# Eliminating the Dichotomy Between Scripting and Interaction

*John F. Schlag*

Three Dimensional Animation Systems Group
Computer Graphics Laboratory
New York Institute of Technology

## Abstract

*Throughout the computer graphics literature, but in modeling and animation lore in particular, the prevalent attitude seems to be that scripting and interaction are irreconcilably different types of user interface. This paper propounds the belief that this dichotomy is a myth and gives examples of systems which encourage this belief. Two of the systems described are a keyframe animation program and a geometric modeling program developed at the NYIT Computer Graphics Laboratory. Both of these systems are used on a daily basis for production, development and research.*

## 1. Introduction

Scripting and interaction are widely used interface styles for modeling and animation programs. The advantages and disadvantages of each are well known. Scripting systems allow command sequences to be incrementally modified and reexecuted, but allow no interactive input of data and operations. Interactive systems allow this input, but lose all record of commands previously executed, leaving the user to start over if some intermediate parameter affecting the final result needs to be changed. In almost every publication describing a new modeling or animation system, there is a paragraph or two which notes these properties and proceeds to ballyhoo the advantages of the method chosen. As with many other such apparent dichotomies, most designers accept without question the division, pick one technique, implement it and resign themselves to writing the obligatory text. From this lamentable situation we may conclude that the production of a user interface which integrates scripting and interaction is ***difficult***, but certainly not that it is impossible.

The integration of scripting and interaction is worthwhile for several reasons. The first is that a more general conceptualization of user interfaces results. This has the potential for expanding the set of problems that can be solved with a computer. On a more practical note, interactive graphics workstations are often expensive and therefore scarce. In many graphics houses, the time available on such resources is a limiting factor in the amount and/or complexity of animation producible. It helps greatly if users can work, albeit more slowly, at ordinary terminals, using scripting instead of interaction.

Resource limits are not the only reason to use scripting at one workstation and interaction at another. The production of computer animation requires many diverse activities, most of which are accomplished more efficiently with one technique than the other. Some establishments may boast interactive *and* scripting tools for particular activities, but rarely do these tools interface to a common database format. More often, the systems are incompatible "competitors", perhaps having been written by different individuals, and any attempt to use both to solve a single problem is frustrated by the need to reorganize data, shuffle files about and translate between formats. A system which integrates scripting and interaction can be used for conceptually diverse activities in whatever mode is appropriate. A banner example of the type of production which needs this is the now famous robot ant animation by Lundin [15], in which the basic 3d path of the model is supplied interactively and the dynamics are supplied by an explicit computational model.

For perspective, the more general problem here is the development of *editing theory*. A general theory of editing should be independent of the data to be edited, so it should be applicable not only to ordinary text editing, but to other activities such as geometric modeling (shape editing), animation and robot programming (motion editing), music production (sound editing), painting (image editing) and computer programming (algorithm editing). What we would like for any given data format is a logical and complete set of functions for manipulating that format, and both interactive and scripting interfaces to these functions.

When differentiating between interaction and scripting, it is natural at some point to quibble about exactly what *interactive* means. Some would argue that if the time around the typical edit-process-view loop of a scripting system is short enough, the system is interactive. Is there some threshold on this loop time which must be met? "Real" time? Historically, the term has been used to mark a contrast with *batch* systems, where the sizes of input and output data sets are usually large. Systems that produce visible results after small amounts of input

(after every keystroke, say, in a screen editor) we classify as being definitely interactive. An often cited parameter for language-based systems is whether the system is interpretive or compiling. Interpretive systems, however, are rarely used without large application dependent scripts, while compilation systems can call themselves "interactive" simply by connecting their inputs to the keyboard. Some systems [13, 16] blur the distinction even further by loading compiled code into interpretive front ends at run time. The answer to all this, of course, is that the set of interactive systems is a fuzzy one, with no clear dividing lines along any of these axes. Hence, we discontinue our quibbling here.

## 2. Combining Scripting with Interaction

This section describes the requirements for an integrated scripting/interactive user interface. Several programs/systems which implement these requirements in varying degrees of completeness are drawn upon for examples. *Emacs* is an "extensible, customizable, self-documenting" screen editor developed at MIT [1]. The Unix[1] implementation with which the author is familiar [2] supports extension through an interpreter for a lisp-like language. *Troff* [5] (or *Scribe* or *TeX*, if you prefer) is a typesetting program which supports macros. The *Macintosh* personal computer [17], with its attendant firmware, is the latest paragon of user interface style. *Em* is an animation program written at NYIT for interactive motion specification (animation) of parameterized models [8]. It is unique among these example systems in its use of a large (rather much larger than that of C [4]) formal grammar [7] for defining its basic input language. *Gem* [16] is an interactive geometric modeling program, also written at NYIT, and is the current object of this research.

To develop a model of a user interface which integrates scripting and interaction, one can think either about adding an interactive front end to a scripting system, or about adding scripting features to an interactive program kernel. Both these approaches have merit. The former is more likely to result in a consistent, well-designed system, since the basis for the system is presumably a well-designed language. The latter is more likely to result in a truly flexible system, since the user interface design would proceed unfettered by syntax. Gem was developed in the latter mode, and this paper attempts to relate the experience gained from that activity. What, then, is required of an interactive system which also supports scripting?

(1) The first and most obvious requirement is that there must be some appropriate script representation. Text is an obvious choice, but the problem with text is that it forces linear formatting. This may be acceptable for document and music production, but for modeling and animation, where instancing is a way of life, a graphic dataflow format may be more appropriate. (In fact, the frustration of ordinary music notation is due in part to its limited support for instancing.) Even if a linear format is tolerable, a more cogent objection to text is that it represents the script at the wrong level. Dealing with the script directly as text ignores the higher-level command structure.

There must be a script representation of every interactive editing command, and there must be interactive access to every script command. This brings up the matter of what is and what isn't script. It is important to differentiate commands to an interactive front end from commands that manipulate the data to be edited. The interactive interface provides an *additional*, rather than an *equivalent* means of access to editing commands. This seems a sensible separation, as no one wants to type in reams of text for low level events like tablet movements anyway. In fact, it is useful to be able to siphon off the raw event stream into a file for testing and demonstration purposes (as the Macintosh can do), but there is no need for that file format to be integrated with the script representation.

(2) The system must maintain some internal representation of the script. Unix Emacs uses three representations for script material, one the lisp-like syntax mentioned earlier for defining functions, the second a buffer of raw keystrokes for defining "macros" and the third a private internal format for supporting 'undo'. Em begins by parsing a script with a yacc-generated parser [6], generating a dependency tree and display segments as semantic results. Instead of converting the script to a syntax tree representation, however (as a compiler would for code generation), the original text is stored in memory for reparsing during the interactive updating of parameter values.

(3) The system must provide a way to execute a script. This is the easy part, as pushing the current input source on a stack and reading a script from a file or memory is simple on reasonable operating systems. The only added wrinkle is that scripts should be able to turn to an absolute source (typically the user) for input. In modeling, for instance, a user may want to create a script segment that creates an instance of a highly parameterized primitive by filling in some parameters on its own and prompting for the rest. Emacs provides an assortment of functions for getting values of various types from the keyboard.

(4) The system must be able to create a script from interactive input. Emacs saves raw keystrokes for macro definition; function definition from interactive input is achieved by mapping keystrokes through a *keymap* which associates functions with keys, and then appending the names of the functions to the script. The Synclavier [21] provides a facility for "reverse-compiling" a real-time keyboard performance into a score.

Creating scripts from interactive input is where the real subtleties begin to arise. For example, most scripted operations will be used as subroutines would be in an ordinary programming language. (This restriction is reasonable, since in many languages everything is part of some

subroutine.) The context in which this "subroutine" is to be executed will dictate end conditions for the recording of the script. One solution, used by Emacs for both function and macro definition, is to provide two special interactive commands to start and stop the translation of interactive input into script. These commands must be treated specially in all contexts, as their meaning in a script is unclear at best, and they should not be translated into script when given interactively.

Some decision must be made as to whether the input being translated is to be executed at the time of recording. A good solution to this is to execute the input when it's coming in interactively, and to simply store it away when it's not. Emacs, being "interactive", executes its input while translating it, which makes clear the effects the script will have when executed later. At the other end of the spectrum, Troff defines very clearly a *copy mode* through which input destined for macros is read.

Above, it was maintained that a direct translation of the low level event stream is not an appropriate script representation. The open question is this: if raw events aren't going to be represented directly in the script, how *are* they going to be represented? High bandwidth events such as tablet and dial movements present a formidable data pollution problem, so some means of compressing these events into a compact interpretation needs to be found.

(5) The system must provide a way to edit a script. This is important, since very little time is spent creating scripts as opposed to editing them (witness software production). For this function, Emacs "cheats" and uses itself to do the editing. It is, after all, a text editor, and its only events are characters from the keyboard. Interactive modeling and animation programs which employ an arsenal of interactive devices are not so fortunate. More special commands are needed here to position the "cursor" where translated script will be inserted, and to delete script elements. Em uses a compromise solution to allow the user to edit the set of current *input modes* (relationships between logical device values and parameter values), which is to edit a text representation of the input modes with a text editor and then read them back in. The text representation is actually a subset of the command language, and the same parser is used for it that is used for the input script.

The support of editing is the most difficult part of integrating scripting and interaction. As an example, consider the support necessary for the special case of 'undo'. At any point in the command sequence, the user can issue an undo command which undoes the previous editing command, and sequences of undo commands have cumulative effect. (An interesting side issue is whether the undo commands should be part of the script, that is, whether you should be able to undo the undoing.) There seem to be only two alternatives for supporting this functionality: either all the operators must be invertible, or the entire state of the system must be snapshotted after every command, both fairly daunting propositions. In the more general case, commands at any point in the script may be changed. Assuming that an interactive system wants to keep up to date versions of the final results on display, this means that to minimize script reexecution time, every intermediate result must be saved. This generates an obvious conflict with storage requirements. Faced with this, one gives up and accepts the difference between compiled and uncompiled forms of the data, as well as script reevaluation time.

## 3. An Implementation

This methodology is being explicitly applied in the design of the user interface to Gem, to our geometric modeling program at NYIT. This program sports the usual socially acceptable features such as on-screen menuing, overlapping windows (that support vector data), icons and on-line help, as well as some more unique features such as run-time loading of compiled object code and a rich symbol table structure. Modeling operations include polygon digitizing, translational and rotational sweeping, mirroring, stellation, truncation, boolean set operations, offsetting and quadratic, bicubic and geodesic subdivision [9, 10]. Output databases are produced for our animation and rendering software.

The modeler is extensible at two different levels. The scripting facility described below provides a means for ordinary users to write new functions in the command language of the modeler. The menu structure is dynamically modifiable, so these functions can be inserted at appropriate places or just executed from the keyboard. The run-time loader and symbol table structure are used by developers and application programmers to extend the program in the base programming language. After a run-time package has been loaded in, the program appears for all intents and purposes as it would have if the package had been linked in by the system maintainers before releasing it.

Scripts are represented both externally and internally as text. Execution is simple, given the i/o redirection facilities of Unix [3]. The problem of turning to the user for direct input has been solved by tagging the data rather than calling a special routine. (In fact there is no explicit syntax in the input language for procedure calls.) When interactive input is desired in a script, a special token is inserted which is understood by all the data collecting routines to mean: "ignore this token, push the current input channel on the stack and use the interactive input channel." (This is similar to the method used by the Unix command interpreter [11], which, alas, also does not support subroutines.)

Script generation is accomplished by sending strings to a script from a small number of places in the program. The names of functions are emitted just prior to invocation. When functions need arguments (integers, strings, parts, etc.) they call 'get' functions which prompt for and return a variable of the appropriate type. These functions send the argument values obtained to the script just before returning. This scheme limits the number of places

in the program that have to know about script generation to one per data type.

Because of Gem's lack of a formal grammar to define the input language, a practical problem in the creation of scripts is the difficulty of determining where a macro ends. When the *define-macro* command is given, the translation of events into script is enabled. When this command occurs in a script, commands are not executed during recording, but consumed by the define-macro command until it sees the *end-macro* command. In this mode, Gem can be confused if the end-macro command appears inside the macro. When the input is interactive, commands are executed during recording. This method does not become confused since the commands consume their arguments.

One of Gem's user interface features which complicates script generation is its use of cancelable commands. Each of the get functions provides a 'cancel' option which causes the calling function to abort. To handle this correctly, the generation of script must be deferred until commands execute to completion. Presently, Gem makes no attempt to handle this problem.

Another interesting issue in script generation is how to handle modelessness. Gem makes an attempt to be as modeless as possible. In the get functions, any input which isn't of the type needed is passed through to a recursive invocation of the command interpreter. For example, while selecting a part, the user can move the camera, make new parts, change the windows around, access the on-line help database and so on. This is one case in which the lack of a formal grammar actually makes the script generation easier: Gem simply prints script commands as they are evaluated. For example:

```
intersect-parts part1 move-camera part2
```

In contrast, if the script format were Lisp, for example, some means would have to be found of splicing in the extra commands:

```
(intersect-parts part1 (progn (move-camera) part2))
```

The conversion of tablet and dial events to script is dependent on context. For example, in several contexts tablet picks are used to select a current element (solid, surface, polygon, edge, vertex, etc.). In these contexts, the tablet pick is considered to be an alias for the appropriate 'select' command, so the script result is the name of the command plus the name of the object selected. Tablet and dial movements, the really high bandwidth events, are dealt with by considering them to be inputs to fancy 'get' functions. A common use for these events, for example, is in the construction of positioning matrices. When the function which collects these events into a matrix returns, it emits an appropriate script representation just like the other 'get' functions.

Two approaches to script editing have been developed. The first is the compromise adopted by Emacs and Em: dump the script into a file, invoke a text editor on the file and read the file back in. This was implemented as a first cut because it was easy. It also has the advantage of being editor-independent.

The more ambitious script editing scheme, still under development, is to open a text editor as a concurrent process. The editor provides two windows, one an interactive channel to the normal terminal i/o of the modeler process, and the other a buffer for the script to be edited. Script emitted from the modeler is sent to the editor's script buffer and inserted at the current cursor location of that buffer. Hence, the commands of the text editor are available in the script window and the commands of the modeler are available in the other. The drawbacks of this approach are that all the terminal data for the modeler must go through the editor (a performance issue), and, more importantly, only one editor (namely Emacs) can support this mode of operation. If the window system were below the process level, things would be much easier, but unfortunately, our window system is part of the modeler process.

## 4. Discussion and Conclusions

Two major improvements are planned for the Gem user interface. The first is to reimplement the upper level of the interface in Lisp [12]. This would eliminate many of the problems that arise because of the lack of a formal grammar. The interactive command interface would stay the same, but scripting would be done in Lisp. This has only recently become practical due to the development of a programming environment [14] which supports both C and Lisp, and guarantees that the conversion can be done incrementally, without discarding the several tens of thousands of lines of existing modeling code.

The second improvement being considered is a provision for user-configurable device interaction via function networks [24]. A function network package has been implemented and tested as a run-time package but has not yet been bound into the program. The use of function networks at the modeling level as well would provide a means of representing the dataflow aspects of a model explicitly. This would also make the storage of intermediate results possible, which, as mentioned earlier, opens the door to the optimization of script execution time, providing one is prepared to pay the price in storage requirements.

The objections raised earlier to text as a script representation were based in part on the limited intelligence of current text editors. This objection could possibly be eliminated through the use of syntax-directed editing, a technique which has been under development by the programming language community for some time. The script form would be text, but the text editor would know the syntax of the language so its elements could be handled at a reasonable level.

As conjectured in the introduction, the implemen-

tation of a user interface kernel which provides integrated support for scripting and interaction has proven difficult, but not impossible. The work done so far has begun to provide the insights necessary for the implementation of a truly useful tool for animation production, and the methodology developed seems applicable to a wide range of data editing problems.

## 5. Addendum: Other Potential Applications

In addition to modeling and animation for computer graphics, another application area that stands to benefit greatly from the integration of scripting and interaction is music production. Composers should be able to create a score in some reasonable frequency content versus time notation (using either traditional staves or some alternative notation [18]), play the score through some instrument, play the instrument to generate a score, and finally edit the score using either instrumental or scripted input. In fact, a first cut at a system with these capabilities can be assembled with current technology [19, 20, 21, 22].

Another area that uses hybrid scripting/interaction techniques already is robot programming. Unimation Puma series robots [23], for example, come with a programming language for scripting and a manual control box for interactively positioning the robot in key configurations. Once a configuration is set, a line of text can be inserted in the program buffer with a press of a button on the manual control. Such a programming environment could be much more generally useful if fleshed out with the remainder of the capabilities described in this paper.

## Acknowledgements

Pat Hanrahan started the Gem project, is directly responsible for most of the geometric functionality, and directly or indirectly responsible for large parts of the user interface. Many others have contributed ideas and/or code to the Gem user interface, among them Dan Hopen, Robert McDermott, Peter Oppenheimer, Michael O'Rourke, Jean-Louis Schulmann, Jacques Stroweis and David Sturman. Em was written by Pat Hanrahan and David Sturman. I thank John Lewis for much discussion, goading, critical reading and the C/Lisp environment, and Kurt Fleischer and Steve Rubin of Schlumberger Palo Alto Research for general discussions on scripting vs. interaction and dataflow techniques in modeling.

## References

1. *Emacs: The Extensible, Customizable, Self-Documenting Display Editor*, Richard M. Stallman, MIT AIM-519A, March 1981.
2. *UniPress Emacs Screen Editor Manual*, UniPress Software, Edison, NJ.
3. *The UNIX Time Sharing System*, Dennis M. Ritchie and Ken Thompson, Comm. ACM, July, 1974 (republished Jan 1983).
4. *The C Programming Language*, Kernighan and Ritchie, Prentice Hall, 1978.
5. *Nroff/Troff User's Manual*, J. F. Ossana, Unix Programmer's Manual, Volume II.
6. *Yacc: Yet Another Compiler Compiler*, Steven C. Johnson, Unix Programmer's Manual, Volume II.
7. *Mint User's Manual*, David J. Sturman, NYIT internal documentation, September 1985.
8. *Interactive Animation of Parametric Models*, Pat Hanrahan and David Sturman, Siggraph Tutorial #9: In-
9. *A Subdivision Algorithm for Smoothing Down Irregular Shaped Polyhedrons*, D. W. H. Doo, Proc. Interactive Techniques in Computer Aided Design, 1978.
10. *Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes*, Catmull and Clark, Computer Aided Design, Nov. 1978.
11. *An Introduction to the C Shell*, William Joy, 4.2BSD Unix Programmer's Manual, Volume IIc.
12. *Lisp*, Winston and Horn, Addison Wesley, 1981.
13. *The Franz Lisp Manual*, John K. Foderaro, Keith L. Sklower and Kevin Layer, 4.2BSD Unix Programmer's Manual, Volume IIc.
14. *Zlisp*, John Lewis, NYIT internal documentation, March 1985.
15. *Motion Simulation*, Richard V Lundin, Proc. Nicograph-84.
16. *Gem User's Manual*, John F. Schlag, NYIT internal documentation (in preparation).
17. *Inside Macintosh*, Apple Computer Corp.
18. *MusicWorks*, Hayden Software, Lowell, MA.
19. *Yamaha Digital Programmable Algorithm Synthesizer Operation Manual*, Buena Park, Yamaha International Corp., 1984.
20. *DX7*, Yasuhiko Fukuda, Amsco Publications, London, England, 1985.
21. *Synclavier II Instruction Manual*, New England Digital Corp.
22. *MIDI Specification 1.0*, International MIDI Assoc., N. Hollywood, CA, 1983.
23. *Unimate Puma Mark-II Robot 500 Series Equipment and Programming Manual*, Unimation, Inc., Danbury CT, April 1984.
24. *PS300 Programmer's Reference*, Evans and Sutherland Computer Corp., Salt Lake City, Utah.

---

[1]Unix is a trademark of AT&T Bell Laboratories. Scribe is a trademark of Unilogic, Inc. Macintosh is *not* a trademark of Apple Computer, Inc.