VIEW Maintenance Guide

TR93-034
February 13, 1993

Larry Bergman

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175

This work was supported by the National Center for Research Resources, National Institutes of Health. UNC is an Equal Opportunity/Affirmative Action Institution.

VIEW

Maintenance Guide

Larry Bergman
University of North Carolina at Chapel Hill
2/13/93

Table of Contents

VIEW maintenance guide

Larry Bergman
2/13/93

1. INTRODUCTION

This manual is intended for use by someone who wishes to understand the internal organization of the VIEW software, either for constructing a similar system, or to make changes to the VIEW code.

The reader of this document should be an experienced C++ programmer who is thoroughly conversant with use of the VIEW system, as described in the *VIEW User's Manual*.

2. FLOW OF CONTROL DESCRIPTIONS

2.1 Control loops

VIEW has two control loops. The main loop polls the SGI event queue, performs virtual trackball updates and checks for tool event triggers.

The tool execution loop is invoked by the main control loop. Within the tool interpreter, the SGI event queue and conditional events are checked after executing each statement. If an event of any kind is detected, the tool is exited (with the state of the interpreter checkpointed as described in section 2.3), and control returns to the main loop.

2.2 The main control loop

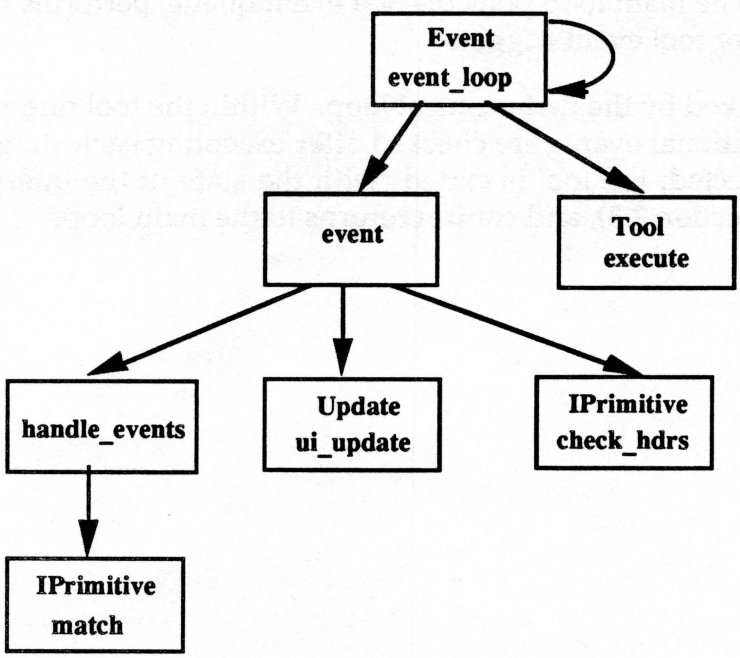
The main loop for event-handling, in the ***event_loop*** method of the **Event** class, looks like:

```
Loop forever
{
    handle mouse, dial events, check for event interrupts  by calling event
      routine.  event returns System_state, indicating any actions to be taken

    switch (System_state)
    {
        handle cases for various System_states
    }

    If there is a tool running and we are not waiting for the user to pick,
        execute the current tool
}
```

The call tree for the main loop is shown below. Arrows point to routines that are called.



The cases for *System_states* and accompanying actions are:

CONTINUE — do nothing

NEW_TOOL — the user has clicked on **Execute** from the **All tools** or **User tools** panel, starting a new tool.

- demote all **Debugger_views** associated with the last executing tool.
- get an instance of **Tool** for the selected tool from *User_tools* (the master **Tool_mgr**) and initialize it.
- flush the interrupt (tool events) and system event stacks.
- push NULL on the system event stack (to give us something to pop).
- set debugging flags.

CALL_SUBROUTINE — invoke a subroutine. The currently executing tool has encountered a subroutine call.

- get the routine from *User_tools* using information from the top of the tool call stack.
- set debugging flags and if appropriate, pop-up a debugger.

IPRIMITIVE_INTERRUPT — an event interrupt has been detected, by the user depressing a key, turning a dial, or a conditional event evaluating to True.

- have the *iprimitive* event-handler initialize the event (method ***init_interrupt*** for **IPrimitive**).
- push the current execution state onto the system event stack.
- initialize the event code and set debugging flags.

EXIT_TOOL — the user has clicked on **Exit** to stop the current tool.

- have *User_tools* clean up execution for the current tool
- if an event was executing, update the tool event queue.
- pop execution state from the system event stack.
- if debuggers are active, demote them.

UNDO_TOOL — the user has clicked on **Undo**.

- have *History_stack*, the master instance of **History**, undo tool operations.

TOOL_ERROR — the currently executing tool has encountered an error.

- create an error debugger or convert a debugger for this tool to an error debugger.
- if the tool is still executing, clean up execution for it.
- if an event was executing, update the tool event queue.

- pop execution state from the system event stack.

TOOL_EXAMINE — the user has clicked on **Examine** to create an examine debugger.

- create a debugger for the selected routine.

RUN_FROM_DEBUGGER — the user has clicked on **Execute** within a debugger.

- deactivate debugger for currently executing tool
- get an instance of **Tool** from *User_tools* and initialize it.
- flush the interrupt and system event stacks.
- push NULL on the system event stack (to give us something to pop).
- promote the debugger that **Execute** was pressed in to *running* and set debugging flags.

TOOL_INTERRUPT — untested feature to support adding an **Interrupt** button to running debuggers in order to pause execution.

TOOL_COMPLETED — the currently executing tool routine has successfully completed.

- if routine is a subroutine,
  - clean up execution for the routine.
  - pop the tool call stack
  - restore the routine at the top of the tool call stack with associated debugging state.
  - update debugger for routine exited and for calling routine.
- else,
  - if the routine is an event, update the tool event queue
  - pop the system event stack.

SET_GRAPHICAL_BREAK — the user has clicked on **Breakpoint** within a debugger. If the user has picked an object, we need to set a graphical breakpoint.

- if the user has picked an object in the display panel,
  - set a graphical breakpoint for the debugger associated with the routine at the top of the tool call stack associated with the first object in the pick list.

2.3 Tool execution

This section describes how flow-of-control passes in and out of drawing tools. The interpretation and management of tools is described under the **Tool** package, below. **Tools** are of three types: main routines, subroutines, and events. Each type of tool is

initiated somewhat differently, although once they have begun executing, the flow-of-control is identical.

A main routine is signaled by *System_state* values NEW_TOOL, or RUN_FROM_DEBUGGER. A tool is retrieved from the system **Tool_mgr**, *User_tools*, by invoking the ***get_main*** method (described under the **Tool** package below) within the ***event_loop*** method of the **Event** class.

A subroutine call is instanced when the **Tool** interpreter encounters a call. The get_subroutine method of *User_tools* is used to retrieve a **Tool** instance for the subroutine. The index number of that tool is stored at the top of the global **Tool** call stack, *Calls*. Control is then returned to the main event loop (***event_loop*** within **Event**) with the *System_state* value CALL_SUBROUTINE. ***event_loop*** retrieves the subroutine **Tool** from *User_tools*, by invoking the ***get_exec*** method. The **Tool** is retrieved by index number (obtained off the top of *Calls*).

An event is signaled by *System_state* value IPRIMITIVE_INTERRUPT. A tool is retrieved from *User_tools*, by invoking the ***get_event*** method from the ***init_interrupt*** method of the **IPrimitive** class, called from the ***event_loop*** method of the **Event** class.

A **Tool**, when retrieved from *User_tools*, is assigned to the global variable *Current_tool_exec*. At the bottom of the system event loop, the **Tool** method ***execute*** is invoked.

The ***execute*** method, in turn, invokes the ***process_stmt*** method. After each tool statement has been executed, ***process_stmt*** checks all conditional events and the SGI's event queue. If any conditionals are true, an event is on the queue, or a pick request is pending (set by a SELECT statement), ***process_stmt*** returns with the return code HANDLE_EVENT, which forces all recursive call in the interpreter to terminate, returning control to the main loop.

Interpretation of tools is described in more detail under the **Tool** package, below.

2.4 Tool events

Event processing for keys or dials is determined in the ***handle_events*** routine (in Display.c). The ***match*** method is invoked for the *iprimitive* instance of the **IPrimitive** class, and is passed the device and state last read from the SGI's event queue. ***match*** checks for the first event that is active, for which the device and state match and which is not already on the tool event queue. This last check prevents an event from interrupting itself. If a match is found, the routine returns IPRIMITIVE_MATCH and adds the index number of the event and the device state to the tool event queue, otherwise it returns IPRIMITIVE_NONE (no match), or IPRIMITIVE_IGNORE (a match, but one already on the queue). If a match, either on or off the tool event queue, is found, ***handle_events*** sets *System_state* to IPRIMTIVE_INTERRUPT and then terminates. After checking conditional events (discussed below), the ***events*** routine

(which called ***handle_events***, also found in Display.c) returns control to *events*, the system instance of the **Events** class.

2.5 History

The system undo feature operates by storing *history* information during tool execution. The **History** class maintains a circular queue of **History_nodes**, each of which stores a checkpoint as well as information about incremental changes since the last checkpoint.

Checkpoints are generated whenever the UNDOABLE keyword is encountered, either in a SELECT statement, or when an event is executed. Checkpoints are created by invoking the **History** class ***start*** method, and each corresponds to creation of a new **History_node**. When the **History_node** is created, the current tool call stack is passed to it. The call stack is copied, and for each routine on the stack, the symbol table (dictionary) and argument list are copied.

The copy of the dictionary is a limited form of copy (using the method ***copy_no_group***). Simple objects: vectors, points, colors, strings, integers, floats, chars, unsigned chars, and Booleans are copied. **Objects** and **Data_records** are not copied but simply have their reference counts incremented. This is because these objects are numerous and large, they are copied only when necessary, i.e. incrementally, when they are actually changed after a checkpoint. The names of **Databases** and **Groups** are copied and their references counts incremented, but the entire object is not copied, for the above reason. **Indexed_data_dictionarys** (arrays) and **Sets** are copied using a ***copy_no_group*** method with the rules just given.

For each routine in the call stack, the ***record_history*** method is invoked for the parse tree. This method recursively stores current state information for the parse tree. There is a circular queue at each node of the tree (in each instance of **Parse_tree**) which operates as a stack, storing the index number of the subtree currently being processed (this is considerable more efficient than copying the entire tree).

If the **History_node** is created by an event header, the tool event stack is copied.

Finally, the checkpoint records the name of all events that are currently active.

Incremental changes to **Groups** and **Objects** are handled as follows. Exactly the same algorithm applies to **Databases** (in place of **Groups**) and **Records** (in place of **Objects**).

Each group that has changed since the last checkpoint has a copy stored in **History_node** in a **Group_mgr** named *changed_grps*. The copy takes place only the first time that the group is changed (in the ***change_group*** method), prior to the change. This group will be restored on undo. If the group is created after the checkpoint, on undo the group should be removed. This is noted by a flag value of one in the integer buffer, *create_grp* (entries in *create_grp* correspond one-to-one with entries in *changed_grps*). Groups that are not created since the last checkpoint will have a zero value in *create_grp*.

For efficiency, **Group** copies do not copy the **Objects** in the group, only the system **Object_mgr**,*Obj_mgr*, index for each **Object**. Thus, when an **Object** is changed in any way, a copy of it needs to be saved, to allow for restore of the **Object** on undo (this is exactly analogous to the lack of copy of Groups when dictionaries are copied, discussed above). The **Object** copies are stored in an **Object_mgr** named *changed_objs*. A corresponding array, *changed_obj_handle_ref_cnts*, stores the reference counts to the object handle in *Obj_mgr*, the system **Object_mgr**. These handle reference counts are necessary to prevent a slot in the **Object_mgr** from being reused when its **Object** is deleted, if there is a **History_node** storing a copy of that **Object**. Such an occurrence would cause a conflict on undo. On undo, the object handle reference count in **Obj_mgr** is restored (from *changed_obj_handle_ref_cnts*) along with the **Object** itself (from *changed_objs*).

On undo, dictionaries are restored, the stack is popped at each node in every parse tree, **Groups**, **Objects**, **Databases**, and **Data_records** are restored as described above, and the active event information and event stack is restored in the system tool event handler, *iprimitive*.

2.6 Manager classes

A number of classes have manager classes which are used to keep track of instances of the class they manage. The manager classes and the class they manage are shown in the following table along with information on which of several possible manager functions, described below, each fulfills.

Manager	Class managed	instance retrieval by name	instance retrieval by index number	determine if file read needed	handle access for history	handle callbacks
Db_mgr	Database	X		X	X	
Record_mgr	Data_record		X		X	
Debugger_view_mgr	Debugger_view	X				X
Selector_mgr	Selector					X
Group_mgr	Group	X			X	
Object_mgr	Object		X		X	
Tool_mgr	Tool	X		X		

Some of the managers (**Db_mgr**, **Debugger_view_mgr**, **Group_mgr**, **Tool_mgr**), retrieve the objects they manage by name (i.e. the access keys are strings). Others

(**Record_mgr, Object_mgr**) manage anonymous objects which are retrieved by index number. The index is into an array of the objects.

The access by name or number serves two possible functions. Those that handle named entities, provide easy access to functions that retrieve objects by name. The accesses by name are:

Database

- DB() constructor and DB_WRITE function within tool language (in **Tool**)
- **Database** access in **Data_nodes and Tags**
- **Database remove** function in user interface (in ui_functions.c)

Group

- GROUP() constructor, GEO_WRITE function, setting the **Group** NAME attribute, and DISPLAY of **Groups** within tool language
- **Group** accesses in **Data_node**
- **Group operations** functions in user interface (in ui_functions.c)
- Retrieval of **Geometry** parent **Group**

Debugger_view

- Retrieval by tool name in **Tool.**

Tool

- Tool instancing from within **Event** ***event_loop*** (main routine initiation), **Tool** ***process_stmt*** (subroutine initiation), and **IPrimitive** ***init_interrupt*** (event initiation).

The other use of access by name or number is to serve as access handles. By accessing the objects through handles, the history (undo) function can replace objects without having to update all container objects that access them. The **Database, Data_record, Group,** and **Object** classes are manipulated in this manner.

Some of the manager classes keep internal representations of objects up to date with external representations on file. The two classes that do this, **Tool_mgr** (whenever a tool is requested) and **Db_mgr** (whenever a database is to be read from file using tool language DB_READ or database read called from **Group** ***file_read***) check a last-update time stored for each object against the last-update time on the file. If the file has not changed, no read is performed. This improves efficiency by buffering these (possibly large) objects.

A final function of some manager classes is to handle callbacks for FORMS widget library buttons. The use of callbacks is described in detail for the **Debugger_view_mgr** class under the **Debugger** package description below.

2.7 Iteration

The container classes in the system (including manager classes) have methods that allow a calling routine to iterate on the contents of the container.

Iteration is initiated by a call to ***start***. Calls to ***get_next*** will then return successive elements of the container in the argument list with a return value of TRUE for a successful access, or EOD if the end of the array has been reached.

Some of the iteration methods have names other than ***start*** and ***get_next***. This is because the class has more than one iterator, or more than a single argument to the iterator. For example, the class Data_subset allows iteration on records (using ***start*** and ***get_next_record***) or on field names (using ***start_field_names*** and ***get_next_field_name***). The iterator ***get_next_field*** for **Record_no_db** and **Record_with_db** has two output arguments — a field name and a data value.

3. CLASSES

3.1 Package descriptions

The VIEW system code is located in a set of *packages*. Each package contains a related set of code, grouped for convenience. Each package is compiled into a separate library. The individual packages are described below.

3.1.1 Alloc

This package contains classes that manage storage allocation. For efficiency, some of the system classes do not allocate storage using the system-supplied ***new*** operator (which in turn invokes ***malloc*** for each allocation), but rather uses classes in this package to allocate a large number of objects in a chunk and then rapidly supply a pointer to one of these objects whenever ***new*** is invoked.

The individual classes are:

Data	— base class for generic storage node
FAllocList	— a List of **FastAllocs**, all of the same chunk size.
FastAlloc	— maintains fixed number of **Data** chunks of a given size.
List	— base class for generic list of storage nodes
Mem	— top-level allocator.

The **FastAlloc** class has methods ***allocate*** and ***deallocate*** which are used to rapidly return storage blocks. **FAlloclist** contains actual calls to ***new*** which prevents recursion. **Mem** (which is requested from Alloc.h), finds the correct **FAllocList**, based on object

size, and returns a chunk from it. Detailed documentation on these classes will be found in the .c files.

To use the fast allocator for a class, include Alloc.h in the class definition file (.h file), and include the string:

ALLOCATION_METHODS()

inside the class definition. See Object.h for an example.

3.1.2 Buffer

This package contains a set of 1-D arrays for storing characters, integers, strings, and **Data_nodes**.

The individual classes are:

Bucket	— individual buckets for storing hashed strings
Buffer	— parent class for all buffers. This class has no data or methods (and could be removed)
Hash	— hash table for rapid lookup of strings in **One_d_string_buffers**. Each **Hash** object contains an array of **Buckets**.
One_d_char_buffer	— character array
One_d_data_node_buffer	— **Data_node** array
One_d_int_buffer	— integer array
One_d_string_buffer	— string array

All buffers are assigned with an initial size (like arrays in C), but if additional storage is required, each will automatically expand by allocating a new, larger storage block, and copying previous information into the new area (using ***enlarge_buffer*** methods).

All buffers support a basic set of operations including copying (operator=), indexing (operator[]) and appending. All buffers also support iteration.

One_d_string_buffers and **One_d_data_node_buffers** do not own items that have been inserted or appended to them, i.e. copy into the buffer is a shallow copy (although **Data_nodes** have a reference count which is incremented by insert operations). Copy operations (operator= or copy constructor) are deep copies, however.

3.1.3 Color

This package contains the single class **Color**, used to store rgb (red, green, blue) color definitions.

3.1.4 Database

This package contains classes for VIEW database and record types.

The individual classes are:

Data_record	— parent class for record types
Data_subset	— individual groups of records, each with exactly the same record format.
Database	— a named database consisting on one or more subsets
Db_mgr	— manages **Databases** by name
Record_mgr	— manages **Data_records**
Record_no_db	— a set of data fields not contained within a **Database**
Record_with_db	— a set of data fields contained within a **Database**

Each **Database** consists of one or more **Data_subsets**. Each **Data_subset** has a *tag_name* (e.g. *atom* or *bond*) and a single *key_name* that will be used to access its records (e.g. *atom_num* or *bond_num*). **Data_subsets** in turn contain records. This is implemented as an array (**One_d_int_buffer**) of index numbers that reference **Record_with_dbs** in the system **Record_mgr**, *Rec_mgr*.

Data_record is a parent class for the two types of records. **Record_no_db** is a record that has not yet been inserted into a database. A record of this type is constructed by drawing tools using a RECORD constructor and then fields added using the record.field_name syntax. **Record_with_db** are records that are contained within **Data_subsets**. Each **Record_with_db** belongs to a single **Data_subset**; it may not be shared. This is because the record has a pointer to its parent subset. This pointer allows a record to obtain its parent **Database** (for resolving the DB attribute for records in the tool language); to obtain a *tag_name* (for setting Object DB_PTRs in the tool language); and to reference a common set of *field_names* in the **Data_subset**, thereby avoiding the need for duplicating this array of names in each record of the subset.

Records are stored in instances of **Record_mgr**. This allows records to be replaced for history (via handles), and also serves as a convenient container (*changed_recs*)for storing records in **History_node** that have been changed since the last checkpoint.

Databases are stored in instances of **Db_mgr**. This allows databases to be accessed by name, the access method used in the DB() constructor in the tool language. **Db_mgrs** are used to determine database index numbers for object tags in geometry files and to store databases in **History_node** that have been changed since the last checkpoint. Both **Data_nodes** and **Tags** reference **Databases** by name in a master instance of **Db_mgr**, *Data_mgr*, used to store all Databases read or created within the system.

3.1.5 Debugger

This package contains all classes required for managing VIEW debuggers.

The individual classes are

Debugger	— stores all debugging information for a particular **Tool**
Debugger_view	— manages a single debugging panel on-screen
Debugger_view_mgr	— manages all **Debugger_views** currently defined
Exec_debugger	— manages **Debugger_views** for the currently executing **Tool**

There are also several files containing C routines:

re_fail.c	— error handler for *re_exec* routine in regex.c
regex.c	— regular pattern matching routines. Used by the find/replace facility in VIEW debuggers.

Debuggers store all debugging information for a single routine (**Tool**). This includes pointers to all **Debugger_views**, a list of all textual breakpoints, and a list of graphical breakpoints. **Debuggers** may be obtained by name using the *get_debugger* method in **Tool_mgr**. The master **Tool_mgr**, *User_tools* keeps a pointer to the **Debugger** associated with each **Tool** defined.

Debuggers manage **Debugger_views** in the sense that **Debugger_views** are produced by invoking **Debugger** methods. Queries about breakpoints (i.e. are we at one?) are always directed to a **Debugger**.

Debugger_views each manage a single debugging panel. **Debugger_views** come in three types, EXAMINE_DEBUGGER, EXECUTION_DEBUGGER (called a *running debugger* in the interface manual), and ERROR_DEBUGGER. Execution **Debugger_views** are also know as *active* views. The type is indicated by a flag in the instance data, and **Debugger_views** have sets of methods for converting from one type to another (*promote* – which means change to Active or Error and *demote* – which means change to Examine).

Debugger_views have methods that are used for handling all button actions — **Execute, Save, Examine, Step, Next,** etc. These methods are all invoked by C callback routines defined in Debugger_view_mgr.c.

There is a single instance of **Debugger_view_mgr**, called *Debug_view_mgr*, which store pointers to all defined **Debugger_views**. This serves two purposes. First, it allow **Debugger_views** to be accessed by the name of the tool that they display. This is used within **Tool**, to access an open debugger view that is displaying the **Tool**'s code, but was not used to run it.

Secondly, the **Debugger_view** functions use callbacks. This requires that a common C function be invoked for each (e.g. pressing the **Step** button in any debugger invokes the C callback function ***step_callback***). The callbacks are associated with a particular debugger by assigning an index number to each **Debugger_view**. The index number specifies an entry in a global array called *Debugger_view_list*, managed by **Debugger_view_mgr** (since we only have a single instance of the class, this global variable is not a problem). Each callback function invokes a **Debugger_view** method for the appropriate entry in *Debugger_view_list*.

The **Exec_debugger** class keeps track of the which **Debugger_view** is associated with the currently executing **Tool** (active **Debugger_view**), and keeps a list of **Debugger_views** for all routines in the current execution trace (i.e. that are on the Tool call stack). There is a single instance of **Exec_debugger**, *Exec_dbg*, instanced in the **Event** constructor (which is also instanced only once).

The information in **Exec_debugger** serves two uses. First of all, *Exec_dbg* is queried to see if we're in the active **Debugger_view** whenever debugger buttons are pushed which are only valid for the executing routine (**Continue, Step, Next**).

Secondly, the **Exec_debugger** demotes all **Debugger_views** in the execution trace, when the executing routine is exited (by the user pressing **Exit**).

The ***add_active_view*** method in **Exec_debugger** will create a new **Debugger_view** for a given **Debugger** and add it to the active list.

3.1.6 Directory

This package contains the single class **Directory**, which returns a set of file names given a UNIX directory.

3.1.7 Display

This package contains C++ classes and C routines for handling a variety of display functions including display management, user-interface management, virtual trackball, and picking.

The Display.c file contains a set of C++ functions (but no class) that manage the display panel, including event handling, display refresh, virtual trackball control, and display group management.

The individual classes are:

Pick	— information about a single object returned from a pick operation.
Pick_list	— all objects (Bs) returned from a pick operation.
Update	— manages the virtual trackball.

There is a single instance of **Update** in the system, *update*, instanced in the *setup_display* routine (in Display.c).

There are also several files containing C routines:

trackball.c	— trackball utility routines.
ui_forms.c	— routines for creating and updating user-interface panels.
ui_functions.c	— routines for processing user-interface actions.
vect.c	— vector functions for use by trackball.

The panel creation routines in ui_forms.c set callbacks for the user-interface objects (buttons and file-lists) that invoke callback routines in ui_functions.c.

3.1.8 Event

This package contains classes which perform the main VIEW event handling.

The individual classes are:

Event	— main event handler for the system
Event_stack	— The system call stack, one stack record per executing routine

There is a single global instance of **Event** in the system, *events*, instanced in the main routine.

There is a single global instance of **Event_stack**, *Event_queue*, instanced in the **Event** constructor. *Event_queue* is used to keep track of system state as described in section 2.2.

3.1.9 Forms

This package contains additions to the FORMS widget library. The bulk of this is a **Selector** class which implements file lists (this is an extension of the **selector** class supplied with FORMS).

The individual classes are:

Selector	— file_list
Selector_mgr	— manages all Selectors

Selector_mgr manages the callbacks for pattern changes in **Selectors** using a global list called *Selector_list*. The mechanism is almost identical to that used by **Debugger_view_mgr**.

3.1.10 Geometry

This package contains the single class **Geometry**, which serves as the base class for **Groups** and **Objects**.

Each **Geometry** has an associated transformation matrix and a set of methods that allow scaling, translation, and rotation of the **Object** or **Group**.

Each instance of **Geometry** may belong to a single parent **Group**. Thus, **Objects** can access their **Group** through the **Geometry** *parent* variable. The reason for providing this variable here (rather than in **Object**), is that transformations for both an **Object** and its **Group** need to be applied when displaying (or retrieving spatial information from) an **Object**. **Geometry** manages this. Locating the parent in **Geometry** rather than **Object** also provides a hook for future development of nested **Groups** (i.e. modeling hierarchies).

Geometry maintains a bounding box. Each **Object** or **Geometry**'s extent can be determined. This allows the displayer (in Display.c) to find the global extent of all data for setting the viewing window size.

3.1.11 Group

This package contains classes for managing sets of **Objects**.

The individual classes are:

Group	— a named list of **Objects**
Group_mgr	— manager for a set of **Groups**

There is a single special-purpose indexed access of **Database** from **Db_mgr** called by the ***file_read*** method in **Group**. This method builds a local **Database** of all files specified in the file header and then uses index numbers stored in Tags on the file to set database points in the Tags associated with each Object created.

3.1.12 IPrimitive

This package contains the single class **IPrimitive**, which handles event-monitors defined for **Tools**. There is a single instance of **IPrimitive** in the system, *iprimitive*, instanced in **setup_display** (in Display.c).

The instance data for **IPrimitive** includes a array of structures (*map*), one for each event defined by a tool. Each element of this array contains (among other things) a pointer to the Tool that the event derives from (parent), the device and (if it is a keyboard event) character that trigger the event, the value of the device state the last time the event was executed (for determining dial rate when the event is executed iteratively) and flags that

indicate whether the event is undoable, may be interrupted, is currently active, and is a conditional event.

There is a corresponding array of structures (*cond*) which stores information about conditional events including a parse tree for the conditional (cond_stmt).

The instance data also includes an event stack stored in two arrays: *which* (stores the index in *map* and *cond* of each event) and *save_state* (stores the device state). The variable *stackp* contains the index number of the top of the stack. This event stack serves two purposes. First, we do not permit any event to interrupt if it is already on the event stack. This is necessary, since we have no mechanism for locking symbol tables (no *critical region* mechanism, in computer science jargon). Thus, if an event begins executing while another instance of the same event is active, they will both affect the same symbol table, and the effect will be indeterminate. To prevent this, we search the stack each time an interrupt is logged, and if the event is already on the stack, the interrupt is ignored.

Secondly, we want to be able to return the rate of dial movement (in tool language variable DIALRATE). This is handled by checking for the same event as that on the top of the stack , and computing the difference in device state between the current state, and that on the stack.

3.1.13 Log

This package manages system logging, a feature which in not currently supported.

The individual classes are:

Log	— stores information about a complete Log session
Log_node	— stores information about a single user-interface action for **Log**.

3.1.14 Node

This package contains several types of *nodes*, used for packaging information required by the tool interpreter, as well as classes for managing nodes.

The individual classes are:

Data_node	— a wrapper for most of the object types used in the tool language including numbers, strings, arrays, sets, records, databases, points, vectors, and displayable geometry.
Data_node_stack	— a stack of **Data_nodes**
Indexed_data_dictionary	— a keyed or indexed array containing **Data_nodes**
Node	— parent for **Data_node** and **Parse_node**

Parse_node	— information for a single node in a **Parse_tree**
Parse_node_stack	— a stack of **Parse_nodes**

The **Data_node** class permits objects of a variety of types to be operated on uniformly. It is basically a way of dealing with C++'s lack of polymorphism. **Data_nodes** may contain Booleans, integers, floats, chars, unsigned chars, strings, colors, vectors, points, displayable geometry objects, databases, arrays, sets, groups, or records. There is also a *special* type that may be assigned. Currently the end-of-data flag, END_OF_DATA, is the only special type.

Data_nodes contain a reference count. This permits them to be shared by any number of container classes (**Indexed_data_dictionary, One_d_data_node_buffer, Set**) with no copying. **Data_nodes** also contain a line number used to record the line in the defining tool at which they are defined or used. This is useful for error reporting.

The **Parse_node** class contains information about a single node in the parse tree for a tool. The information includes start and end line and column for the code statement that the node corresponds to, data for the node (may be a number, a string, a Boolean, an event name, or a database key), a variable name (*id*), an index in the symbol table (*dictionary_index*), an op code, and a flag that indicates whether or not to stop for debugging when the node is encountered in interpretation.

Indexed_data_dictionary is a keyed or indexed array of **Data_nodes**. **Indexed_data_dictionarys** are used to store **Tool** symbol tables, and also to store keyed or indexed arrays defined in tools.

Data_node_stack is a stack that manages **Data_nodes**. There is a single instance of it, *data_stack*, used by the tool interpreter. Use of *data_stack* is described in the **Tool** package description.

Parse_node_stack is a stack that manages **Parse_nodes**. There is a single instance of it, *Node_stack*, used by the tool parser. Use of *Node_stack* is described in the **Tool** package description.

There are also a single file containing C functions:

mem.c	— storage tracking

The routines in this class allow the code developer to track storage use. The number of objects of each type, dynamic allocation and deallocation, and improper use of memory can be tracked using special environment variables and/or compiler options. The use of this facility is extensively documented in the code.

3.1.15 Object

This package defines all displayable geometric objects and their management.

The individual classes are:

Bbox	— a bounding box for object of type **Geometry** (either an **Object** or a **Group**)
Cylinder	— a cylinder
Line	— a 3-D line
Object	— parent class for all displayable geometric objects
Object_mgr	— manages a set of **Objects**
Sphere	— a sphere
Tag	— a database pointer for an **Object**
Tag_buffer	— an array of **Tags** associated with an **Object**
Text	— a 3-D text object
Triangle	— a triangle

The parent class for displayable objects, **Object**, stores color, database pointers (**Tags** in a **Tag_buffer**), a pointer to the tool that created the object, and a copy of the call stack from the tool that created this object (if it was created in this VIEW session). The call stack is used to determine which debugger is to have an entry added to it if the object is selected as a graphical breakpoint (by referencing the **Debugger** associated with the **Tool** pointed to by the top of the call stack). The object also contains in index number in *Obj_mgr*, the system **Object_mgr** where the object is stored.

The **Tag** database pointer consists of a **Database** name, an index of a **Db_mgr** that contains the Database, a subset index number within the **Database** (*type_index*), and a key value for the record pointed to. This is all the information required to retrieve a particular record from any **Database** in the system. The reason for both a **Database** name and a **Db_mgr** index (which appears to be replicated information), is that the **Db_mgr** index is used in producing a **Tag** representation for output in a geometry file. This process is described in the **Group** package description. For other uses, the **Tag** must reference the **Database** by name, not by index number, since the locations of **Databases** in *Data_mgr* (the system **Db_mgr**) may not be fixed. Note that the term *tag type*, which is used within the **Tag** routines, is synonymous with subset name.

Tag_buffer is a one-dimensional array of **Tags**. This allows more than one **Tag** to be associated with an **Object**. Note, however, that if more than one **Tag** is of the same type (i.e. has the same subset name), only the first will be retrievable. This is because of the way in which ***select_tag***, the method that gets a **Tag** from the buffer by type, is implemented.

The displayable objects, **Cylinder**, **Line**, **Sphere**, **Text**, and **Triangle**, each have a methods for reading themselves from geometry files. The ***read_all*** method reads all data for the object by first invoking the **Object** ***read_all*** method to read data common to all objects and then by invoking the object-type-specific ***read_grp*** method to read the data for that type. The ***write_all*** and ***write_grp*** methods operate similarly for writing geometry files.

The *render_sgi* method is implemented for each object and executes the GL library commands required to render the object. The **Cylinder** object is a bit unusual, in that intermediate geometry (spheres or triangles) is produced for the caps of the cylinder and stored in a local **Group** created by the **Cylinder**. The **Cylinder**'s *render_sgi* method invokes *render_sgi* for each of these intermediate objects.

3.1.16 Profile

This package manages system snapshots (previously know as *profiles*). The single class in this package, **Profile**, is instanced by user interface function in ui_functions.c (in the **Display** package) which read and write snapshots.

Profile has a single method, *open_profile*, which is passed a flag indicating whether to read a snapshot (flag value PLAYBACK) or to write a snapshot (flag value RECORD). Snapshots are written in a human-readable ascii format, which is interpreted by **lex** and **yacc** code.

3.1.17 Queue

This package contains the single class **Circular_int_queue**, a circular queue for storing integers, used by **Tree** classes.

3.1.18 Set

This package contains a single class **Set**, which implements mathematical sets for storing Data_nodes.

The methods include standard set operations such as union, intersection, and difference. In addition to the standard iterators, **Sets** have a method, *get_indexed*, that allows an element to be retrieved by index number. Although sets (in the mathematical definition) have not ordering, this function allows **Tool** to iterate on a **Set** and keep track of where in the **Set** the iteration is (important since the **Tool** may be interrupted by user actions).

3.1.19 ThreeSpace

This package defines 3-D mathematical entities including points, vectors, planes, and 3-space transformations.

The individual classes are:

Mat3x3	— 3-dimensional transformations, excluding translation
Mat4x4	— 3-dimensional transformations, including translation
Plane	— 3-D plane

Point — 3-D point
Vector — 3-D vector

The **Mat3x3** class is provided to allow for more efficient operation than with **Mat4x4s** (with a corresponding loss of compact notation / elegance). It is currently used for rotating the axes of cylinders into position when tessellating them.

The **Vector** class stores an x,y,z triple that represents the vector. It also may store a starting point that is used to position a graphical representation of the vector when using the debugger **display** facility.

3.1.20 Tool

This package contains all code required for running drawing tools including the tool parser and interpreter, subroutine calls and stack management, and the history (undo) mechanism.

The individual classes are:

Argument — information about a single argument for a subroutine
Argument_list — a list of **Arguments** for a subroutine
Call_stack — call stack information for managing subroutine calls
Ext_call_stack — additional call stack information not required for all uses of **Call_stack**
History — history stack used for managing undo
History_node — information for a single history checkpoint
Tool — information for a single routine (main tool or subroutine)
Tool_mgr — manages all Tools

Each **Argument** stores an index in the subroutine dictionary (symbol table) for the argument and also an index in the calling routine dictionary. The subroutine index is set when the PARAMETER statement of the subroutine is parsed. The index for the calling routine is set when the **Tool** interpreter processes the subroutine call. **Argument** also contains a pointer to a **Data_node** which hold the argument value.

Call_stack stores a running index (i.e. an index into *exec_tools* and associated arrays as described below) in *User_tools* , and a line number in the tool of the current statement for each **Tool** on the call stack.

History contains a circular queue of **History_nodes**. Contents of **History_node** and its use is described is Section 2.5 above.

The **Tool** class is split into two routines, Tool.c and Tool2.c for tractability. The **Tool_mgr** class invokes a **yacc** parser for the tools, stored in Tool.yacc which in turn invokes a **lex** syntax scanner stored in Tool.lex.

Tool_mgr keeps track of all **Tools** (main routines, subroutines, or events) that have been invoked. Each **Tool** (each event is assigned an individual Tool) is assigned a *template* that will be used to instance running versions of the routine. The template consists of (among other things) a parse tree, a symbol table (dictionary), a debugger, an argument list (NULL if the routine is not a subroutine), and a table of event names. In addition, if the **Tool** is an event, the index of the *parent* **Tool** is stored, i.e. the **Tool** that defines the event.

Each time a routine is invoked, an executing version of the **Tool** is created. This allows the routine to be called recursively, with each copy retaining local data copied from the master template. The array *exec_tools* contains the **Tool** template. *exec_template* contains the index number of the template. *exec_reserved* indicates whether this slot is in use, enabling us to reuse slots when an executing **Tool** has gone out of scope. If also allows us to reuse the **Tool** in a slot without recopying its parse tree, symbol table, and argument list (merely resetting them) if the **Tool** code has not changed since last use (determined by checking the last-update time on the file (stored in the *tool_last_update* portion of the template) against the last-update time for the executing version (stored in *exec_last_update*).

When a Tool is being parsed, an execution slot needs to be reserved for it in advance. This is because error processing (in **Debugger_view**) uses the execution slot indices in the call stack to determine which routine to display. If a parse error is encountered, this information must already be set for error reporting to operate properly. A special method, ***create_reserve_running***, in **Tool_mgr** creates a slot without copying dictionaries, parse trees, and argument lists. This method is also used for storing event information, which does not requires copying on slot creation (since this information comes from the parent **Tool**).

When parsing a **Tool**, a **Parse_tree** is built containing a representation of the code, one **Parse_node** and subtree for each statement or phrase. The **Parse_node** for a statement contains the line number and column of the start and the end of the statement. Since statements may be nested, the parser keeps a ***Parse_node_stack***, *Node_stack* of nodes that it is processing. Each node is pushed on the stack by the **lex** (syntax scanner) code. **Lex** is signaled that a new statement is beginning, and that a node is to be pushed, by having **yacc** (the parser) set a flag, *start_cmd_flag*, to TRUE. When this flag is true, and **lex** is processing a string (all commands start with a variable name or keyword), a node is created and pushed onto *Node_stack*. *start_cmd_flag* is carefully set in such a way that it will already be true when **lex** starts a new statement, even though **lex** may scan one token ahead (i.e. have already processed the first token of the next statement when **yacc** recognizes a completed statement).

Tool interpretation is done by recursive descent. The **Parse_tree** is passed to the ***process_stmt*** method of **Tool**, which does some processing to determine if we are at a breakpoint and need to stop for debugging (described below), and then using an operation code from the node at the top of the tree to determine the type of statement being processed. The code for the various statement types is contained within a case statement, and the interpreter selects the appropriate code segment and executes it.

Sequences of statements are handled by having a special op code (SEQUENCE). The **Parse_trees** that correspond to each statement of the sequence are stored as subtrees of the sequence node. An index is stored at the node, indicating which statement is currently being processed. As each statement is completed, the index is updated in the node, making it possible to relocate at the correct statement if the **Tool** is interrupted (by an event). The same mechanism is used to keep track of iteration counts on interruption.

Each statement of the sequence is executed by invoking ***process_stmt*** with the **Parse_tree** that corresponds to the statement. Phrases within a statement are processed by passing the **Parse_tree** for the phrase to ***process_substmt***, which is also a recursive descent interpreter with code segments for each phrase type.

The interpreter uses a **Data_node_stack**, *data_stack*, to store results of intermediate calculations. When ***process_substmt*** obtains or calculates a result, it is pushed onto the stack, and later popped by a calling instance of ***process_substmt*** or by ***process_stmt***.

process_stmt determines whether the tool needs to pause for debugging, because it is at a textual breakpoint, is at a graphical breakpoint, or **Step** or **Next** have been pressed. If the tool is at a breakpoint, a **Debugger_view** is created (if none exists) or retrieved, the breakpoint highlighted, the global flag *Tool_state* is set to PAUSE, and control returned to the main control loop. With *Tool_state* set to PAUSE, the control loop will not invoke the **Tool** interpreter. When the user presses **Continue**, **Next**, or **Step**, the value of *Tool_state* is changed, and interpretation resumed.

Because the interpreter is re-entrant, some record-keeping is required to determine whether or not to pause for debugging at a breakpoint. Whether or not to pause is further complicated by fact that after **step** is pressed, the tool is to pause at the next statement encountered, whereas after **next** has been pressed, the tool does not pause until the next statement at the same nesting depth or higher is encountered (by nesting depth I mean how many nested blocks the statement is within).

Nesting depth is retained by a global variable, *block_depth*. As a statement is started, *block_depth* is incremented; when a statement is completed, *block_depth* is decremented (by the ***complete_statement*** method). Thus, at any time, *block_depth* will give the nesting depth of the current statement, even if the interpreter has been interrupted for event processing. Note that *block_depth* is reset to zero each time the Tool execute method is called. This avoids any worry about whether we are reentering the routine or not. The proper *block_depth* will be available when the interpreter has returned to the interrupt point.

Another depth indicator, *stop_depth*, allows us to distinguish between **Step** and **Next**. *Stop_depth* is restrained to be less than or equal to *block_depth* (this restriction is imposed in ***complete_statement***). *Stop_depth* is also set equal to *block_depth* when **Step** or **Next** is pressed. When the tool is entered, if **Next** has been pressed, the tool will pause for debugging only when *stop_depth* equals *block_depth*, i.e. when the statement AFTER the one currently being executed is encountered (it will not stop on the current statement due to use of debug flags discussed below). If **Step** has been pressed, the tool pauses if *block_depth* is greater than or equal to *stop_depth*. This allows for pauses after stepping into a block.

There is also a check to see if we are **Step**ping, or **Next**ing with *stop_depth* equal to *block_depth* , within iterators. This allows us to single step through iterators.

Parse_nodes have a flag (*stop_for_debug_flag*), which indicates whether the interpreter is to stop at the corresponding statement when a breakpoint is encountered or step or next pressed. When we stop at a statement, the flag is set to FALSE. We will not stop for debugging at that statement again, until the flag is set back to TRUE, which happens when the statement has completed (in ***complete_statement***).

3.1.21 Tree

This package provides parse trees used by the Tool class.

The individual classes are:

Loop_tree	— **Parse_tree** for a loop structure
Parse_tree	— stores parse tree for portion of a tool language statement, an entire statement, or a group of statements
Tree	— parent class for all Trees

Trees contain a single **Parse_node** and a variable number of children or (*subtrees*) each of which is a **Tree** (actually a **Parse_tree**. The **Tree** class is really superfluous - all the code could be folded into **Parse_tree**).

3.2 Class structure

Below is a an alphabetized listing of all parent classes and the classes that derive from them. Note that each VIEW class derives for a single parent class (that is, no multiple inheritance is used). Each derived class is shown offset by a tab from its parent class. The package that each is found in is given in parentheses.

Argument (Tool)
Argument_list (Tool)
Bbox (Object)
Bucket (Buffer)

Buffer (Buffer)
    One_d_char_buffer (Buffer)
    One_d_data_node_buffer (Buffer)
    One_d_int_buffer (Buffer)
    One_d_string_buffer (Buffer)
    Tag_buffer (Object)
Call_stack (Tool)
Circular_int_queue (Queue)
Color (Color)
Data (Alloc)
    FastAlloc (Alloc)
    List (Alloc)
        FAllocList (Alloc)
Database (Database)
Data_node_stack (Node)
Data_record (Database)
    Record_no_db (Database)
    Record_with_db (Database)
Data_subset (Database)
Db_mgr (Database)
Debugger (Debugger)
Debugger_view (Debugger)
Debugger_view_mgr (Debugger)
Directory (Directory)
Event (Event)
Event_stack (Event)
Exec_debugger (Debugger)
Ext_call_stack (Tool)
History (Tool)
History_node (Tool)
Hash (Buffer)
Geometry (Geometry)
    Group (Group)
    Object (Object)
        Cylinder (Object)
        Line (Object)
        Sphere (Object)
        Text (Object)
        Triangle (Object)
Group_mgr (Group)
Indexed_data_dictionary (Node)
IPrimitive (IPrimitive)
Log (Log)
Log_node (Log)
Mat3x3 (Node)
Mat4x4 (Node)

Plane (Node)
Mem (Alloc)
Node (Node)
    Data_node (Node)
    Parse_node (Node)
Object_mgr (Object)
Parse_node_stack (Node)
Pick (Display)
Pick_list (Display)
Point (Node)
Profile (Profile)
Record_mgr (Database)
Selector (Forms)
Selector_mgr (Forms)
Set (Set)
Tag (Object)
Tool (Tool)
Tool_mgr (Tool)
Tree (Tree)
    Parse_tree (Tree)
        Loop_tree (Tree)
Update (Display)
Vector (Node)

4. UNDOCUMENTED FEATURES AND HOOKS

4.1 Undocumented language features

A number of tool language features have been implemented but are not well tested and therefore not covered in the language description. Many of them will work as described below, others will require additional debugging and/or development.

4.1.1 Mouse, keyboard, and dial information

Two system-defined variables are available that give information about the mouse state. **MOUSE_BUTTON** returns the state of the right mouse button. **MOUSE_BUTTON** will have either the string value "D" (for down) or "U" (for up).

MOUSE_POSITION returns a point containing the screen location of the mouse cursor within the display panel. The x and y coordinates of the point are the x, y location of the cursor (origin at the lower left) in pixels. The z coordinate is zero.

The system-defined variable **KEYCHAR** contains the last key that triggered an event as a string. This could be used outside an event to determine what the last keyboard event was.

The system-defined variable **DIALNUM** contains the last dial that triggered an event as an integer. This could be used outside an event to determine what the last dial event was.

4.1.2 Executing tool information

The system defined variable **CURR_TOOL** contains the name of the currently executing tool as a string.

4.1.3 Screen origin

A system-defined variable **ORIGIN** can be used to retrieve or set the screen center. Thus, a statement of the form:

ORIGIN = pnt;

Where *pnt* is a point, will recenter all geometry with *pnt* as the screen center.

The screen center in world coordinates can be retrieved with a statement of the form:

pnt = ORIGIN;

4.1.4 Variable dump

The function **SHOW_DICT()** produces a print of all dictionary (symbol table) entries for the tool being executed. This is analogous to dbx's ***dump*** statement.

4.1.5 Optional command arguments

An optional last argument to **DISPLAY**, **NOREDRAW**, provides for invoking all **DISPLAY** functions but without forcing a screen refresh. The syntax is:

DISPLAY (<variable 1> [, ... <variable n>] ;NOREDRAW) ;

The same argument with identical syntax is available for **UNDISPLAY.**

An optional last argument to the **POINT** constructor, **SCREEN**, provides for specification of points in world coordinates using screen space locations. x and y for the point should be specified in pixels relative to (0,0) at the lower left corner of the display panel. the z coordinate should be zero. The syntax is:

POINT(<x>, <y>, <z> ;SCREEN)

The system will convert x, y, z to world coordinates.

An optional argument in an event header, **UNINTERRUPTABLE**, allows one to define events that cannot be interrupted by other events. **UNINTERRUPTABLE** appears as a phrase preceded by a semicolon, directly before, directly after, or in place of the **UNDOABLE** keyword.

4.1.6 Additional attributes

Screen coordinates can be obtained from a point or vector using the attributes SCREENX and SCREENY. The coordinates will be in pixels with the origin (0,0) at the lower left corner of the screen. These attributes may be used only for retrieval, not for setting coordinates.

EXAMPLES:

```
x = pnt.SCREENX;
y = pnt.SCREENY;
```

4.1.7 Display of points

The SGI supports a point display primitive which will appear on-screen as a constant sized (2-pixel wide) dot. These display points can be generated by specifying a SPHERE of radius zero.

4.1.8 Line thickness

An optional final argument to the LINE contructor may be supplied specifying the line width. This should be an integer value between one and ten. The default value is one pixel wide. Note that not all models of SGI support variable line widths, others support only one-pixel and two-pixel-wide lines.

4.1.9 Conditional Events

Conditional events provide a mechanism for automatic evaluation of conditional expressions within the main event loop, and triggering of events when conditions evaluate to True. The syntax is of a conditional event is:

EVENT (<event name> **;** <relational expression> [key phases] **)**
<event body>

<key phrases> may be either **;UNDOABLE, ;UNINTERRUPTABLE,** or both. The <relational expression> (or conditional) is evaluated on each iteration of the event loop using the current values of any variables contained within the expression. Thus the event header:

```
EVENT ("check_distance"; (DIST(pnt, obj.CENTER) < dist_limit))
```

Will trigger the event "check_distance" whenever the three-dimensional Euclidean distance between the point, *pnt*, and the center of the geometric object, *obj*, is less than the value, *dist_limit*.

Conditional events are not automatically activated (unlike ordinary events). You must execute the START_EVENT statement to signal the system that the event header is to be evaluated.

Conditional events slow the operation of VIEW substantially, both tool processing and user-interface response. This is because the interpreter must be run on the relational expression at each iteration of the main event loop. For this reason, you should limit a tool to two or three concurrently active conditional events.

Conditional events have one major known bug. Once one is defined, there is no way for the system to disable it (although the user can disable the event using **STOP_EVENT**). This means that when a error condition in the event is encountered, the event will continue to be evaluated and a whole sequence of pop-up debuggers produced. Having the system disable the conditional event on an error condition should be easy to implement.

4.1.10 Three-dimensional search

A three-dimensional search function, especially designed for use in conditional event headers (but also useful elsewhere) is available. The SEARCH3D function allows you to search a list of points for any that are located within a specified distance of a 3-D point. If any points fall in the search volume, they are returned in an output list, and the function returns a value of TRUE. If no points meet the search criteria, the function returns a value of FALSE.

SEARCH3D, has the syntax:

SEARCH3D (<number pnts>, <input pnt list>, <probe center>,
<probe radius>, <output pnt list> [<optional phrases>])

where <number pnts> is an integer value specifying the maximum number of points to be retrieved in the search. This number can be used to limit the amount of computation, and improve efficiency. If all points are to be returned, use the keyword **ALL** in place of an integer value.

<input pnt list> is an array (either indexed or keyed) or a set of points. This is the list to be searched. <probe center> and <probe radius> specify a search location (point) and a search radius (number) respectively. Each point within <input pnt list> will be tested for having a distance from <probe center> that is less than <probe radius>. Each point that meets this criteria (until <number pnts>) have been found will be copied into an indexed array of points, <output pnt list>. If <number pnts> has the value of one, this is treated as a special case, <output pnt list> will consist of a single point rather than an array. If any points are added to <output list>, the function will return TRUE.

The optional phrases are:

;INDEX = <index array>

and

;KEY = <key array>

the **INDEX** phrase, which is only valid if <input pnt list> is an indexed array, will return an indexed array containing the index number in <input pnt list> for each entry in <output pnt list>. Thus <index array> and <output pnt list> will have a one-to-one correspondence. Similarly, the **KEY** phrase returns the key values for each entry in <output pnt list> in an indexed array, <key array>, and requires that <input pnt list> be a keyed array.

4.2 Partially implemented features

4.2.1 ASK_FILE

A tool language keyword to pop up a file_list, **ASK_FILE**, is partially complete. This feature is intended to be analogous to **ASK_STRING** and **ASK_NUMBER**, but for selecting file entries. The keyword is recognized by the **Tool** syntax scanner and the parser (creating a **Parse_node** with the op code **ASK_FILE**). No interpreter code has been written for it, however.

4.2.2 Per-object shading attributes

There are instance variables and methods in Object.h that define Phong lighting model attributes for an **Object**. Since SGI's do not currently support Phong shading, these have not been used, but could easily be activated if VIEW were ported to another graphics engine or if SGI's become available that support this lighting model.

4.2.3 Debugger pop-up and database information on object selection.

A previous version of VIEW was able to provide database information when an geometric primitive was clicked on in the display panel, and also create a pop-up debugger showing the line of code that created the element, with the accompanying call stack. This latter facility used the call-stack stored in the **Object**. This code has not been maintained, but is still present, commented out, at the bottom of the ***event_loop*** method in **Event**.

4.2.4 Interrupt

Some untested code to support an **Interrupt** button, to allow the user to halt execution of a tool for debugging, will be found under the case TOOL_INTERRUPT in the

event_loop method in **Event**. The button and callback code required for this feature have not been written, but follows in a very straightforward manner any of the existing debugger buttons.

4.2.5 Depth-cueing and anti-aliasing

A depth-cueing flag is available as a command-line parameter when invoking the VIEW program. The flag is -depthcue. The problem with this feature is that SGI's depthcueing overrides object shading. Thus 3-D primitives such as spheres and cylinders will appear to be flat-shaded when using this object. I've experimented with using SGI's atmospheric shading feature (which is not supposed to have this problem), but have not been able to get it working properly.

There is commented-out code in the main panel creation routine (in ui_forms.c in the **Display** package) that provides toggle buttons for anti-aliasing and depth-cueing. I have been unable to get this feature to operate.

4.2.6 Logging

Logging operates in two mode, record and playback. Record mode is initialized by the command-line option: -record *log_file_name* . Playback is initialized by the command-line option: -playback *log_file_name*. When recording, all user-interface actions will be recorded (except debugging operations) during the entire VIEW session. When running a playback, all the log commands will be executed, and when the log file is completed, the user may continue to work manually.

The log files are stored in a subdirectory with the name *log_file_name* created in the Log subdirectory of the data directory used to run VIEW. Each log file is ascii human-readable text, and stores information about operations selected, on-screen geometry selection and names of files chosen. In addition to storing a log file, VIEW creates a copy of any geometry files, database files, and tool files required to completely recreate the VIEW session. The reason for duplicating these files is to ensure that changes to the files in their original location will not prevent an accurate replay of the session at a later time.

Logging is close to completely implemented, but a few features are not included (most notably database remove). Additionally, there is a bug in storing information about on-screen geometry selections.