Rainer Buhtz

CGM-Concepts and their Realizations

Invited talk at the Eurographics Conference, September 8-10, 1982
University of Manchester
To appear in: P. Bono, I. Herman (eds.): GKS Theory and Practice
Springer Verlag 1987

Preprint SC - 87 - 4 (June 1987)

Konrad-Zuse-Zentrum für Informationstechnik
Heilbronner Straße 10, D - 1000 Berlin 31

1) Requirements and Design Concepts

Common Graphics Manager (CGM) is the name given to the implementation of GKS, Level 2b, developed at the Free University of Berlin (1,6). This paper is a survey over the "early GKS implementation phase" 1980-1982.

Work commenced in February 1980. At the outset some basic design decisions were necessary on account of the special scientific computer environment in Berlin, because the "Berlin GKS" was intended to be a common graphical software package for all the machines (4).

- The software package had to be adaptable for different machine types with minimal effort. The original host computer for CGM is a CDC Cyber 835 (before that a Cyber 172). Further installations were planned for at least Siemens 7000-series (BS2000) and Harris H 100-series. CGM is now installed on many other machine types, for example IBM, Vax and UNIVAC. The interfaces to operating system dependent modules had to be defined on such a level as to allow installation without a deep knowledge of GKS, because many of the satellite computers in the university environment are run by users and in general there is no local graphics specialist available.

- The implementation language had to be FORTRAN. From a modern viewpoint, FORTRAN is neither an elegant nor a comfortable programming language. In addition the fact that FORTRAN is available worldwide is not a decisive argument in its favour, for by a similar argument one would drop all GKS activities in favour of an existing, well known standard package such as Calcomp. Nevertheless it was decided to use FORTRAN because interfaces between FORTRAN and higher level languages such as ALGOL 68 and Pascal may be provided rather easily, whereas the problem of providing interfaces in the reverse direction has never been solved satisfactorily. Hence a FORTRAN implementation covers a larger number of users. Taking this in conjunction with the first requirement the language chosen was ANSI FORTRAN 66.
  Meanwhile, we migrated to ANSI FORTRAN 77, of course.

The GKS standard contains some important basic concepts which have to be realized by an implementation. GKS uses a set of internal tables which are conceptually dynamically allocated, that is they may be requested (for example Workstation State List during OPEN WORKSTATION) or cancelled (for example Workstation State List during CLOSE WORKSTATION). The total number of workstations simultaneously open or of segments stored on different workstations is unknown in advance and is conceptually limitless. These considerations lead to the conclusion that for the efficient and correct realization of GKS some basic software components are required for dynamic allocation and deallocation of memory. To ensure fast access, and to avoid the storage of redundant information, data blocks have to be stored only once and additional modules for handling pointers and chained tables had to be provided.

GKS requires output primitives of unrestricted length to be handled, for example for transformation, clipping and routing to all active workstations; and requires such primitives to be accessed in internal segment storage. This leads to further

dynamic requirements for a GKS implementation. Temporary auxiliary arrays of different lengths have to be provided for:

- copying information (e.g. area clipping; the clipped polygon may have more border points than the original, or may even be split into disjoint parts);

- reading information from segment storage (e.g. for picture regeneration or INSERT);

- storage of dynamic attribute changes for the next UPDATE.

Such software components may be provided very easily in a high level language such as ALGOL 68 or Pascal. Algorithms may be found in any textbook for such languages (7).

The decision to implement CGM in ANSI FORTRAN 66 resulted in the definition of two smaller software packages:

- CGM Memory Manager

- CGM Table Chain Manager

These are not part of GKS but were necessary to enhance the functional capability of ANSI FORTRAN 66 to provide pointers and dynamic arrays. There is no loss of portability here for the packages themselves are written in ANSI FORTRAN 66.

In this paper it will be shown how CGM implements GKS. Special issues of interest are:

- portability;

- efficiency;

- closeness to the standard.

2) CGM Memory Manager

The CGM Memory Manager is a subroutine package for handling dynamic memory blocks. It contains two subroutines which provide the following basic features: Get Block which allocates a block of length N and delivers a start address (relative to a fixed COMMON block) to the calling program; Release Block which marks the specified block as no longer in use, the block is then available for later Get Block calls. These two routines solve all the dynamic problems of GKS, for example "how many workstations may be opened at the same time?", "how many segments may exist at the same time?", "how large can output primitives be on segment storage?". These questions are reduced to one question: "how large does the dynamic buffer area of the CGM Memory Manager have to be?". By choosing a suitable dimension for one unstructured COMMON block, all applications can be accomodated.

The standard buffer limit can be tuned at installation time by setting appropriate installation parameters. It is not necessary to change the source code. Equally, it is so easy to increase the buffer limits that this may be left to the application programmer. Most systems allow the length of a common block to be increased, for example by using a BLOCK DATA subprogram. The user may work with any buffer length (within machine constraints) by linking such a module and calling a special routine which

causes the CGM Memory Manager to rearrange its pointer chains appropriately. Thus a computing centre may provide a standard buffer length which meets the requirements of the majority of its users, without having to consider peak requirements.

When the CLOSE GKS function is executed, a statistic about memory usage is written to the error file.

The dynamic buffer area is used throughout CGM, using only the two routines described above. Thus it is possible to replace them by operating system oriented routines as it is being done for the CDC machine (2,5). The CDC operating systems provide an interface for requesting new field lengths during lifetime of an application program, the so-called Common Memory Manager. The two CGM routines can be replaced by calls to the Common Memory Manager, delivering the base address of the requested block relative to the same COMMON block. Thus from the point of view of the higher CGM routines nothing has changed.

This leads to a basic concept of CGM. Whenever operating system oriented optimizations are possible (and useful) they may be performed easily by local staff, because an ANSI FORTRAN solution is provided which may be used immediately, and the interfaces are small and simple, so that special knowledge about the internal CGM structure is not necessary. The optimization may be performed in due course once the system is running.

3) CGM Table Chain Manager

The CGM Table Chain Manager is a subroutine package for handling pointers and chained tables. An example is the list of all workstation state lists for open workstations which may vary in length depending on the number of workstations. Every workstation state list itself contains further dynamic tables, for example pen tables. In practice this means that the workstation state list contains a pointer to the first pen table entry which points to the next and so on.

The CGM Table Chain Manager consists of routines for the following basic functions: adding new members to an existing (or new) chain; taking a member outside a chain and rearranging the chain; accessing specific chain members; releasing complete chains. For these purposes memory allocation and deallocation is necessary. This is performed by the CGM Memory Manager. Hence the CGM Table Chain Manager is written in ANSI FORTRAN. Machine dependent optimization is unnecessary. The CGM Table Chain Manager can be used with every memory handling package that respects the Get/Release Block interface. CGM table chains may coexist with unstructured temporary arrays because the CGM Table Chain Manager is only one special user of the CGM Memory Manager.

The following examples illustrate the cooperation of the different CGM table chains. First, as an easy example, consider setting a pen representation. If the user defines a new pen, the CGM Table Chain Manager requests a new memory block and fits it into the workstation pen table (see Fig.1). The specified attributes are then entered into the block. Such a pen table can only grow during lifetime of a workstation because GKS does not provide a "delete all pens" function. The chain is released when the workstation is closed.

As a more complicated example, consider segments. Segments (and workstations) can be created and cancelled during the lifetime of an application program. The data blocks (the state lists themselves) are held centrally in GKS. The cross-references

between segments and workstations are performed by pointers or chains of pointers (see Fig.2). For example the CLOSE WORKSTATION routine should cancel everything in GKS connected with the workstation, so that after the function is executed it is as if the workstation had never been open. This is achieved by first accessing the data block of the workstation state list and releasing its subtables, such as the pen table, with the CGM Table Chain Manager. The chain of segments stored in the workstation is checked. For every segment this subtable contains a pointer to the segment state list. The segment state list contains a chain of pointers to all workstations where the segment is stored. This chain has to be accessed for every segment stored on the actual workstation, and the pointer to this workstation has to be cancelled. If as a result the segment is not stored on any workstation, the segment itself is cancelled, its segment state list is released and its name is cancelled from the central list of segment names in use (inside GKS).

After this process, the chain of pointers to all these segments may be released itself, and - last but not least - the data block of the workstation state list is cancelled as well as the name of the workstation (from the central list of workstation identifiers in use). After this, all references to the workstation are cancelled.

"Releasing" chain members is handled cooperatively: the CGM Table Chain Manager only rearranges the chain, the single memory block is passed to the CGM Memory Manager. If the neighbouring blocks to the released block are also unused, they are combined together so that the free space is maximized.

4) CGM's Segment Storage Concept

Segment storage is an internal workstation in GKS. Segments stored in segment storage can be used by the INSERT function. Segment storage in CGM has an additional function. Such a file is generated internally for simulating picture regeneration on unintelligent workstations, even without explicitely opening a segment storage workstation. Interactive applications access the segment storage frequently, and so efficiency of access is important. ANSI FORTRAN 66 does not support direct access files and so it was necessary to define an interface (which was kept as simple as possible) to machine dependent direct access routines. The interface only supports the transfer of unstructured memory blocks of fixed (installation dependent) size. There is a higher level layer (ANSI FORTRAN) which handles the decomposition (and reconstruction) of the GKS data structures at the DI/DD interface (device independent/device dependent) into unstructured memory blocks.

It was known from the outset that it is not easy to generate modules to a special interface and localization of errors is difficult. A possibly unconventional method was used. In addition to defining the interface an ANSI FORTRAN solution was also provided by simulating direct access on a sequential binary FORTRAN file. Thus there is a reference version of CGM for comparison whilst an installation specific solution is developed. The full software can be tested immediately. The reference routines are not suitable for production use, though they are not so inefficient as one might suppose. This surprising fact is due to the way CGM used segment storage for picture regeneration. Segments are in the main read in the same order as they were "generated" (i.e. as they were written to segment storage). Conse-

quently in most cases segment storage is read sequentially and in such cases direct access cannot be faster than sequential access. There are only two cases in which real direct access is needed:

(1) Use of segment priorities: the user specifies explicitely the order in which segments are to be redrawn, i.e. which segments are "foreground" (high priority) or "background" (low priority). This opens up interesting applications, especially on raster devices. If the raster primitives FILL AREA and CELL ARRAY are used then automatically hidden surface removal is possible.

(2) Explicit usage of the segment storage workstation (INSERT SEGMENT function). If a segment has to be copied into the open segment, direct access to the segment storage file is required.

Benchmarks on a CDC computer have shown that the ANSI FORTRAN routines are comparable in speed with assembler direct access routines for sequential reading (normal regeneration), whilst for real direct access usage (INSERT SEGMENT) there is a factor of 1:10 in favour of the assembler routines.

Meanwhile, CGM uses ANSI FORTRAN 77 direct access I/O. Thus, no machine dependent routines are required.

The same direct access interface is used by CGM to handle a second internal file, the CGM software text file.

5) Software Text in CGM

One of the most interesting applications of GKS is the generation of software text using different fonts. The total number of available text fonts is implementation dependent. The storage of coordinates is carefully optimized in CGM.

There are now 9 text fonts available in CGM. These include standard text and high quality roman, italic and Greek fonts. In addition, special symbols for mathematics and natural sciences (e.g. integral and differential operators) and centered symbols are available. For CGM Version 4.0, which will implement GKS 7.0 (planned for the end of 1982) it is intended to implement all 21 of Hershey's character fonts including gothic fonts, cyrillic alphabet and further special symbols.

For all these fonts, an 8-bit representation for x and y coordinates is sufficient and so efficient data packing is possible if there is only one font in memory at the same time. CGM text fonts are therefore kept on a direct access file in binary packed format. This file is accessed through the same interface as segment storage. At installation time, this file is generated from a coded data file contained in CGM, using a "CGM pre-processor". Portability was achieved because the file must be generated afresh for each installation. The modules themselves are portable. This concept is carried further in CGM.

6) Pre-processors in CGM

Despite all portability considerations, two installations of CGM on different machines cannot be identical; even a package written in ANSI FORTRAN needs adaptation. Care was taken to try to reduce this adaptation work to a minimum. The work is performed using pre-processors which are supplied with the CGM code.

The idea is the following. One or more "files" are required for installation of a CGM component. They are contained in the CGM code or have to be supplied at installation time. A pre-processor operates on these files and generates new files containing the component in a form suitable for use on the given computer. Pre-processors are provided for the following tasks:

- Source code tuning and setting installation parameters: the input files are CGM code and a small data file containing the installation parameters (e.g. block lengths, dynamic buffer size, computer word length). The pre-processor produces several output files containing a configured CGM source. Setting parameters by such a mechanism is not only easier than by hand, but is also safer.

- Generating the internal software text file: this pre-processor was mentioned in the last section. The input file is a coded text file (containing the text coordinates), the output file is a binary packed direct access file. This pre-processor can also be tuned through installation parameters (because it is a part of the CGM code) and thus it can use a machine dependent word length for packing etc. Another interesting feature is that the pre-processor is fully portable, but a binary WRITE produces different results on different computers, so that the resulting binary files are different. When this pre-processor is used after the installation of "real direct access" the software text file will be a direct access file.

- Generating the DI/DD interface: this pre-processor has to be used more than once when CGM is in use (every time when a new driver has to be installed). It configures CGM for the local graphical environment. The input file is a coded file containing a data block for every graphical device, the so-called Workstation Description Tables. Entries include screen size, resolution and special features of the given workstation, for example whether it is able to change colours of primitives dynamically.
  These data blocks are easy to maintain. The resulting output file contains the source for two subroutines. One subroutine is the physical DI/DD interface, in other words it contains FORTRAN CALL statements for every driver. The second subroutine contains all the workstation description tables in fast-access form. Hence, at runtime, no internal file or data conversion is required to access such data which is frequently used. The first subroutine contains all statements for all drivers. This implies that all drivers are linked to the application program even if they are not needed. In larger configurations this may lead to excessive memory requirements.

Following the CGM principle of allowing machine dependent optimizations, calls to dummy routines were provided in the OPEN WORKSTATION and CLOSE WORKSTATION routines which may be replaced by assembler routines for dynamic loading and unloading of drivers. Such modules are now available for CDC and IBM machines.

7) The Layer Structure of CGM

In conclusion, some remarks about the CGM code structure, especially with reference to portability, are in order (see Fig.3).

The entire central code, "GKS itself", is located in the highest machine and device independent layer. The special GKS drivers for Metafile Output, Metafile Input and Segment Storage also belong to this layer. Thus CGM is immediately available on every computer, though some modules may be replaced for optimization. As soon as real devices are linked to CGM, modules must be device dependent. CGM drivers are structured such that some driver components are portable, to facilitate moving drivers to other machines and writing new drivers.

At the highest device driver interface, the central CGM code is followed by a device dependent but machine independent layer. Here all device dependent operations, for example conversion to ASCII-bytes, are performed. Experience has shown that such components can be written in ANSI FORTRAN without loss of efficiency. Some device independent tasks (for example workstation clipping, pick simulation and software text generation) may be performed using a set of device independent driver utilities which are contained in layer 1. Exporting a driver together with CGM code means that this second layer is included.

Connecting a file to a terminal and sending ASCII bytes to a terminal or plotter requires machine dependent modules, because they are impossible to write in ANSI FORTRAN (or at least not efficiently enough).

There are well-defined interfaces to be adhered to, but they are simple interfaces and do not require knowledge of the internal features of GKS. Only very simple I/O routines have to be provided and for terminals they can be provided in a device independent manner. Portability is possible in another sense, the same modules may be used for a range of drivers. This feature makes the installation of new drivers easier.

Note

Since this work was first presented in September 1982, the project has continued. As of March 1987, the implementation has been brought up to GKS Level 2B (as defined in ISO Standard 7942). It runs on a wide range of machines from IBM-PC's to the Cray X-MP. The system is designed for use in open networks (on top of the "Deutsches Forschungsnetz (DFN)") and is in use worldwide in universities, research centres and private industry. A further account of the system appears in (3).

Fig. 1

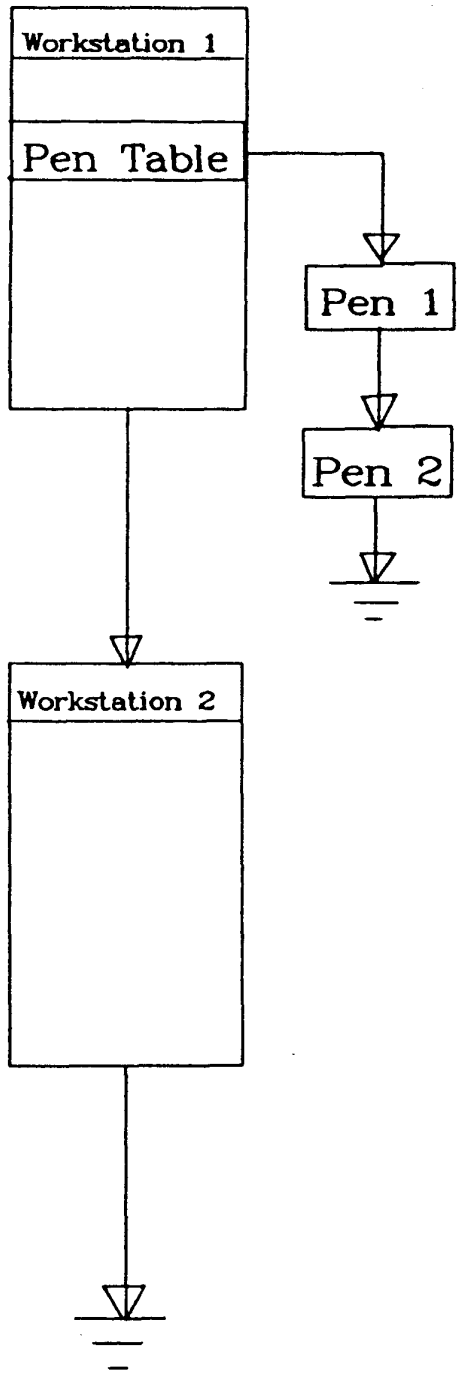
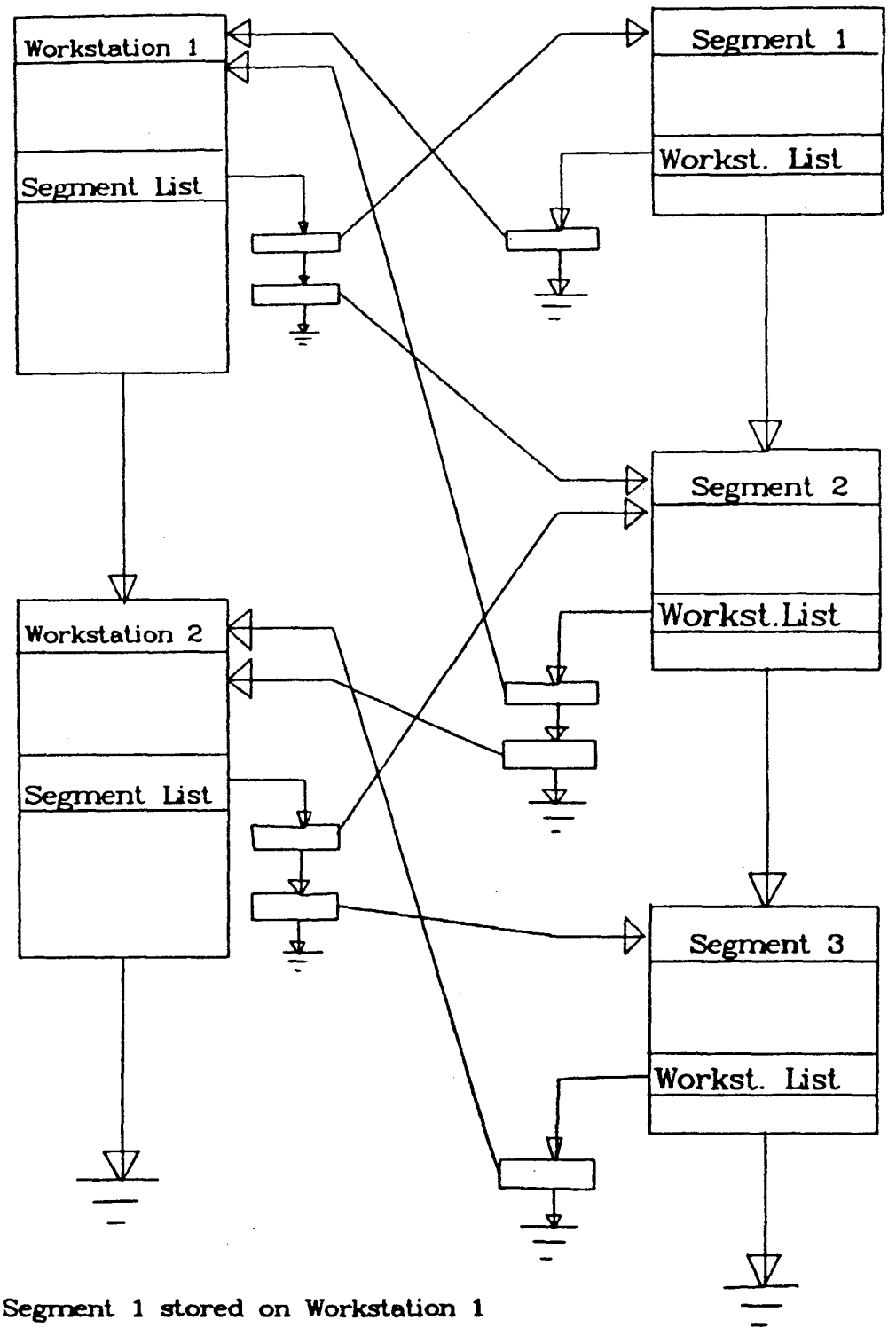


Fig. 2

Cross References between Workstations and Segments



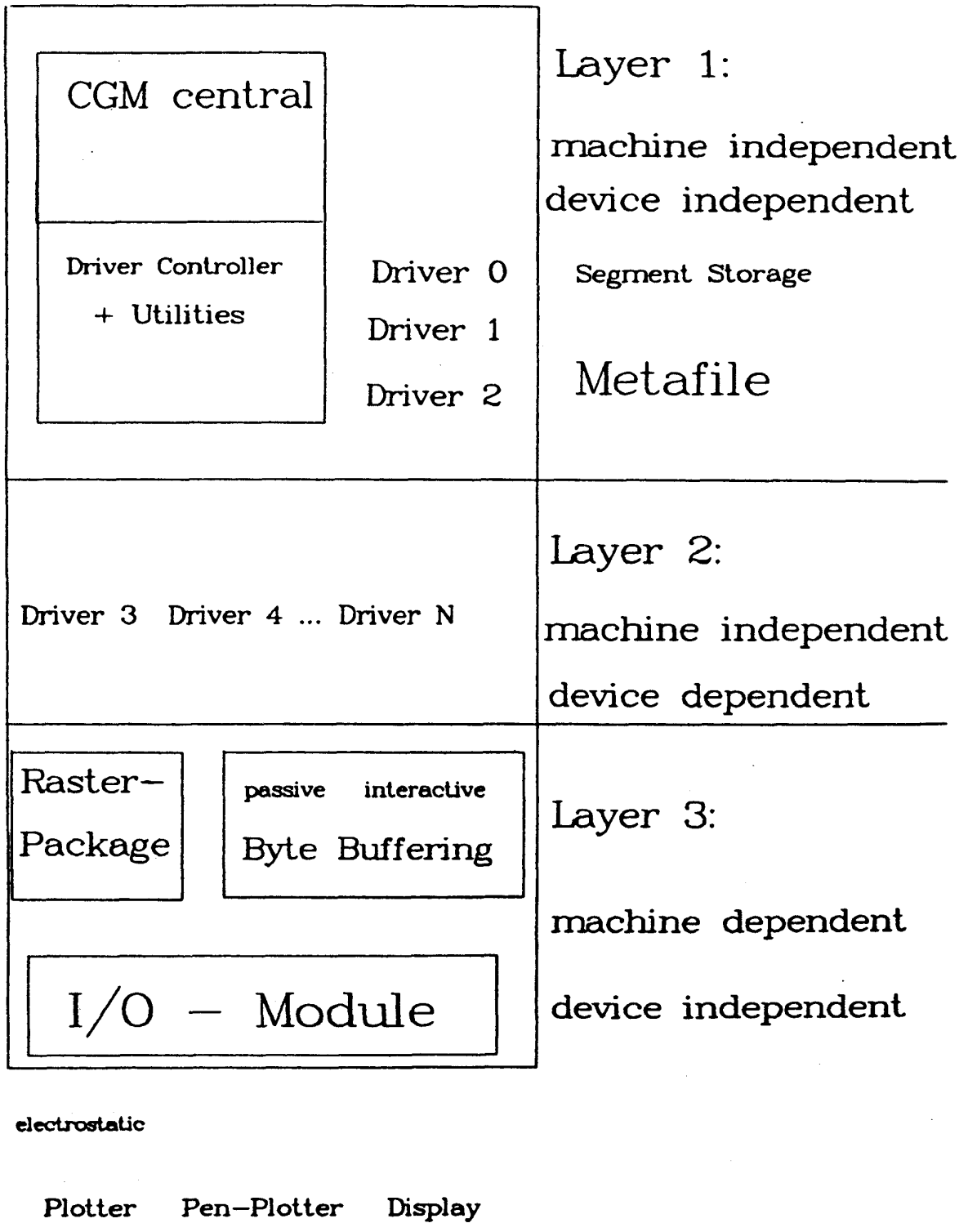
Segment 1 stored on Workstation 1

Segment 2 stored on Workstations 1 and 2

Segment 3 stored on Workstation 2

Fig. 3

CGM Layer Structure



References

1. J.Bechlars and R.Buhtz, CGM Handbook: Special Design Concepts and Installation Guide for COMMON GRAPHICS MANAGER Version 3.1, Freie Universität Berlin (April 1982).

2. J.Bechlars, "Experiences with CGM, the Berlin GKS Implementation", Conference Proceedings ECODU 33, St. Paul, Minnesota (1982).

3. J.Bechlars and R.Buhtz, GKS in der Praxis, Springer Verlag (1986).

4. R.Buhtz, "Implementation Policy at the Free University of Berlin", in : Report on the Workshop on the Implementation of GKS at ECMWF, ed. H. Watkins, Reading, Berkshire (UK) (1981).

5. R.Buhtz, "The Berlin GKS - concepts and their Realization on a CYBER", Conference Proceedings ECODU 31, Helsinki (1981).

6. International standard ISO 7942
   Information processing systems
   Computer Graphics
   Graphical Kernel System (GKS)
   Functional description
   ISO 7942/1985
   (This paper refers to the earlier version 6.4)