simple but functioning graphics module is available whenever a new or old program needs interactive 3D graphics.
(2) Since all graphics commands are written via standard I/O, program development can take place without a graphics device being present.
(3) Since special network communication, event handling or graphics libraries need not be linked to the master program (as would be the case using an Xwindows server for graphical output), disruption of the master program is minimal.
(4) All changes that are needed to convert 3D graphics to a new graphics device are well localized in the separate GCI module.

RECOMMENDATIONS FOR USE

Suppose one has a program in need of 3D graphics, e.g., a molecular mechanics package. After compiling and testing the GCI example as distributed (currently limited to hardware on which the Silicon Graphics GL library has been implemented), the main task is to write subroutines that transform a given object, e.g., a protein molecule, into a series of lines and dots. The appropriate data structures, e.g., residue tables with atom–atom connectivities, usually exist already. Next, a simple series of commands in the GCI metalanguage (Table 1) are sent to standard I/O and can be verified in ASCII form. Finally, the master program is started as a child process of GCI in order to test the resulting 3D image. Bug reports and other feedback to the authors are appreciated.

AVAILABILITY

In summary, GCI is a simple and powerful tool to couple 3D graphics facilities to existing or new programs. Currently, GCI runs on all Silicon Graphics Iris machines. GCI source code in ANSI C, an autoinstallation file, documentation and a small example master program in FORTRAN will be placed on the EMBL electronic mail file server. Send the email message *help programs* to NETSERV @EMBL-Heidelberg.DE (internet address) for information on how to obtain the program.

ACKNOWLEDGMENTS

GCI was conceived in discussions in the Protein Design Group at EMBL and was written by G. Tuparev and supervised by G. Vriend. We are grateful to the Computer Group at EMBL for excellent systems support and to D. Fedronic for help with sockets communication.

REFERENCES

1 Jones, T.A. *J. Appl. Cryst.* 1978, **11**, 268–272
2 Vriend, G. *J. Mol. Graph.* 1990, **8**, 52–56
3 Dayringer, H.E., Tramontano, A., Sprang, S.R. and Fletterick, R.J. *J. Mol. Graph.* 1986, **4**, 82–87