To: Distribution

From: C. D. Tavares

Date: 09/01/81

Subject: Directions for Multics Graphics

Introduction

This MTB serves three purposes:

1) To compare the Multics Graphics System (MGS) with other existing graphics systems;
2) To document the rationales behind major design features of the MGS which seem to be in conflict with aproaches taken by other systems, where the rationales exist; and
3) To identify a course of improvement to make the MGS more attractive to users and as much in line with current graphics technology as possible.

I have attempted to write this MTB in a format which will be understood by readers with little to no knowledge of computer graphics or the Multics Graphics System. I have done this not only to afford those readers with a free informal mini-course on computer graphics, but because I hope as many Multicians as possible will take the time to read and understand the MTB, in the hope that many of them will make additional or better suggestions on how to improve the product.

Comments may be sent via any of the following methods:

Multics mail: Tavares.Multics on System M (MIT only in a pinch)

Continuum: >udd>m>cdt>mtgs>Graphics (mgs)

US mail: C. D. Tavares
Honeywell (HED VA20-601)
7900 Westpark Drive
McLean, VA 22102

Overview

The Multics Graphics System (more properly, Version II of the MGS) was designed and implemented in 1973/1974. Since then, what MGS development and maintenance there has been has carefully stayed within the original design concepts and framework set up in 1973, when we attempted to design a system responsive to the directions in which we thought the field of computer graphics was heading.

Needless to say, the field of computer graphics has undergone quite an evolution since that time. Still, by and large, the current MGS design has served for the past seven years as a reasonable (and in some areas, still quite advanced) method of performing computer graphics. Now however, forces in both the technical arena and the marketplace make it necessary for us to extend, expand, or redesign the MGS to retain its competitiveness.

Some of the aforementioned forces are:

- The rise of raster-scan graphics technology. In 1973, raster devices (displays which work on the same principle as television sets— moving the beam in a predefined pattern across the screen, turning the electron beam on and off as appropriate) were considered toys— incapable of "serious" graphics. "Serious" graphics devices (like the Evans and Sutherland LDS-1) were calligraphic (stroke, vector) displays. Today, raster devices (like the Ramtek, Chromatics, Sanders, and a host of "personal" computers capable of various types of graphics) have come into their own right. While they do not possess the capabilities of calligraphic displays as far as detail goes[1] they are significantly less expensive, and they possess some capabilities that calligraphic displays do not— e.g., no noticeable flicker regardless of the number of lines displayed, and the ability to provide colored areas as well as colored lines.

- The proposed ANSI Graphics Standard. Released upon the world about two years ago, it takes a basically different view of graphics than the MGS does. It does not have the capability for distributed intelligent graphics that the MGS has, and has no concept of structured graphics objects as "saveable" entities. It does, however, provide a number of features that we do not, including some that we cannot provide on a conceptually-(as opposed to practically-) equivalent basis under the current design of the MGS.

- The rise of DISSPLA as an industry-standard graphics application-level interface. DISSPLA, marketed by ISSCO Inc., is a machine-independent, terminal-independent graphics system that provides CalComp-style lower-level interfaces and impressive higher-level applications interfaces such as line, bar, and pie charts; contouring; cartography; text presentation (e.g., for slides, overheads) and related features. A separate package, TELL-A-GRAF, is a DISSPLA-oriented package similar to the MGS graphics_editor.

---

[1] calligraphic displays usually start at about 1024x1024 addressible points; most rasters, due to the TV technology involved, rarely exceed 512x480

- The ubiquity of terminal-independent graphics systems. Currently, a roster of terminal-independent graphics systems would include the MGS, the West Point GCS, DISSPLA, several ANSI implementations such as DIGRAF and TIGS, and foreign standards promulgated by ANSI analogues, such as GINO-F in the UK. Although the MGS (along with Multics itself) was a pioneer in the field of terminal-independence, it is hardly any longer unique. As a matter of fact, with the exception of the MGS (and possibly TIGS), all packages in the above list are not only terminal-independent but machine-independent; written in a transportable language such as FORTRAN, with minor adaptations necessary to accommodate it to the I/O system of the host. In such a marketplace, the host-dependent MGS must show clear advantages to a customer if he is to choose it over one of these other packages— yet, it must at the same time be able to provide a large degree of compatibility with these other systems in case such is needed. This compatibility can range, for example, from the low end of being able to communicate with a device that implements the ANSI standard, to the high end of being able to offer similar sophisticated graphics applications.

Basic Conceptual Disparities

Despite all the other existing graphics packages on the market, the MGS still has basic capabilities which are addressed by no other major graphics system. It also lacks (or provides conceptually-different forms of) capabilities common to many other graphics systems. Primary examples of both are the ability to define structured graphic objects, the lack of a window/viewport orientation per se, and the ability to control highly-interactive, distributed graphics applications using intelligent terminals.

STRUCTURED GRAPHIC OBJECTS

Before the Multics Graphics System, a typical commercial graphics system usually consisted of a set of subroutines which drew pictures directly— put the pen up or down, moved it to such-and-such a point, and so on. Later, these were conceptualized into "shifts, vectors, and points", but the basic mode of operation was the same— direct drawing of a picture. Whether the picture was drawn in real-time or the commands were put onto tape to be brought to a plotter and drawn later is of little consequence— the fact is, once the picture was drawn, it was drawn. If it needed fixing, you had to edit, recompile, and rerun the program(s) that generated it; if it needed saving, you had to freeze a copy of the program and all its subroutines. (I tend to refer to this as the "CalComp method", for historical reasons.)

Under the Multics Graphics System, the user assembles a graphic object, as opposed to assembling a display or a picture. He invokes entrypoints to create primitive elements (such as lines, points, and text strings). He then assembles these elements in any way he chooses— he can assemble them in any order, use any of the elements multiple times, or even not use one or more of the elements at all, as he chooses. The result is, for all

practical purposes, another "single" element, which can be used as a building block of larger structures, and so on.

In addition to objects which can be drawn and seen ("graphic effectors"), the user can also create other primitive objects which modify the appearance of graphic effectors: "modal effectors", or "modes", which control things like color, visibility, dottedness of lines, blink, etc.; and "mapping effectors", or "mappings", which control things like orientation (rotation), size (scaling), and extents (clipping, masking). These can be applied to any object without modifying the object itself. For example, if the user has a solid object and wishes to create a dotted version of it, he creates a two-element list— the first element of which is a "dotted mode" effector, and the second of which is a reference to the original object. The object itself is still solid— but the higher-level object represented by the new list is a dotted representation of the original object. The original object is not copied, but shared.

Presumably, at some point the user stops building his object and displays it. At this time, and at this time only, it is interpreted as a given, ordered sequence of items to be drawn; it is only then that the terminal is told what instantiations of what effectors fall under the influence of which modes and mappings; in fact, it is only then that we are allowed to know where any given effector falls on the screen, and then only for the duration of that single display operation. (A more thorough discussion of this topic occurs in the section entitled "Image Space and Object Space".)

Later, the user may use the editing primitives to reach inside the object and change or replace pieces, perform restructuring, and so on, as a result of the visual feedback he gets from the display operation ("does it look like what I wanted?"). He need not re-create the "correct" portions of his object. And, if his terminal is sophisticated enough to have some local processing power, the MGS assists it in keeping a local copy of the graphic object which can be edited <u>in place</u> either locally or by the program running in Multics, with low data-transmission requirements— a true "distributed graphics database". The user may also save objects in their structured form for later editing, redisplay, or use as components in still-higher-level objects.

The concept of structured graphic objects is central to the MGS, and several of our customers have expressed the requirement that any future implementation of MGS retain this feature.

THE ANSI CORE SYSTEM

The proposed ANSI standard handles the structuring of graphic information in a manner which I personally consider to be the worst of both worlds. The user opens a graphic "segment", which is best described as a conceptual container for a graphic object. The object is then created in strictly sequential form, similar to the CalComp method. From time to time, the user might call subroutines that will set up (or change existing) modes and mappings, which are automatically applied to subsequent graphic effectors created. Ultimately, the graphic segment is closed by the user, and displayed.

The graphic segment may not be displayed until it is closed. Once the graphic segment is closed, it may not be re-opened, meaning that the graphic object inside may not be edited. No facilities are provided for editing a graphic object in an open segment (the reasoning here probably being: why should your program want to edit an object it is in the process of constructing when it could have just created it correctly to begin with?) The segments are not meant to be saved in files on the host system. All this might be forgiveable if graphic segments could be referenced by other graphic segments; alas, they cannot. Thus, the user of ANSI graphics has on his hands a computerful of uneditable, unshareable graphics segments. Users wishing to do truly dynamic graphics have been forced, in some cases, to create hundreds of segments containing one line each, link them together in various ways, and operate on structures of these, making these graphic segments more or less equivalent to our graphic atoms, and their interconnections equivalent to our graphic lists.

IMAGE SPACE AND OBJECT SPACE

The basic difference between the MGS and other graphics systems lies in how it handles the dichotomy between what is generally called "object space" and "image space". Image-space orientation implies that all graphics is conceptualized and designed with respect to how the image appears on the drawing surface. Philosophies like the "CalComp method" work entirely in image space— the user pays explicit attention to where, on the plotter page, every item appears that he is drawing. Conversely, object-space orientation implies that graphics is conceptualized and designed with respect to collections of "graphic objects" without regard to where they appear on the screen, if at all. We've already touched upon what this means in the MGS. Sooner or later, of course, all graphics must be mapped into image space— but the longer one can stay in object space, the more general are the operations that can be performed. This is because there is a conceptual "binding" that occurs when a graphic object is finally set into image space, along with the usual loss of information that accompanies any binding. (The section below, entitled "Windows and Viewports", adds to the discussion of some of these points.) The other side of the coin is that the amount of work the graphics system must do to maintain this image-independence goes way up; mainly because it must track and transform all the extra information that is thrown away by systems which make the jump into image space earlier.

This last point deserves explanation. Take the following example: since a graphic object can be displayed either by itself, or shared by any number of higher-level graphic objects, or both at the same time, it is impossible to tell at creation or editing time where it will appear on the screen. In fact, if a higher-level structure contains the appropriate clippings, scalings, invisibility modes, or similar effectors, it is not even possible to tell <u>if</u> it will appear on the screen. Generally, the knowledge about position binding is reserved to the virtual terminal itself— meaning either the intelligent terminal or the Multics-resident graphic support procedure (GSP) simulating it. Even after the object has been displayed, the terminal-independent portion of the graphics system is not allowed to assume anything about its position, since the graphic object can be edited in an arbitrary manner while inside the memory of an intelligent terminal.

Other graphics systems deal with the problem in different ways. Some simply fail to deal with it; some restrict the operations their users can request; some convert to image space quickly to lighten the load. None of them have to handle the problems of a hierarchical data structure in which transformations such as rotations, perspectives, clipping, hidden line elimination, and so on, can be done at any level, in any order.

This is not to say that Multics cannot take advantage of certain "image-space" tricks to boost its efficiency. Multics Graphics defines two list-structuring elements: lists and arrays. Both serve to collect and order other graphic elements "below" them, and can serve as elements themselves in other lists and arrays in a classic tree-structure. The difference between lists and arrays is that the graphics system is constrained to retain the structuring of lists even to as far out as the local memory of an intelligent graphics terminal if required, in order to allow the real-time editing, control, and animation of this structure; while it is free to reduce and optimize arrays in any way it pleases, since the user has pledged (by using arrays instead of lists) that he will never reference the subsidiary elements as individual entities after the structure is compiled by the graphic compiler. In this fashion, the present graphic compiler pre-digests all mappings, one mode (invisibility), multiple consecutive shifts, etc. inside arrays, performing a job of compaction and simplification in a central location. Because the properties of the structured representation used by MGS contribute an essential integrity to each unit/level of graphics structure (any object can be used as a primitive, and the integrity of the primitive cannot be affected by operations performed by other pieces of graphic structure that may reference it) these optimizations make use of this integrity: since the state of the object does not depend on how it is referenced, the graphic compiler is free to set up a sort of "temporary image space" in which to make its optimizations. Also, GSP's for non-intelligent devices can request that all lists be turned into arrays at run time, saving themselves the trouble of having to handle complex mapping operations.

WINDOWS AND VIEWPORTS

In "classical" (read, "Newmann and Sproull" or "ANSI") graphics systems, the user has to work within two separate sets of coordinates. "World coordinates" are an arbitrary, non-dimensionalized set of values which the user defines to speak about the items he is constructing. For example, a city transportation planner may want to draw street maps using vectors in units of miles; a machine-tool programmer may want to do his work in "mikes"; a circuit-board designer may want to work in millimeters. It frankly doesn't matter to the graphics system what unit the user thinks he is working in; the only thing that matters is that "2 gruzzles" is twice as big as "1 gruzzle". The origin of world coordinates similarly do not matter— a insurance analyst working in "years" may be drawing in the region of 1960 to 2020. The coordinates need not even be taken from the same domain— the analyst using his x axis to represent years well be using his y axis to represent deaths due to auto accidents. World coordinates are the coordinates of object space.

"Screen coordinates", on the other hand, are important to define the size and extents of the graphic device being used. They are fixed for

different devices and are in some real-world unit such as inches, or some hardware-defined unit such as "points". Screen coordinates are the coordinates of image space.

In these graphics systems, the user constructs his display by first defining a window over his object space: "I'm interested in the years between 1960 and 2020, and the number of deaths between 0 and 10,000". Then he defines a viewport onto his screen: "I want to use the upper right-hand quarter of the screen" (e.g., from (0,0), the center, to (512,512), using the MGS convention). The display operation then maps (1960, 0-deaths) to (0,0) on the screen, and (2020, 10,000-deaths) to (512,512) on the screen. Choosing a different window or viewport will distort the picture differently (use of the word "distortion" presumes that the picture is one which can be thought of as having some "normal" scale in the first place, such as a face).

One advantage of this representation is that the user can use all the absolute graphic effectors he chooses. (An absolute vector, for instance, is one which represents, "Start from here and draw a line to the point (10,56)"; as opposed to a relative vector, which represents, "Start from here and draw a line of length 43 in the x direction and 20 in the y direction".) After all, due to the window/viewport mapping inherent in the display operation, anything "absolutely" positioned in the object space ends up being relative to the window and viewport anyway, and thus relative to the screen.

But certain capabilities are lost in this design. Consider a user who is tasked with drawing a building whose facade contains 100 identical doors. (Obviously windows are indicated here, but so as not to confuse our terminology between glass windows and viewport/windows, let's call them doors.) He can go into "CalComp mode" and write a subroutine that creates all 100 doors (possibly at various rotations— the facade isn't necessarily flat) at all the appropriate locations in the object space— inelegant, hard-coded, and brute-force; besides the fact that doing things like rotations is a job for a graphics system, not a graphics user. On the other hand, the user could create the door once, choose an appropriate window around it, and then display it 100 times through 100 cleverly-chosen viewports— elegant, and completely non-reproduceable the next day, because the window/viewport chosen is a parameter of the display operation, not of the object being displayed. As a result, the user has nothing he can save which represents "the building". Of course, the program itself could be saved, in which case we're back in hard-coded "CalComp mode". Or, the window/viewport selection could be made part of the top-level of the graphic object; but this object could never be used as a part of a higher-level object because its position on the screen is determined forever (i.e., the jump to image space has already been made).

Instead of postulating multiple projections from a object space to a screen system, the current MGS takes the tack that the object space and image space are fixed with respect to one another; the screen acting as a window on some small central portion of a much-larger object space. It is then the job of the user to move any objects he may create into the range of this window by inserting the proper positioning effector (e.g., a setposition or a shift) at the beginning of the object— something that can be done at

object creation time or at any time afterward. He can control its distortion and orientation by inserting scaling and rotation effectors at the beginning as well.

Thus the current user of MGS has about the same practical capabilities as the user of a window/viewport system, with certain tradeoffs. First, his use of absolute effectors is severely curtailed: it is difficult to "pull" a graphic item into view of the screen by inserting shifts at its start, when one or more of its components is securely anchored somewhere out in the object space's left field; or to display a building containing 100 doors when the door insists on appearing only at one place on the screen. The logical result of this philosophy is that the ideal perfectly-manipulable graphic item is made up completely of relative effectors, and in fact has no "home position" in the object space; instead, it floats with respect to an origin which is to be set later (or, if it is not set before the item is displayed, is by convention (0,0), the center of the screen). (The astute reader will have already deduced that the virtual size of the object space has just been made completely moot, aside from its utility in "holding" pieces of large objects which have been clipped off the screen.)

Because of this limitation (and because of symmetry considerations with respect to the capabilities of graphic hardware circa 1973) only two absolute effectors were defined: setposition and setpoint. No absolute vector currently exists, because we have no "magic translation function" to move its endpoints with respect to the screen, as windows and viewports do.

Extensions Necessary to The MGS

Conceptual differences aside, these other graphics systems provide features and capabilities that the MGS does not: hidden lines, colored areas, perspective, clipping, and text handling. Some of these capabilities are (theoretically) simple to implement in the MGS; we simply have never gotten around to it. Others are difficult because our unique framework of structured graphic objects makes it necessary for us to devise methods for their application that are much more general in scope than the algorithms that exist in other graphics systems. Some reviewers have gone so far as to suggest that some of the new capabilities (e.g., perspective) will be impossible to handle in a structured manner. I believe this not to be the case. If the new capabilities are designed in a correct and general way, we should have no problem fitting them into a standard, structured environment.

WINDOWS AND VIEWPORTS, MGS-STYLE

One particularly elegant solution to the problem of allowing multiple, stacked windows and viewports without ever leaving object space is to define a new mapping effector to go with scaling and rotation. This new "reference point" effector will establish the current graphic position (or some point relative to the current graphic position, or some absolute position) as a new "relative origin" until reverted or superseded. When no "refpoint" is active, the default is the actual screen origin. Since refpoint is a mapping

effector, refpoints on different levels will interact benignly with each other— e.g., a refpoint setting the the relative origin to a certain absolute position may actually be choosing what it thinks is an "absolute position" relative to some superior refpoint effector previously encountered in its parent level. Best of all, it frees the programmer to use any and all of the graphic primitives available (including, since we have just made it meaningful, a new absolute vector element), secure in the knowledge that his object is still easily relocatable in object space.

The benefit remains that the configuration of the picture on the display screen is always expressed by the configuration of the graphic object being displayed— that is, that the user has a graphic object which carries its own positioning, distortion, orientation, perspective, and so on along with itself. Thus, each picture a user creates is also an object, which can then be used as a component of a larger object, ad infinitum, without having to recompute or alter its "display parameters".[1] Furthermore, the user, by the simple transmission of this object to a satellite intelligent graphics terminal, has told the terminal everything it needs to know about these attributes of the object to enable the terminal to display the object exactly as the user wishes it displayed— a principal requirement of our distributed graphics database.

HIDDEN-LINE ELIMINATION AND SURFACES

The current MGS does not perform hidden-line elimination because there is no basis for it. In geometric fact, lines cannot be hidden by other lines; and lines are all the current MGS knows how to express. It is clear, however, that users want to speak about surfaces, and have them hide other objects. Aside from hiding properties, surfaces are also necessary to express such concepts as colored areas on the screen.

To allow this, a new surface effector will be implemented. Two choices of primitive are possible: planar surfaces with three points (triangles) or convex planar surfaces with N points (irregular convex polygons). Each has its advantages. Triangles' vertices are always coplanar and convex by definition, and certain hidden-line routines work best when their object space is restricted to triangles. On the other hand, implementing coloring or "patterning" in device drivers is needlessly complex when the input primitives do not communicate cases in which many primitive triangular surfaces are really all part of the same overall surface. Most terminals that can color areas already know how to do almost all of such cases in their hardware. Plotters that would have to cross-hatch areas to express a similar effect would go crazy. If the input primitives do communicate this linkage, then they are simply taking the long way around actually describing the original irregular convex polygon after all. If they don't, some sort of postprocessor is required

---

[1] Those familiar with the proposed ANSI core may note the kludges which had to be introduced into their single-level objects to implement two- and three-point perspective; a conceptually-simple prospect in a system like the MGS which permits layered graphical entities, each layer simply applying another perspective point to its subsidiary objects.

to "re-derive" the lost connections. In point of fact, this has been the subject of much discussion, and the jury is still out on it.

The surface primitive will have no "outline". Its major visible effect will be to hold color and/or to hide other things. If an outline is desired, one may be supplied by surrounding the surface with the appropriate vectors.

Due to the expense of hidden-line elimination, the "opacity" of a surface will be controlled by a mode, whose default will be "transparent" for efficiency. This mode has a non-obvious effect on non-surface objects as well. When this mode is in the opaque state, not only does this enable surfaces to hide things, but it causes other effectors to allow themselves to be hidden. When it is in the transparent state, not only do surfaces no longer hide things, but other effectors will always appear "right through" to the screen surface— even through a later (or earlier) surface that may have been specified as "opaque". In other words, hidden-line processing must be turned on for an entire object before any of the elements of that object can either hide or be hidden.

This is not an unreasonable restriction, since hidden line processing is typically desired at the level of "scenes" (complete displays), and turned on or off on that basis. Although there are certain cases where this feature may be misused to produce an unreasonable display, such uses are practically equivalent to drawing an arbitrary scene on the display before (or after) some other scene which is processed for hidden lines, without an intervening screen erase. In such a case, it is obvious why old (or new) lines have broken through otherwise "hidden" areas.

TEXT PRESENTATION

The days when graphics systems can decline to provide text support are over. Text is an important part of any presentation-quality graphics, and market pressures alone dictate that a graphics system give its users fine control over high-quality text capabilities.

The present MGS allows a user to choose between two treatments of character strings: "text" and "varying text". A text element is a character string that is presented in the terminal's native character set. If the terminal has more than one native character set, one is chosen in a non-negotiable fashion by the GSP. Text appears, like any other printed text, in a horizontal line from left to right, at the current graphic position. It is not affected by mappings— text cannot be rotated, scaled, or what have you. A modicum of control is given the user to specify which corner or edge of the string is aligned with the current graphic position.

Varying text is not a primitive element. The user supplies a string and several additional arguments to an entrypoint of graphic_macros_, which assembles a large collection of vectors and shifts to "draw" the desired string. The user can specify the height and (average) width of the characters, the name of a font (a graphic character table) which describes the drawn representation of the individual letters, and which corner or edge of the

string is aligned with the current graphic position. Since the resulting element is a large collection of vectors and shifts, varying text can be rotated, scaled, mapped, and so on.

The limitations of the text element are obvious. The limitation of varying text is that its identity as text is totally lost within graphic_macros_. This has several disadvantages, most of them connected with GSP's. For example, a GSP for a Diablo-type terminal might want to know that a user is requesting "the string 'Nottingham' in English Gothic type measuring 6 points by 12 points" so that it can use its discretion to punt the entire request and replace it with a typed version of the string rather than plotting ten small, black clouds. Alternatively, a GSP for a plotter that has the capability to scale and rotate its own internal character set (this is a common capability) cannot use this speedy feature— plain text elements are not allowed to rotate or scale, and varying text elements are not identifiable as text. Finally, there are several attributes of textual presentation that neither of the existing forms of text address: inter-character spacing, text plane (the direction of the baseline and plane face on which the text is drawn before it is rotated), and text direction (possibilities that cannot be handled by text plane are right-to-left lettering without reflection, and up-to-down lettering where the letters remain right-side up).

The ANSI Core design of text provides most of the definition necessary, and it will be used as a basis for new text capabilities in the MGS. Three unresolved points remain. First, we will need to redefine some of the ANSI constructs to handle fonts with variable-width characters. For example, the ANSI definition of character size and spacing makes the basic assumption that all characters are the same width to start with— yet we need to retain our support of our variable-width, graphic-art-quality fonts that we currently provide under the varying text capability. Second, we need to define the appropriate interactions between textual mappings (plane, direction, size, spacing) and graphic mappings (rotation, scaling). Finally, we need to specify how fonts are identified as part of a graphic structure and how this specification is transmitted to and used by a GSP.

EASING IMPLEMENTATION RESTRICTIONS

There are three implementation restrictions in the current MGS that users seem to run into more than others: size of the WGS, precision of Multics standard graphics code (MSGC), and size of graphic buffers. Incompatible changes will be necessary to ease these restrictions.

Currently, the working graphics segment (WGS), the buffer in the process directory in which graphic objects are assembled and edited, is limited to one segment. A list structure manipulator package especially designed for the graphics system[1] performs storage allocation, item threading, and garbage collection on this single segment workspace. This limitation extends to

---

[1] lsm_, based upon a 1969 design by Meyer and Skinner and heavily modified since.

permanent graphics segments (PGS's) as well, since they are just lsm_ segments kept in permanent filespace.

A new version of lsm_ that handles multi-segment WGS's and PGS's has already been coded and debugged. It runs some 15% slower than the old version in worst cases, due to the double indirections and the extra cleverness and care necessary to reclaim all possible available free space— a consideration that was unimportant when graphics structures were smaller.

It implements a (technically) incompatible change: the "node values" returned by lsm_ (via the graphics system) as object identifiers were actually 18-bit offsets into the graphics segment. Thus, the calling sequences of most entries in the graphics system deal with a good many "fixed bin (18)" parameters. With the new multi-segment WGS, node values are now constructed in a fashion similar to vfile_ descriptors, and thus are fixed bin (35). This results in a change to many include files and most of the graphics manual, but since fixed bin (18) is done using fullword instructions, existing graphics programs should not break. (When they are recompiled, of course, the compiler will issue warnings triggered by the new declarations in the include files.) The only programs that would be broken by this incompatible change are those which store node values in unaligned variables. I don't know of any, nor do I think it likely there are any, since the current calling sequences won't accept unaligned variables by reference anyway. No one currently stores unaligned node values in a permanent file, because node values are not conserved across processes.

Multics standard graphics code (MSGC) is a representation of graphic structures designed to be transmitted over communication lines to a terminal. The graphic compiler walks a specified graphic tree structure in the WGS, converts it into MSGC, and uses the I/O system to ship it out to the terminal. In the event that MSGC is not directly digestible to the terminal (which is usually the case), it is intercepted by the graphics support procedure (GSP) and translated into something equivalent but more palatable.

Currently, MSGC possesses a well-defined but rather intractable set of formats. Various graphic effectors are coded in various formats, largely by executive fiat, for reasons rooted in oral history. The extensibility of MSGC is next to nil. Users have complained that they need to exceed the allowable extents of object space— -4096 to 4095 in increments of 1/64— a restriction imposed solely by MSGC format limitations. Also, the precision of the current node format is insufficient to hold the node values from a new multi-segment WGS— already requiring an incompatible change. Finally, the remaining available character codes under the current assignment strategy are too few to accommodate the new primitive effectors we need to add to the graphics system. For all these reasons, MSGC must be changed.

No matter how MSGC is changed, a flag day will result for all sites with home-grown GSP's. Philosophically speaking, MSGC is the "native language" of the Multics virtual graphics terminal (VGT)— and therefore, of anything that needs to simulate it, which includes the entire class of GSP's and intelligent programmable graphics devices. (It in fact is fortunate that no one uses any of the latter with the MGS, or the incompatibility would be even more serious.)

A new MSGC has been devised, implemented, and debugged that addresses the problems of extensibility and limits on object space. Measurements on the present prototype show that it averages 12% more compact than the old format, even though its range and precision are many orders of magnitude greater. It will have to be changed to address the character code reassignment problem. Its format and the new proposed interface to GSP's is the subject of an MTB to be distributed in the near future.

Finally, the size of graphic buffers has been identified by customers as a limitation. In the operation of the MGS previous to MR9, the graphic compiler assembled an entire structure description into MSGC in a single-segment buffer and passed it to the graphic_dim_ in a single write call. The graphic_dim_ parcelled it off, effector by effector, to the GSP for translation, then assembled the output (native device code) into another single-segment buffer. This was done so that the entire graphic object could be validated before sending any of it on to the device. If the graphic_dim_ found any error in the graphic code or received any error indication from a GSP, the buffer would not be sent. Additionally, it needed to delay sending any output to the device on the chance that it would encounter a directive in the GDT ordering it to turn a list into an array. This directive would cause "backtracking" and the resultant discarding of a portion of the device output buffer.

In MR9, input graphic code is checked for backtracking contexts before any part of it is offered to the GSP. Output from the GSP is then collected in a buffer whose size is determined by the GDT, and sent to the graphic device as the buffer fills. This not only removed one of the segment limitations, but resulted in an improvement with respect to the elapsed real-time in graphics display operations.

The other segment limitation is not as easy to resolve. The graphic_dim_ requires each graphic write call to consist of a complete graphic structure. This was done not only so the graphic_dim_ could double-check the correctness of the data in each write call, but because intractable asynchronous output problems cause enough trouble in the graphics system without our providing further encouragement and larger windows. The only apparent solution is to bite the bullet on the asynchronous output problem, and to invent a new "continuation effector" in MSGC which would be placed at the end of incomplete transmissions of MSGC to indicate that more was on the way. We may be able to put this improvement off a bit, due to the increased density of the new MSGC.

Conclusion

This MTB has attempted to contrast the design of the MGS with those of other contemporary systems; to explain the reason the current design decisions were made; to describe the current limitations of the MGS in terms of desirable features; and to propose mechanisms for implementing these features in a manner compatible with the current operating philosophies of the MGS. I welcome and solicit input on any of the topics or proposals discussed, as well as any the reader feels have been ignored.