MULTICS TECHNICAL BULLETIN

MTB #175

To: MTB Distribution

From: C. D. Tavares

Subject: Proposal for a Graphic Editor

JUSTIFICATION

At present, creation of graphic structures for use with the Multics Graphic System must be performed by coding PL/I procedures which create, edit, and display these structures on an individual basis. The program must be re-edited and recompiled to alter the structure created. This is especially teuious while picture descriptions are still in the debugging stage. It is unreasonable to expect users of the graphic system to code specialized routines to create graphic structures every time a new structure is desired.

PRECEDENTS

Users of the Version 1 Graphic System had available to them an Author-Maintained program, pix_edit, which functioned as an interactive picture editor. With it, users could enter picture descriptions, view the results immediately, and perform limited alterations of their pictures. As pix_edit was not designed to be a generalized editor, it lacked all but the most rudimentary means of altering picture elements (i.e. retyping the entire subconstruct.) Users found that it was usually easier to use a text editor to place the description into a file, call pix_edit to parse and display the construct, and re-enter the editor to make alterations. The author of pix_edit (Ken Pogran) later proposed a graphic editor with extended features in an RFC. The extended editor was never implemented.

PROPOSAL

The attached documentation describes a graphic editor very much like that proposed in the RFC mentioned. Because of improved structure editing capabilities in the Version 2 graphic_manipulator_, it incorporates several new features which were not possible to perform using the Version 1 gsm_ package. The functionality provided by this interactive tool would be invaluable to both the casual user of graphics and to the implementor of extensive graphics applications.

Comments and suggestions may be mailed to Tavares.Multics on System M (Phoenix).

Command
Administrative/User Ring
12/30/74

Name: graphic_editor, ge

The graphic_editor is an interactive tool which may be used to create and edit graphic structures. It is capable of storing these structures into, and retrieving them from, permanent graphic segments (PGS's).

Usage

graphic_editor [seg1] [seg2] ... [segn]

1) segi (optional) is a pathname specifying a segment to be read into the graphic editor. This segment may contain a list of editor commands or assignments, in the same format as they might have been typed into the editor interactively. The segments will be interpreted by the editor in the order specified.

If any errors occur while reading any segment specified on the command line, processing of that file will cease.

When graphic_editor is ready to receive input from the user's terminal, it replies with "Edit.". The user may then begin to issue requests.

Requests fall into two categories: commands and assignments. In general, commands may be terminated with either a semicolon (";") or a newline. Assignments (due to their ability to be quite lengthy) may be terminated only with a semicolon. Sometimes one of more of the arguments of a command may be an assignment. In these cases, only the semicolon is accepted as a terminator.

Comments which are enclosed by "/* ... */" may be interspersed with any input lines.

Symbols

Symbols in the graphic_editor are alphanumeric representations of node values. A node number is a "receipt" which the graphic system returns whenever it is asked to create some graphic element. (For a more complete description of node values, refer to Section 1 of the Graphics Users' Supplement.) Symbols have a value which consists of exactly one such node value.

Symbols may be divided into three classes: the system symbol, which is predefined and represents a primitive operation or element; the user symbol, which is defined by the user at some time with an assignment; and the macro, which is defined by the user, but takes "arguments", and has no permanent value of its own.

System symbols have no permanent value. They take one or more arguments, either implied or explicit. The use of a system symbol represents a request that a new element be created. The node value returned from that creation is then used in any subsequent operation of that particular expression.

Examples of system symbol expressions are:

| | |
|---|---|
| vector 12 14 | a vector of length (12, 14, 0) |
| "Axolotl" uc | a text string containing the string "Axolotl", aligned by the upper center edge. |
| array (a,b,c) | an array containing the nodes represented by user symbols a, b, and c. (See Tuples, below.) |
| lin dotted | A mode element for dotted lines. |

A list of system symbols and descriptions of their use may be found at the end of the document.

User symbols may be up to 32 characters in length, and may consist of any combination of upper-case and lower-case alphabetics, numerals, and the underscore ("_"), provided that the first character is non-numeric. System symbols and commands are considered "reserved words", and may not also be used as user symbols. Attempts to define commands as symbols will result in ill-formed execution of those commands.

Examples of user symbols are:

foo
Front_porch
bolt_23w9

User symbols are stored in the graphic symbol table of the working graphic segment (WGS). They are transferred to and from PGS's whenever the "save", "use", "put", and "get" system commands are used. (For a more complete explanation of graphic symbols, see Section 1 of the Graphics Users' Supplement.)

Macros are user symbols which take arguments like system symbols. Whenever a macro expression is evaluated, the arguments supplied are substituted for the dummy arguments with which the macro was defined. Macros must be defined by macro assignments. For example:

```
macro box x y = vec x 0, vec 0 y, vec -x 0, vec 0 -y;
```

defines a macro named "box" with dummy arguments "x" and "y". The reference:

```
box 10 30
```

represents a rectangle 10 units in x and 30 units in y, and is exactly equivalent to the expression:

```
vec 10 0, vec 0 30, vec -10 0, vec 0 -30
```

Macro names are stored in the graphic symbol table of the WGS, and may be transferred to and from PGS's with the "save", "use", "put", and "get" commands.

## Tuples

A tuple is simply a group of one or more values. Every complete symbol (i.e. a user symbol, or a macro or system symbol with its arguments) is a tuple in itself (a one-tuple). A tuple of more than one element may be expressed as its elements separated by commas, e.g.:

```
a, b, b, vec 10 4 3, intensity 1, xxx
```

This is a tuple of 6 elements.

A tuple which has more than one element represents more than one graphic entity. Therefore, it cannot have one node value. To convert a tuple to a single graphic entity, two system symbols are available: array, and list. These two "functions" gather the elements of the tuple into a graphic array, or a graphic list (respectively). (For a more complete explanation of graphic arrays and lists, see Section 1 of the Graphics Users' Supplement.) The creation of this array or list produces a node value, which may be assigned to a user symbol, or may be used without assignment in some larger expression. For example:

```
one_array = array (a, b, c, d, b):
```

is an assignment which creates a graphic array with the elements (a, b, c, d, and b), and assigns to "one_array" the value of this list.

## Assignments

An assignment is an operation which extracts the value of one tuple and assigns it to another tuple. The assignment operator is the infix "=" sign.

The simple assignment:

```
foo = bar;
```

specifies that the the value of "foo" is to become the symbol "bar". An important point to keep in mind is that this does not mean that "foo" and "bar" both refer to the identical piece of graphic structure. Rather, "foo" contains "bar", and (of course) indirectly also contains the entire structure contained by "bar". (It is possible to assign the value of a symbol to another symbol, rather than assigning one symbol to another; this operation will be discussed in the section describing qualified expressions.) If "foo" is undefined at the time of assignment, it will be created. If it had a previous value, that value will be replaced. Any other graphic structures which referenced "foo" will still refer to it, but will now contain (indirectly) its new value.

In general, only tuples of like dimensionality (i.e. having the same number of elements) may be assigned to each other. For example:

```
a, b, c = d, e, f;
x = array (p, q, r);
```

are both valid assignments. However,

```
one, two = three, four, five;
```

is not a valid assignment.

Two exceptions exist to this rule: First, if the object to the right of the assignment operator is a one-tuple, it may always be "promoted" into the dimensionalty of the object to the left of the assignment operator. For example:

```
a, b, c = d;
```

is equivalent to

```
a = d; b = d; c = d;
```

The second exception is that if the object to the left of the assignment operator is a one-tuple, and the object to the right of the assignment operator is not a one-tuple, then the "array" operator is assumed. For instance, the assignments:

```
a = b, c, d;
a = array (b, c, d);
```

are equivalent. Note that the promotion facility and the implicit-array operator can never be used simultaneously. This feature disallows statements such as:

```
one, two = three, four, five;
```

which more probably represents a user error than a useful statement.

Assignments also have values. The value of an assignment is the value of the tuple into which the assignment is done. For example, the value of

```
foo = bar;
```

is the new value of "foo". This feature allows nested assignments, as in the following example:

```
pic = some_setpos, (line = vector 100);
```

This is equivalent to:

```
line = vector 100;
pic = some_setpos, line;
```

Note the use of the parentheses for precedence definition. The parentheses in the expression are necessary since tuple formation is a "stronger" operation than assignment. If the expression had been written as:

```
pic = some_setpos, line = vector 100;
```

it would have been performed as the operations:

```
some_setpos, line = vector 100;          /* a promotion */
pic = some_setpos, line;            /* an implicit array */
```

## Qualified Expressions

It is possible to refer to any element (or tuple of elements) of a symbol which represents an array or list by the use of qualified expression. The simplest qualified expression consists of a symbol, followed by a period. This represents "the value of". In our first example,

```
foo = bar;
```

we assigned "bar" as the value of "foo". The relationship of "foo" to "bar" was a superior/inferior, or father/son relationship. If, instead, we say

```
foo = bar.;
```

we are assigning the <u>value</u> of "bar" to "foo". This makes both "foo" and "bar" refer to the identical piece of graphic structure. The symbols now have a "brother" relationship.

Successive trailing periods denote further levels of evaluation. Assume the following assignments:

```
box = vec 10, vec 0 10, vec -10, vec 0 -10;
a = b = c = d = box;
```

The following relations hold on these symbols: (Read "≡" as "is equivalent to")

```
a. ≡ b
a.. ≡ b. ≡ c
a... ≡ b.. ≡ c. ≡ d
a.... ≡ b... ≡ c.. ≡ d. = box
```

The assignment

```
a... = null;
```

actually assigns "null" to "d".

Additional types of qualified expressions make it possible to refer to elements of lists. The element desired is denoted by an integer following the appropriate levels of qualification. For example,

```
box.2
```

is the second element of "box" (vector 0 10). Tuples of contiguous elements may be specified by using a range expression, which consists of two integers (representing the first and last element desired) separated by a colon (":"). For example,

```
bottomless_box = array (box.2:4);
```

will create a symbol which contains an array made up of all elements of "box" except the first.

The star ("*") has a special meaning in a qualified expression. If used by itself, e.g. "box.*", it refers to a tuple made up of all the element of "box". It may also be used as the last part of a range expression, e.g. "box.2:*", which refers to a tuple made up of all the elements of "box" from the second to the last. The assignment

```
bottomless_box = array (box.2:*)
```

is equivalent to the example above. Note that if a star occurrs in a qualified expression, it must be the <u>last</u> character. It may neither be followed by the second component of a range expression (e.g. "box.*:3") nor by further levels of qualification (e.g. "box.*.1").

Because a user may not always know exactly how many levels of symbol indirection exist between the symbol name he is working with and the arrays or lists with which he desires to work, any reference to an element (or range of elements) of a list found in a qualified expression will cause the evaluator to skip any number of levels of symbol indirection. Using one of our previous examples to elucidate, this means that

```
a.1 ≡ a.....1 ≡ box.1
```

This frees the user of typing in long, and possibly inaccurate, strings of periods; but allows the user who wants to maintain fine control of his indirect symbol structuring to do precisely that.

Certain qualified expressions may have different meanings on the left side of an assignment than they do on the right side. This is particularly important to note when using nested assignments. In particular, qualified expressions which evaluate to an element of an array or list, or to a tuple of such elements, have different meanings in these two contexts. If such an expression occurs on the right side of an assignment, its value consists of references to the values of the elements which make up the list. A previous example ("bottomless_box") showed how this usage is interpreted. On the left side of the assignment, however, the expression denotes element replacement. For instance, assume the following assignments:

```
box = vec 10, vec 0 10, vec -10, vec 0 -10;
elem = box.3;
box.3 = shift -10;
```

The first assignment defines "box". The second assignment causes "elem" to refer to the <u>same piece</u> of graphic structure which is the third element of box. The third assignment changes the "top of the box" from a visible vector to an invisible shift by redefining the third element of "box" to be a shift of equal magnitude. This does <u>not</u> change the value of "elem". It simply breaks the association between the list "box" and the construct which was its third element. If the actual changing of that construct were desired, the third assignment of the above example could be replaced with

```
box.3. = shift -10;
```

This assignment would in fact change the value of elem. A side-effect of this property is that the expressions "symbol.<u>n</u>" and "symbol.<u>n</u>." are equivalent on the right side of an assignment, but are not equivalent on the left side.

## Node Constants

It is possible for node values to exist in the WGS without being assigned to any symbol. For instance, a user program could be called from inside the editor to construct a particularly intricate "canned" graphic structure which may be inefficient or difficult to construct by hand. The program could print the number of the top-level node in the structure, so that the user could "pick it up" by assigning a name to it. The number of this node may be typed in, preceded by the character "#". This is a "node constant".

For example: if the node constant "#12345" appears as such an output, and it is wished to assign to this node the name "orphan", the assignment:

```
orphan = #12345;
```

may be used.

Octal node values may be expressed directly as node constants without user conversion by immediately following the "#" with the lowercase letter "o", e.g. "#o144" is equivalent to "#100".

Although node constants and qualified expressions based on node constants are allowed on the left-hand side of assignment statements, their use is strongly discouraged.

## Commands

Following is a list of editor commands. Arguments enclosed in angle backets ("< ... >") denote necessary arguments. Arguments enclosed in square brackets ("[ ... ]") denote optional arguments. Each command whose argument is signified by <exprn> will accept single elements, tuples, assignments, or any combination of these as its argument. For example:

```
display pic = array (house, street, parked_cars);
```

serves the dual purpose of defining "pic" and displaying it.

```
>--->     display <exprn>
          di <exprn>
```

causes the screen to be erased and the graphic structure specified to be displayed. If the argument is a tuple, no erase is performed between each element of the tuple.

```
>--->     list [options]
          ls [options]
```

will list selected symbol tables. Any number of options may be specified. The following options are allowed:

-commands -com list the editor commands and their abbreviations.

-system -sys list the available system symbols and their abbreviations.

-macros -mc list the defined macros.

-symbols -sym list the user symbols.

-all -a list all of the above.

If no options are given, "-symbols" is assumed.

```
>--->     execute <command_line>
          exec <command_line>
```

causes the <command_line> to be passed to the command processor.

>---> show <exprn>
causes an abbreviated description of the tuple <exprn> to be printed to the user's terminal. If the value represents a terminal graphic element, its contents will be printed. If it represents a non-terminal element, it will be described and the number of its elements given.

>---> replay <exprn>
like show, except that the entire graphic subtree inferior to the chosen node is described in assignment notation, along with nested assignments where appropriate. This command allows a user to "replay" a graphic structure in a form acceptable as input to the graphic_editor.

>---> remove <symbol1> [symbol2] ... [symbol<u>n</u>]
causes those elements named to be removed from the table of known user symbols. The symbol in the WGS is also deleted, and all references to it will be transformed into direct references to whatever contents it possessed.

>---> use [pathname]
causes the permanent graphic segment (PGS) specified by [pathname] to be loaded into the WGS. This allows the editor to use a previously-constructed set of graphic structures. If [pathname] is not supplied, graphic_editor will use the pathname which was last supplied to a "use" or "save" command. If no such pathname exists, an error will occur. If an error occurs during the execution of a "use" command, the "last pathname" will be deliberately forgotten.

>---> save [pathname]
causes the contents of the WGS to be saved in a PGS specified by [pathname]. If [pathname] is not supplied, graphic_editor will use the pathname which was last supplied to a "use" or "save" command. If no such pathname exists, an error will occur. If an error occurs during the execution of a "save" command, the "last pathname" will be deliberately forgotten.

>---> get [mode] [(pathname)] <sym1> [sym2] ... [sym<u>n</u>]
gets the structures <sym1> ... [sym<u>n</u>] from the PGS specified by [(pathname)]. (This notation means that "pathname", if it is given, must be within parentheses.) The [mode] argument determines what action is taken on attempts to redefine an existing name:

-safe leave the old symbol as is and print an error message.

-force redefine the symbol and all subsidiary symbols.

-replace_only
-rpo redefine the symbol. If subsidiary symbols are duplicated in the WGS, use the copies in the WGS. For any subsidiary symbols not so duplicated, create null (empty) symbols.

-replace_all
-rpa redefine the symbol. If subsidiary symbols are duplicated in the WGS, use the copies in the WGS. For any subsidiary symbols which do not exist in the WGS, use the ones in the PGS.

If [mode] is not specified, "-safe" will be assumed. The [mode] and [(pathname)] arguments, if present, may occur in either order, but must precede any symbol names.

>---> put [mode] [(pathname)] <sym1> [sym2] ... [symn]
stores the structures <sym1> ... [symn] into the PGS specified by [(pathname)]. The [mode] argument determines what action is taken on attempts to redefine an existing name:

-safe leave the old symbol as is and print an error message.

-force redefine the symbol and all subsidiary symbols.

-replace_only
-rpo redefine the symbol. If subsidiary symbols are duplicated in the PGS, use the copies in the PGS. For any subsidiary symbols not so duplicated, create null (empty) symbols.

-replace_all
-rpa redefine the symbol. If subsidiary symbols are duplicated in the PGS, use the copies in the PGS. For any subsidiary symbols which do not exist in the PGS, use the ones in the WGS.

If [mode] is not specified, "-safe" will be assumed. The permissible order of the arguments is the same as for "get".

>---> read <pathname>
causes the file specified by <pathname> to be interpreted as a set of editor commands. Any "read" command encountered in a file will switch the input source to the specified file. When the

commands in the specified file have been exhausted, control will return to the user's terminal, or to the original file issuing the "read". Errors encountered while reading from a segment will cause control to be immediately returned to the user's terminal.

>---> quit

is used to exit from the editor.

>---> restart

will re-initialize the editor, the working graphic segment, and all associated symbol tables. Any remaining command line, as well as any file "reads" pending, will be flushed without execution. The state of the editor after a "restart" is the same as the state of the editor when it is first invoked.

>---> help
?

directs the user to relevant documentation.

>---> macro <name> [arg1] ... [argn] = <exprn>
macro show <name1> ... [namen]
macro replay <name1> ... [namen]

The first form defines a macro with name <name>, and arguments [arg1] ... [argn]. The other forms do for macros what "show" and "replay" do for symbols.

>---> input <symbol> [device_name]

requests that a "what" input be requested from device [device_name]. The input will be collected, interpreted, made into a graphic structure, and assigned to symbol <symbol>. This feature is not yet implemented.

## Defined System Symbols

### Positional Elements

All positional elements take arguments of the form "x y z". If any of these arguments are not supplied, it will be assumed to be zero. It is possible to supply no arguments, only "x", only "x y", or all of "x y z". No other combinations (e.g. "x z") are parsable.

| | | |
|---|---|---|
| >---> | setposition | (sps) |
| | setpoint | (spt) |
| | vector | (vec) |
| | shift | (sft) |
| | point | (pnt) |

## Modal Elements

>---> intensity (int)

Argument: Integer, 0 through 7, or "off" (0), "on" (7), or "full" (7).

>---> linetype (lin)

Argument: Integer, 1 through 5, or:

| | |
|---|---|
| "solid" | (1) |
| "dashed" | (2) |
| "dotted" | (3) |
| "dash_dotted" | (4) |
| "long_dashed" | (5) |

>---> blink (blk)

Arguments may be any from the following correspondence list:

| | |
|---|---|
| "steady" | 0 |
| "blinking" | 1 |

>---> sensitivity (sns)

Arguments may be any from the following correspondence list:

| | |
|---|---|
| "insensitive" | 0 |
| "sensitive" | 1 |

## Mapping Elements

>---> rotation (rot)

Arguments: "x_rotation y_rotation z_rotation" in floating or integer degrees.

>---> scaling (scl)

Arguments: "x_scale y_scale z_scale" in integer or floating notation.

## Miscellaneous Elements

>---> null

No arguments. This element represents the "zero node". It is a placeholder, or a graphic no-op.

>---> text "string" [position]

"string" [position]

The second form of the text string is implicitly understood. The optional argument [position] specifies the string alignment. (For a more complete explanation of string alignments, refer to Section 1 of the Graphics Users' Supplement.) Any character may appear within the string. If it is desired for a quote to appear as part of the string, it may be doubled, as in PL/I. The argument may be either an integer or a string, from the following correspondence list:

| | | |
|---|---|---|
| upper_left | ul | 1 |
| upper_center | uc | 2 |
| upper_right | ur | 3 |
| left | l | 4 |
| center | c | 5 |
| right | r | 6 |
| lower_left | ll | 7 |
| lower_center | lc | 8 |
| lower_right | lr | 9 |

>---> datablock <element>
data <element>

creates a datablock containing the element <element>. This element may be of a form acceptable as a symbol name, or numeric, or a string enclosed in quotes. It may not be a break character (";", ",", etc.) unless enclosed in quotes. Datablocks may be used to hold information relevant to the structure, within the structure itself. (For a more complete explanation of datablocks, refer to Section 1 of the Graphics Users' Supplement.)

Note: No dynamic operations are presently defined for the graphic_editor.

 (END)