IMPLEMENTATION OF DATA SEGMENTATION IN A GKS BASED GRAPHICS SYSTEM

by

REBECCA EDWARDS MAY

B. S., Appalachian State University, 1978

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

Approved by:

W. J. Hankley
Major Professor

CONTENTS

1. SUMMARY 1

1.1 Definition of Graphic Kernel System (GKS)
1.2 Scope of Report
1.3 Scope of Implementation
1.4 Conclusions

2. GKS SPECIFICS FOR DATA SEGMENTS 7

2.1 Concept of Data Segmentation
2.2 Data Segment Attributes
2.3 Clipping and Transformations for Segments
2.4 Workstation Independent Segment Store
vs Workstation Dependent Segment Store
2.5 Segment Functions Defined by GKS

3. GKS VS CORE ON SEGMENTATION 13

4. DISCUSSION OF SEGMENTATION DESIGN ISSUES IN GKS 18

4.1 Attribute Binding for Segments
4.2 Permanent Segment Library Concept

5. KSU-GKS IMPLEMENTATION OF DATA SEGMENTS 22

5.1 Data Flow
5.2 Data Structures
5.3 Segment Manipulation Functions
5.4 Project Status

APPENDICES

REFERENCES

FIGURES

1. GKS LAYERS 1a

2. PICTURE STRUCTURE 7a

3. TRANSFORMATION PIPELINE 9a

4. FLOW of PRIMITIVES into WISS and WDSS 10a

5. ATTRIBUTE BINDING 19a

6. KSU - GKS DATA FLOW 22a

7. DATA FLOW WITH SEGMENTS 22b

8. WISS DATA STRUCTURE 25a

1. SUMMARY

1.1 Definition of Graphic Kernel System

The Graphic Kernel System (GKS) [Draf83], as proposed by the International Standards Organization (ISO), is expected to be approved in 1983 as an ISO standard for developing device-independent graphic software. At the highest level, GKS defines a language and device independent nucleus of graphics procedures allowing access to a wide range of physical devices, within the particular conventions of a given language. A layered model of GKS is shown in Figure 1, GKS Layers.

With the cost of hardware technology steadily dropping, graphics hardware now represents a much smaller percentage of the system cost than it once did. Users are becoming increasingly aware of the costs of writing and maintaining software. Thus, the primary motivation behind the GKS proposal is software compatibility across product lines and into future products, preserving software investments [Fich83]. And equally important, standards would allow a programmer to transfer the experience gained on one application to others; thereby increasing the productivity of now scarce graphics programmers [Bail83].

The GKS package provides application programs the

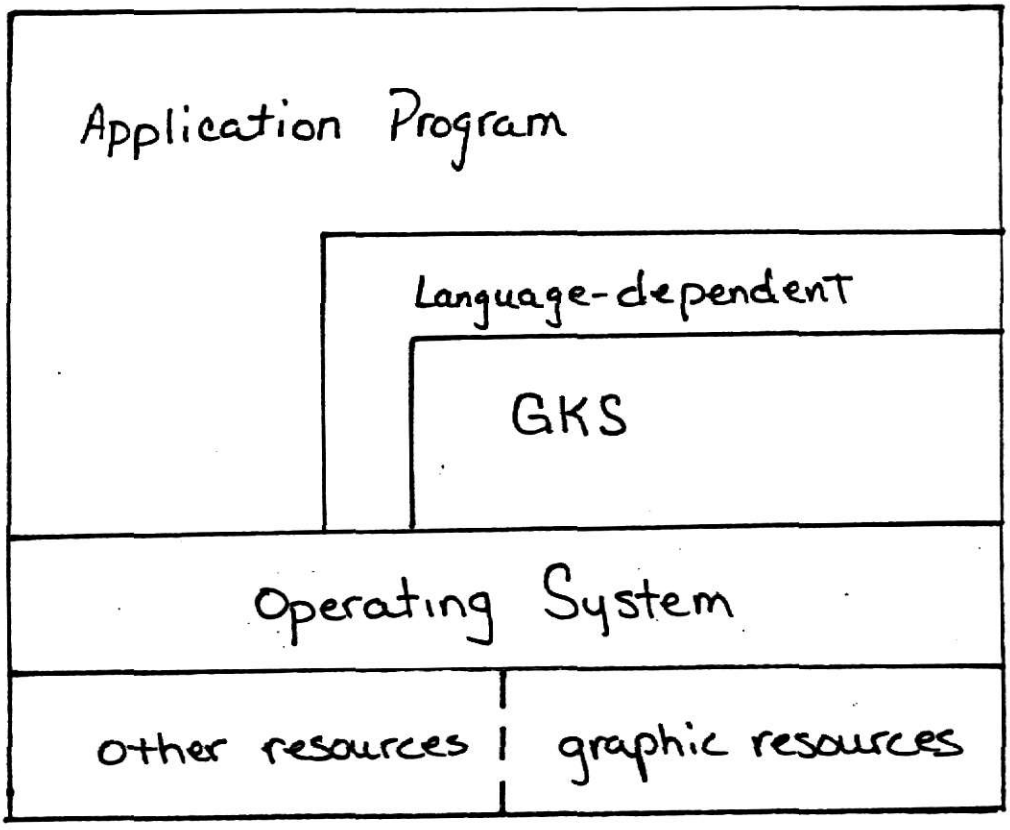


Figure 1

GKS LAYERS

capability to work with varied input and output devices. This capability is provided through callable graphics procedures which in turn drive the virtual graphics devices. Each device is driven by a device-dependent driver which maps the virtual interface onto the hardware interface. This mapping is defined by ANSI's Virtual Device Interface (VDI). VDI specifies the data format for describing a picture in a device-independent form amenable to a wide range of graphic devices [Warr83]. For an application program to access a new device, all that is needed is a new device-dependent driver. It is not necessary to rewrite the program [Shre82].

A workstation is a console that includes at least one or more various input devices such as an alphanumeric keyboard, function keys, a joystick, a control ball, or a lightpen. The various devices that make up a workstation are treated as one logical unit. For each workstation in a given GKS implementation, an entry exists in a workstation description table defining the capabilities and characteristics of that workstation. These tables are maintained at the GKS level, not at the application program level. The program may inquire via GKS as to which capabilities are available on a given workstation and proceed accordingly [Enca80]. The VDI specifies both a lean set and a rich set of functions. The lean set is designed to the lowest level and is intended to be supported by

all devices. The rich set, while not bound to the host language, could be emulated as device firmware and includes such optional functions as color and polygon fill.

A related standard also supported by GKS is the Virtual Device Metafile (VDM). VDM defines a data format for describing a picture in a device-independent form for translation and display on a variety of graphics devices. The real advantage here is the ability to transfer computer-generated pictures not only between graphics devices, but also between any graphics package supporting VDM.

1.2 Scope of Report

GKS supports two-dimensional output to and input from single or multiple workstations. The output primitives include line drawing, text drawing, and raster graphic primitives. Five classes of logical input devices are supported: locator, valuator, pick, choice, and string. Coordinates are transformed in a two-stage transformation process. The first stage can be set for each primitive; the second stage can be set for each workstation.

A segment facility provides the means for structuring a picture into subparts. Segments may be created and deleted, the segment attributes may be dynamically modified, and the segments

may be independently transformed. They can be displayed simultaneously or alternatively on different workstations.

Segmentation allows a logical association between individual primitives, such that each segment can be uniquely identified and manipulated as an entity. This provides one way of making selective modifications to the picture. The concept of data segments leads to the notions of multiple instances of a given segment within a picture, segments composed of smaller segments, and libraries of stored segments.

GKS specifies a set of allowable operations for data segments and defines a unique set of attributes to be associated with each segment. In addition to CREATE and DELETE, segments can be COP(Y)ied, INSERTed as a part of another segment, and independently ASSOCIATEd with a given workstation. The set of attributes includes a transformation matrix, visibility, highlighting, segment priority, and detectability.

1.3 Scope of the Implementation

A GKS based computer graphic system has been implemented at Kansas State University. This implementation evolved over a period of time as the result of the efforts of several students. Data segmentation capabilities were incorporated into KSU's already-running system. The GKS concepts were adhered to as

much as was feasible. Therefore, the data segmentation also followed the GKS guidelines. The data flow is as specified in GKS. Data structures were designed within the constraints imposed by an Interdata 3220 host under a UNIX* operating system. The heart of the system was implemented in PASCAL, not including some of the device drivers.

The preliminary study of the GKS specification raised several questions that were not clearly defined or well justified in the ISO document. The decision whether to bind segment attributes at creation time or display time can be argued both ways, depending on the intended uses for the system. Provisions for a permanent segment library are not explicitly defined. In fact, certain GKS requirements preclude a flexible segment library.

1.4 Conclusions

Through research and the experience of an actual GKS implementation, certain conclusions became obvious. Clearly, GKS was intended to be used by an application program as opposed to being used interactively. The defined functions are hardly 'user friendly'. It seems there are an unnecessarily large number of functions defined. The user is burdened with every detail. An intimate knowledge of both the internal and external

coordinate systems is a requirement. Transformations must be defined by a full transformation matrix. Some of these things could be more transparent to the user.

The data segment concept should be extended to explicitly provide for a flexible segment library. This would seem a basic requirement; to preserve 'building blocks', or segments, from one session to the next.

Of course, GKS claims that a standard promotes software portability. The reader may or may not subscribe to the idea of portable software. However, standardization does improve the user's understanding of any particular graphics system, thus programmer portability. That is surely worth something in a time when software costs outweigh hardware costs.

As the first international graphics standard, the GKS will certainly have a big impact on future graphics software design. Several European and U.S. companies with GKS-compatible software are now supporting the standardization of the system. Whether the supporters of Core will abandon their efforts and considerable investment to convert to GKS is unclear at present. Peter Bono, ANSI x3h3 chairman, isn't sure that Core will disappear. He feels that GKS doesn't provide enough functionality for all users.

* UNIX is a trademark of BELL LABORATORIES.

2. GKS SPECIFICS FOR DATA SEGMENTS

2.1 Concept of Data Segmentation

Data segments provide a means for structuring a picture into subparts. Output primitives, bound with their attribute values, can be grouped together in a segment. The segment defines a logical association between the individual primitives such that each segment can be uniquely identified and modified as an entity. It is the basic unit for manipulation and change. The primary motivation behind structuring a picture into segments is to allow selective modifications to the picture, which is simply a collection of segments. Any segment can be individually transformed or deleted without affecting the rest of the picture. Primitives that are generated outside segments, non-segment primitives, are sent to all active workstations, but the application program has no further access to them. A picture can contain multiple instances of a given segment. And, a segment can be a part of several pictures, or even a part of another segment, thus implying a hierarchical picture structure. See Figure 2, Picture Structure.

A segment is simply a collection of the output primitives that define a particular object, a subpart of a picture. The attributes which control the appearance of the primitives are

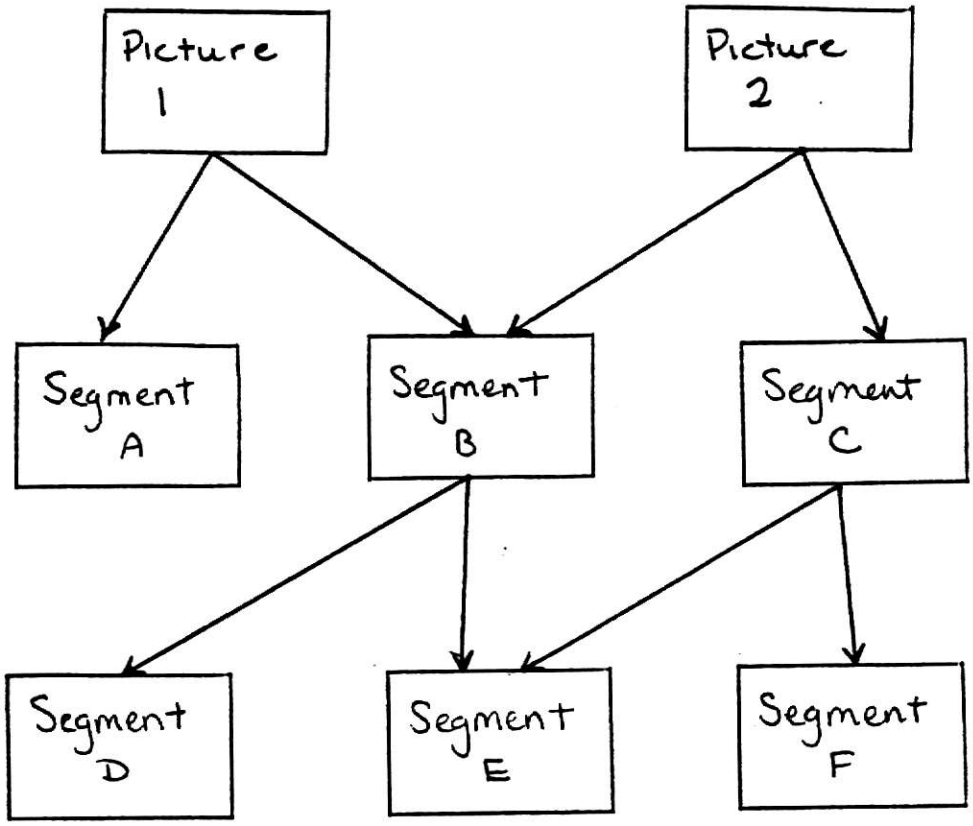


Figure 2 Picture Structure

specified modally and are bound to the primitives when it is created. These attributes include all the geometrical aspects of the primitives such as line width or character height.

2.2 Data Segment Attributes

In addition to the primitive attributes associated with the primitives of a given segment, the segment itself has attributes associated with it. The segment attributes affect the appearance of the view whose description is stored in the segment. They are characteristics of a segment's image, not of the object of which the image is a view. They are defined uniquely for each segment and they affect all the primitives of a given segment.

VISIBILITY indicates whether or not a segment is displayable. It is possible to have a segment included as a part of a picture, but not be visible. HIGHLIGHTING simply indicates whether a visible segment is highlighted or not. It is a variation on the visual characteristic "blink". DETECTABILITY indicates if a segment is selectable with a pick device. PRIORITY establishes a preference for overlapping segments. It determines how GKS will handle picking and display of overlapping segments. And finally, a TRANSFORMATION matrix is specified for each segment. It defines any translation, scaling, or rotation to be applied to the primitives of the

segment.

2.3 Clipping and Transformations for Segments

Segment transformations are done in normalized device coordinates. They are performed after the normalization transformation, but before any clipping. The transformation is actually stored in the segment state list, it is not performed in the segment storage. Each time the segment is redrawn, the transformation is performed. Clipping is performed after the normalization and segment transformations have been applied. It is delayed until the segment is to be displayed on the workstation. Each primitive is clipped against the clipping rectangle associated with the primitive when it was created. Figure 3, Transformation Pipeline, shows the relationship between the normalization transformation, segment transformation, and clipping.

2.4 Workstation Independent Segment Store vs Workstation Dependent Segment Store

The Workstation Independent Segment Store (WISS) is the "community" storage for segments. The segment manipulation functions access this storage. Only one WISS is permitted in a GKS implementation. The point in the flow of data at which

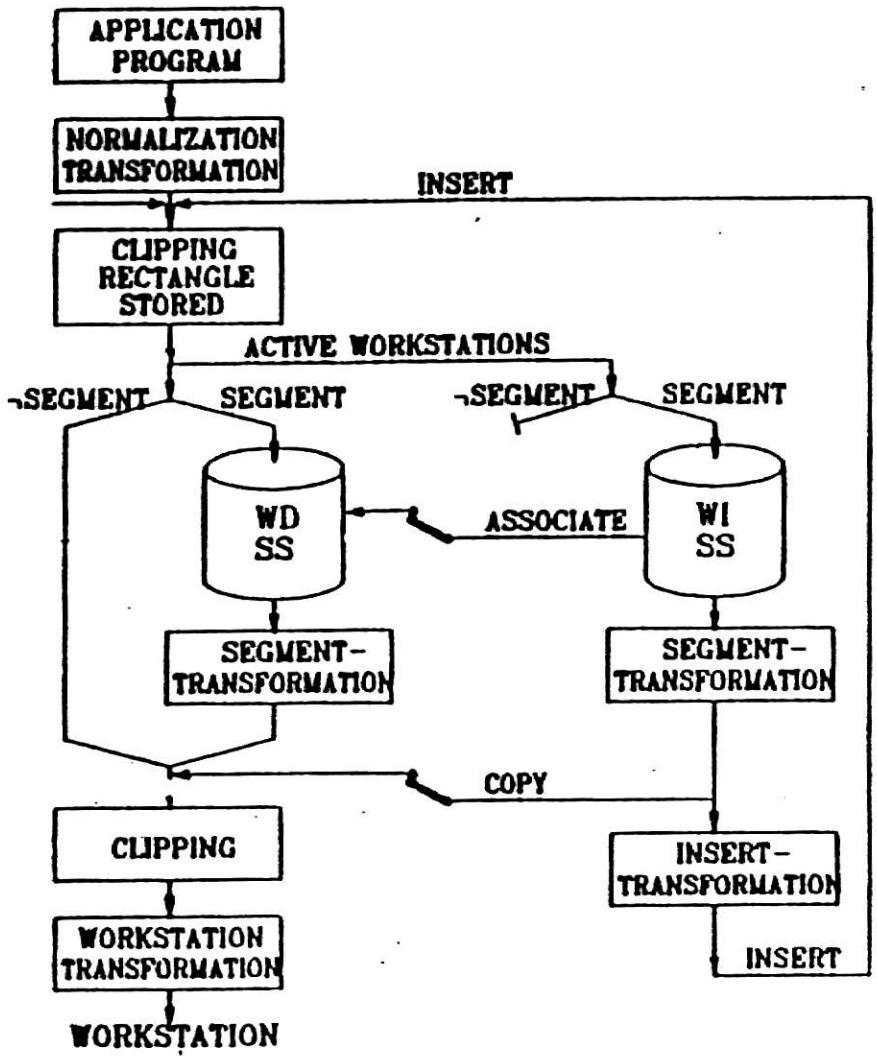


Figure 3
Transformation Pipeline

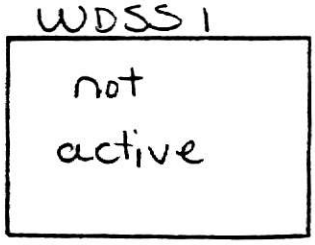
primitives are recorded in the WISS immediately follows the point at which data are distributed to the workstations. For this reason, WISS is a valid workstation category. The relationship between the WISS and the WDSS is shown more clearly in Figure 4, Flow of Primitives into WISS and WDSS. If no segment is open at the time a primitive is generated, then it is recorded in the Workstation Dependent Segment Store (WDSS) for each active workstation as a non-segment primitive. The WDSS stores the picture definition for that workstation. Because a WDSS exists for each workstation, and they are independent of each other, an application program can be building several different pictures simultaneously. And, each picture may share some common elements through the WISS. The non-segment primitives in a WDSS are transient. They provide a one time view. Invoking any of the following functions will clear all non-segment primitives from the specified workstation: CLEAR_WS, REDRAW_WS, or UPDATE_WS.

2.5 Segment Functions Defined by GKS

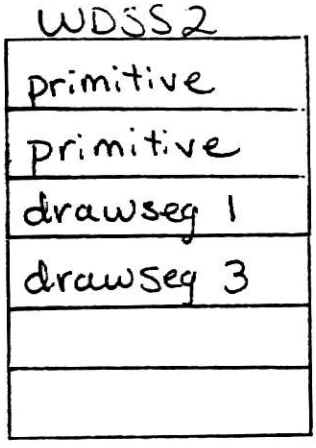
There are seven segment manipulation functions defined in GKS: CREATE, CLOSE, DELETE, DELETE_FROM_WS, ASSOCIATE_WITH_WS, COPY_TO_WS, and INSERT. They are discussed individually below.

The CREATE function opens or begins a new segment. All

No non-segment primitives added to inactive WDSS.



New primitives added here only if no segment is open.



New primitives added here only if a segment is open.

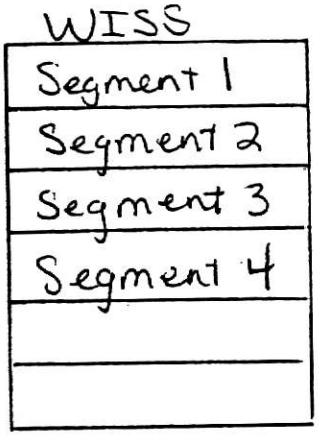


Figure 4
Flow of Primitives into WDSS and WISS

subsequently generated primitives will become a part of the current segment. CREATE does not force an implied CLOSE of a currently open segment. Instead, this function returns an error if another segment is already open. A segment may be RENAMEd while it is open, and any of its dynamic attributes may be changed.

The CLOSE segment function completes the previously open segment. No further primitives can be appended to it. However, any of its dynamic attributes may be changed. GKS does not check to see that the segment is non-empty. The program could conceivably generate null segments.

The RENAME function replaces each occurrence of the old name with the new segment name. The old name may be reused by the program.

The DELETE function removes all access to the segment. It is effectively canceled and its name may be reused. A segment cannot be DELETEd while it is still open.

The DELETE_FROM_WS function disassociates the segment from the specified workstation. If afterwards, the segment is no longer associated with any workstation, the segment itself is DELETEd. This seems a somewhat rigid restriction imposed on the user, not lending itself to libraries of stored segments.

The ASSOCIATE_WITH_WS function adds the segment to the set of segments associated with the specified workstation. The clipping rectangles are copied unchanged.

The COPY function dumps the segment primitives to the specified workstation after transformation and clipping. The primitives are not stored in a segment, i.e., the segment does not exist on the workstation as with ASSOCIATE_WITH_WS.

The INSERT function copies the primitives of another segment into the currently open one after the segment transformation has been performed. A second transformation can also be specified, possibly the normalization transformation. Each primitive processed is assigned a new clipping rectangle in the same manner as directly created primitives. This is not intended to be a recursive function: a segment cannot be copied into itself. After a segment is closed it cannot be modified or extended, i.e., no primitives can be added or deleted from it. No provisions were made to allow segments to be "edited". Segments can be "extended" awkwardly by beginning a new segment, INSERTing the segment to be extended, then appending the additional primitives.

3. GKS VS CORE SEGMENTATION

In April of 1974, ACM SIGGRAPH commissioned a Graphics Standards Planning Committee to develop a package of basic graphics primitives (Core) to allow creation and viewing transformations of 2D and 3D line drawings. Soon thereafter, the German Standardization Institute, DIN, began design of a graphics package that became GKS. Although very similar to Core, GKS was designed to be a 2D system with less extensive functionality than the Core specification [Shre82]. There are many basic ideas that both systems share. But the systems differ in the implementation of some of the concepts [Stat77].

Both GKS and Core are intended to provide device independence by shielding the applications program from specific hardware characteristics. This shielding is done at the functional level: the user's device-independent routines call device-dependent internal routines to map specific commands to the selected hardware device driver.

Floating point world coordinates are used by both Core and GKS to describe an object to the graphics system. Thus, application programs can define a picture in device-independent coordinates. Because the coordinates passed to either graphics package are inherently dimensionless, a program can use coordinates that are natural to the application at hand

[Berg78].

In order to produce an image on a view surface, the user's world coordinates must be mapped onto the appropriate coordinates of the physical device. First, the world coordinates of the window are scaled to the normalized device coordinates of the viewport such that the window boundaries coincide with the edges of the viewport. This is the normalization transformation of GKS. Both Core and GKS allow the 'window' to be dynamically specified. Secondly, the picture is clipped to the current window so that only those portions of objects that lie within the window will be displayed.

GKS allows a workstation transformation in addition to the normalization transformation. The appearance of a primitive is essentially defined by two stages. In a first step, a "workstation independent picture" is drawn on an abstract viewing surface. In a second step, for each active workstation, this "picture" is combined with workstation dependent data and sent to the physical viewing surface associated with each of these workstations. The ability to specify a window and a viewport for each workstation allows different views of the composed picture to be displayed on different workstations. For example, a drawing could be output on a plotter to scale and simultaneously some part of that drawing could be displayed on the entire view surface of an interactive terminal.

The workstation concept of GKS (based on a description table, a state list, and a set of management and inquiry functions) not only provides a means for the optimal adaptations of application programs to hardware capabilities of graphic terminals, it also lends itself to an adequate treatment of remote graphic operations. Also the idea of treating the workstation as a functional entity supports the trend toward intelligent graphical systems: an operator handles a collection of graphical I/O devices as one operational unit [Enca80].

When an output primitive is called, it is defined by the values of several attributes, including color, intensity, line style and width, and text font, size and precision. The Core system defines each attribute singly. This method of specifying attributes requires that the programmer set each attribute individually and then redefine each one as the program changes or moves to a different output device. The programmer is forced to anticipate the device-dependent attributes, and additional coding for each accessed device is necessary. The GKS method allows the flexibility of singly defined attributes, and also provides a bundling technique. With bundling, an output primitive can be assigned a group of attributes selected by a primitive index.

The idea of data segmentation is conceptually the same in GKS and Core. The purpose being to facilitate picture

modification. Both systems define a segment to be a separate entity, and a picture to be a collection of segments.

Two segments types are specified in Core: non-retained and retained. Non-retained segments provide for picture definition without the possibility for subsequent access or modification. They correspond to the non-segment primitives of GKS. Non-retained segments or non-segment primitives are displayed on the view surface until the view is updated or the picture is redrawn in its entirity. Retained segments are the counterpart for GKS segments. They can be renamed and deleted, also, a retained segment's attributes can be dynamically modified. The segment manipulation functions specified by Core are CREATE, CLOSE, RENAME, DELETE, and DELETE_ALL. In addition, GKS specifies INSERT, DELETE_FROM_WS, ASSOCIATE_WITH_WS, and COPY_TO_WS.

A segment's dynamic attribute values can be changed any time after the segment has been created. Both GKS and Core define visibility, highlighting, detectability, and a segment (view) transformation. However, segment priority is only defined in GKS.

When a segment is CREATEd, it becomes the currently open segment and all subsequent primitives become a part of that segment. The dynamic attribute values for the newly opened segment are determined from the current attribute values, but

may be changed at any time. Also, the current selection of view surfaces (workstations) is recorded in a list associated with the segment. This list is permanently bound to the segment in Core. But, GKS allows dynamic view surface selection via DELETE_FROM_WS and ASSOCIATE_WITH_WS. In Core, there is no corresponding list associated with the view surface (workstation) as in GKS.

Core specifies that at segment creation, the "current position" is set to the origin of the world coordinate system. GKS supports only absolute positioning, where each primitive has its coordinates fully defined. The use of current position value is inconsistent and can be confusing at times [Bail83]. In addition, when a segment is created in Core, the current value for each attribute is reset to a prespecified system default for that attribute. Core does this in an effort to allow the appearance of objects described by various segments to be completely independent of the order in which the segments were created.

4. DISCUSSION OF SEGMENTATION DESIGN ISSUES IN GKS

4.1 Attribute Binding for Segments

Each instance of an output primitive has both geometric and non-geometric attributes assigned to it at creation time. The geometric attributes restrict the shape and size of a primitive. They are individually specified and therefore workstation independent, and where appropriate, are expressed in world coordinates. Once these attributes are bound to their respective primitives at creation, the values are subject to the same normalization transformation as the data contained in the definition of the primitive.

Non-geometric primitive attributes restrict the appearance of output primitives. They are specified via a bundle table, there is one attribute per primitive called the index. The index is used to access the specific bundle table entry that defines the non-geometric attributes. There is a separate bundle table for each primitive and a separate set of tables for each workstation. Therefore, the non-geometric attributes are considered to be workstation dependent and their values are restricted to the valid values for a given workstation. Even though a primitive's attribute index cannot be changed, the values in the bundle tables may be dynamically modified. When

the primitive is displayed, its non-geometric attribute values are resolved.

Once a segment is closed, the primitives that make up that segment cannot be modified. The clipping rectangles and the primitive attributes are stored along with the primitives in the segment storage. But, it is still possible to change the segment attributes and the workstation attributes, and to apply geometric transformations. Figure 5, Attribute Binding, shows the flow from creation through display and where attributes are bound.

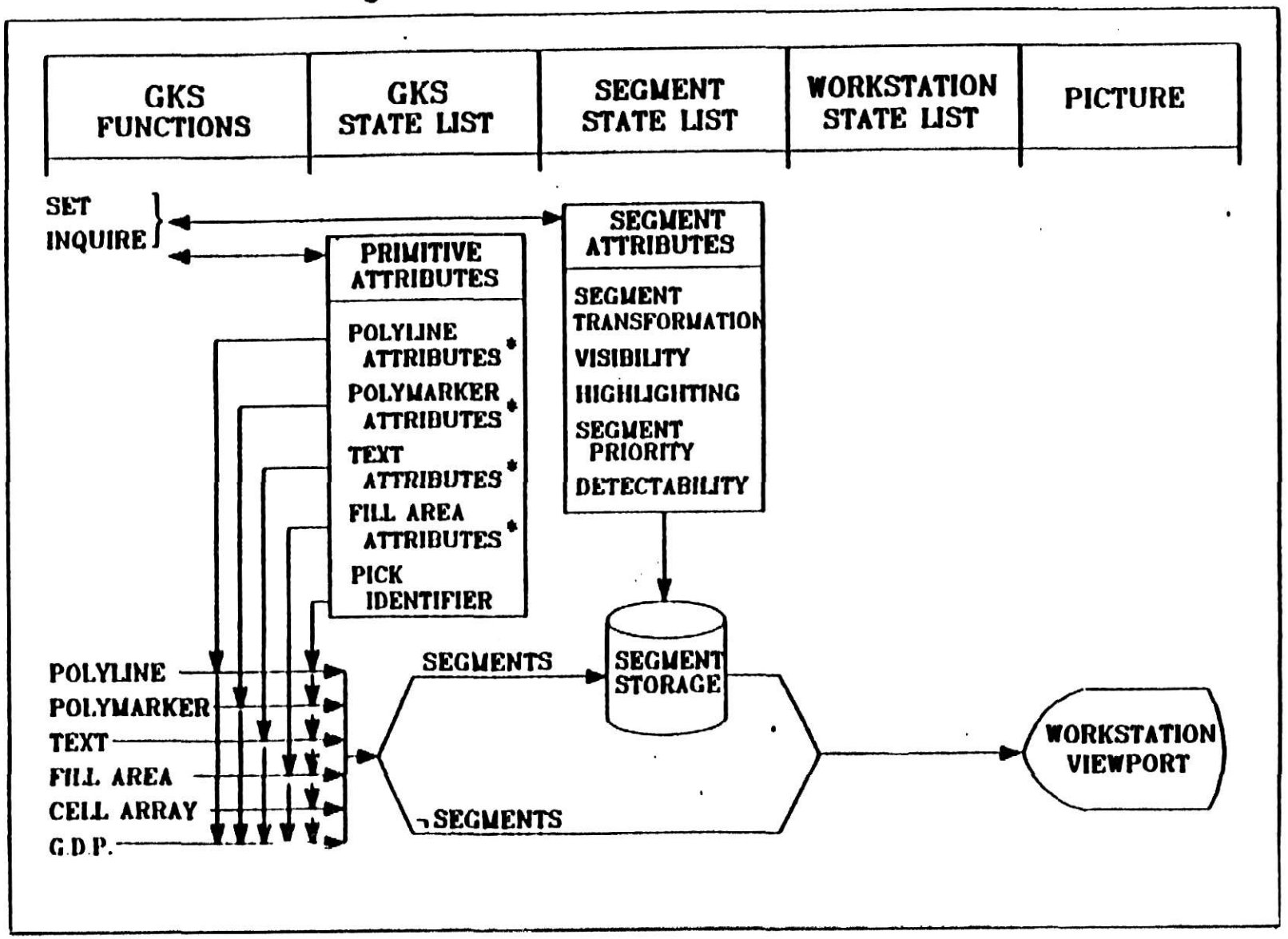
The alternative would be to have a segment's primitive attributes bound at display time. The view of the object would be incorporated into the prevailing environment, i.e., influenced by the current attribute values. The view of the object, its appearance, would depend upon the picture at hand. This approach is in fact the approach that was taken with the implementation at Kansas State University. It was felt that the segment would be more usable, less rigid, this way. A segment that is intended to be used in several pictures, may indeed need to take on the characteristics of each.

4.2 Permanent Segment Library Concept

Another very important concept in computer graphics is that

ATTRIBUTE BINDING

Figure 5



of stored segments, with a segment being a modular picture component. Data segmentation was initially implemented to allow selective modifications to the picture. It is also advantageous in that individual segments can be shared across several pictures, saving generation time and encouraging uniformity. It would seem only natural to build up a library of segments to be used over and over and from one session to the next. At the beginning of each session the library would be copied in from disk storage, and at the end of each session a possibly updated library would be copied out to disk for storage.

The GKS specification includes no provisions for such a library. Nor is there any specification for a facility to edit segments after creation. In addition, two other GKS characteristics do not easily lend themselves to the library concept; attributes bound at creation, and fully specified absolute coordinates.

Binding primitive attributes at display time affords the application program much more flexibility in using a segment. As an item of a library, a segment is intended to be "generic" and used repeatedly. However, this is less likely if the view of the object defined by the segment has been permanently bound to a previous environment.

Given that a segment is expected to appear in several

pictures, several times within a picture, and even within another segment, it would make sense to express the primitives in relative coordinates utilizing the current position. Relative coordinates would preclude the need to transform the segment for each picture.

Several of the segment manipulation functions could easily be adapted to support a stored segment library. Already, the segments attributes can be dynamically modified. The DELETE_FROM_WS function should allow a segment's set-of-associated-workstations to become empty. Now it seems reasonable that all segments are not in use at all times. The counterpart, ASSOCIATE_WITH_WS, should allow non-existent segments to be associated with a workstation, providing only a warning message. The intent being to generate the segment at a later date. Let it suffice to initially create only a skeleton of a drawing.

5. KSU-GKS IMPLEMENTATION OF DATA SEGMENTS

5.1 Data Flow

The GKS implementation at Kansas State University was installed in stages. At the time when data segmentation capabilities were added, only a minimal implementation had been completed. Only one workstation existed, and therefore only one WDSS instance existed. Multiple workstations were not enabled in the GKS package nor did any appropriate device drivers exist. The basic Line and Marker primitives were available, including attribute bundles. The Text primitives had not been completed. See Figure 6 KSU-GKS Data Flow.

To simplify the initial implementation, the normalization transformation as well as window clipping were applied against the primitives before the corresponding WIS record was built. (Note that the WDSS refered to in the GKS document [Draf83] was implemented as the WIS, and the WISS was implemented as the SEGSTORE plus the SEGTBL.) At a later date however, the window clipping will be applied just as the data is sent on to the workstation. The clipping rectangle in effect when the primitive is generated will be saved at the time the WIS record is built. This will comply with the data flow prescribed in the GKS document [Draf83]. See Figure 7 Data Flow with Segments.

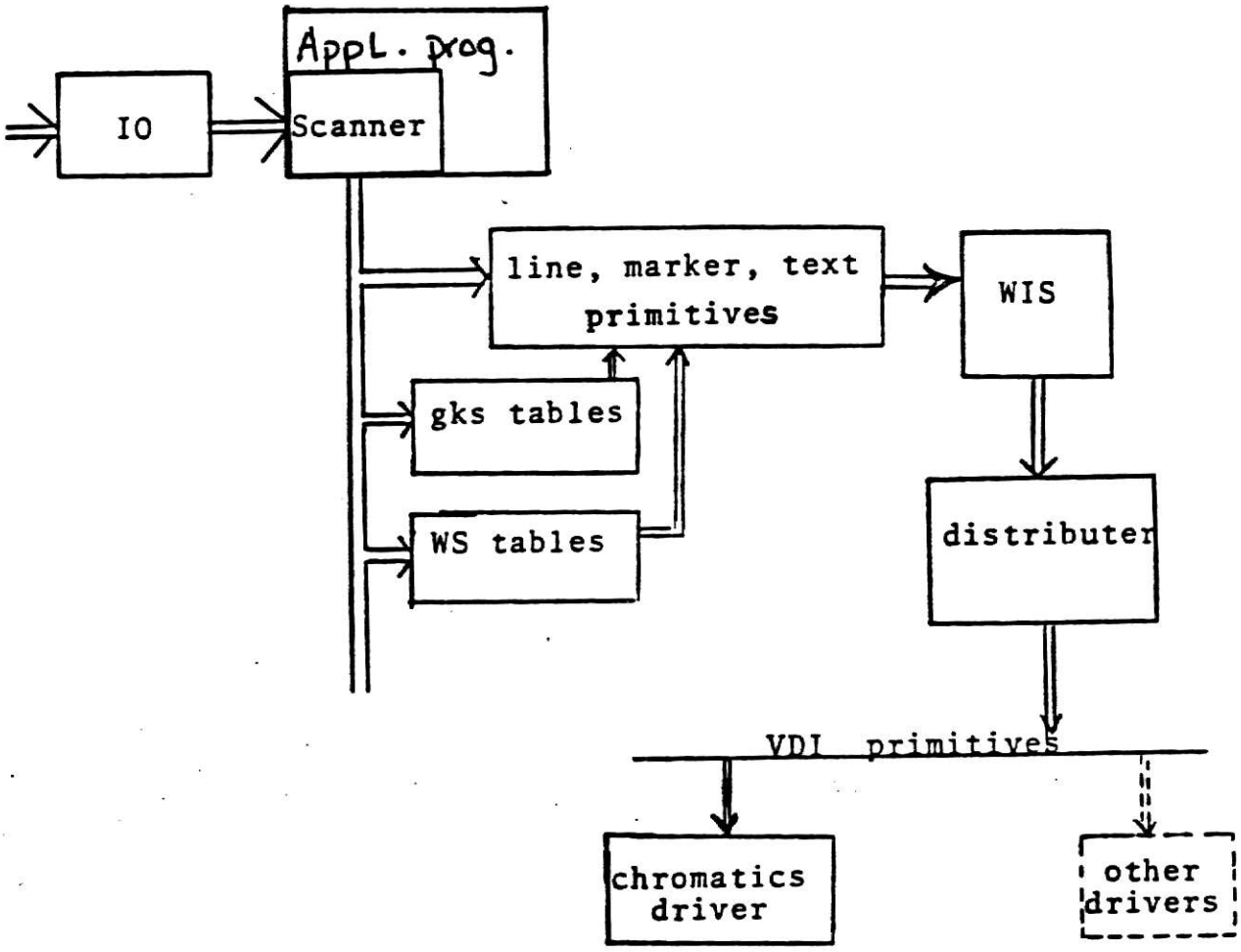


Figure 6

KSU-GKS DATA FLOW

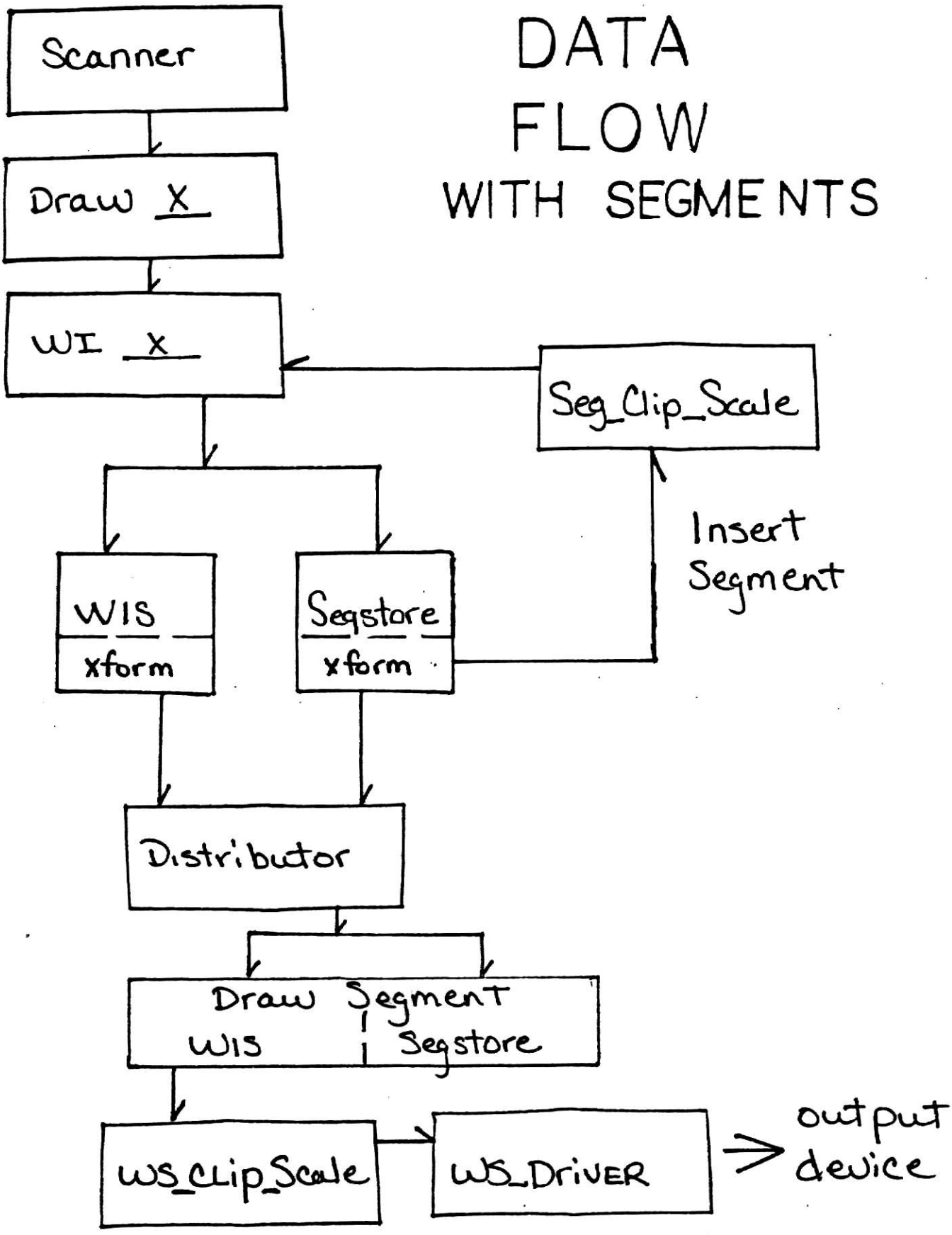

DATA
FLOW
WITH SEGMENTS
Scanner
Draw X
WI X
Seg_Clip_Scale
Insert
Segment
WIS
xform
Segstore
xform
Distributor
Draw Segment
WIS
Segstore
WS_Clip_Scale
WS_Driver
output
device

Figure 7

The "Draw" routines (DRAWLINE and DRAWMARKER) perform the normalization transformation from world coordinates to normalized device coordinates. Initially, the window clipping is also done here. This will be moved to the DISTRIBUTOR and applied against the primitives as they are processed for display.

When an output primitive is generated, the normalization transformation is applied to it, the current attribute table index is bound with it, and the WIS record is built. The attribute index cannot be changed, but the values in the bundle tables can be dynamically modified. This implementation is less rigid and allows the user more flexibility in generating pictures. It especially made sense when segmentation was added. When a picture is displayed, the individual segments will reflect the current attribute values in effect as they become a part of the picture.

At the point at which the primitive record would be put into the WIS structure, if a segment is open, the data flow is intercepted, and instead the record is put into the SEGSTORE. Refer to Figure 7, Data Flow with Segments. The "WI" routines (WILINE, and WIMARKER) accept primitives expressed in normalized device coordinates from the "DRAW" routines, refer to Appendix A. "WI" builds the appropriate primitive record. At this point the record would be put into the WIS (WDSS) if there

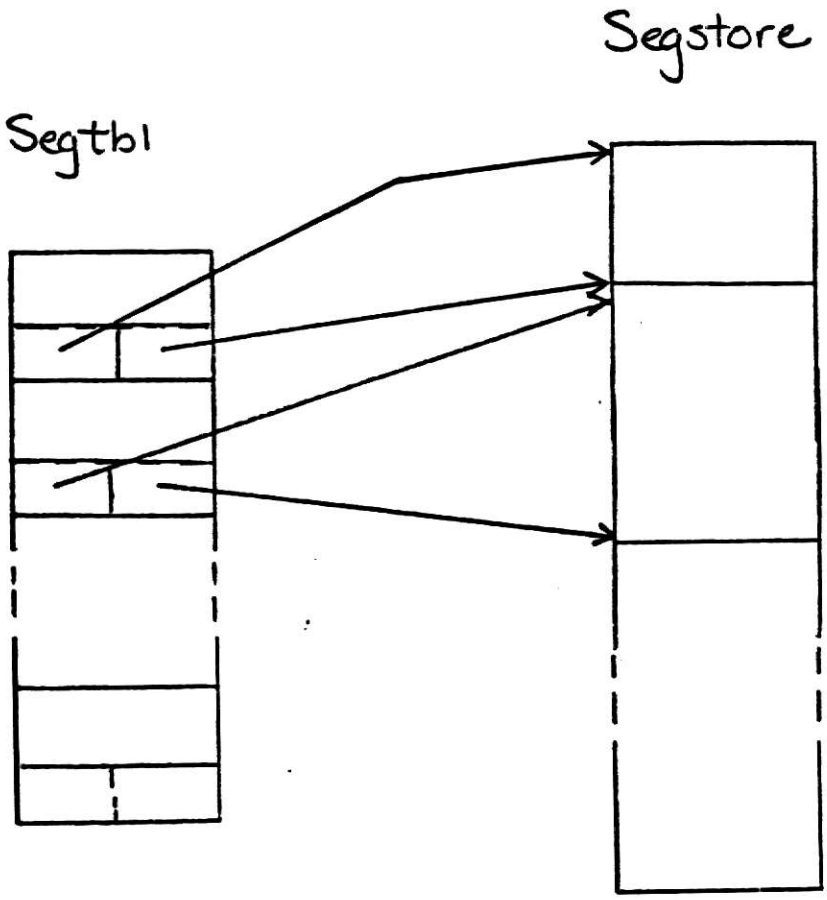


Figure 8

WISS Data Structure

is not an open segment. Otherwise, the record is put into the SEGSTORE (WISS).

The WIS and SEGSTORE are processed by the DISTRIBUTOR, refer to Appendix B. All non-segment primitives for a given workstation and all of its associated segments are processed for display. The DISTRIBUTOR binds each primitive with the attribute values currently in effect based on the stored attribute index. In addition, each segment's dynamic attributes are applied against the primitives of that segment. After the attributes are bound the primitive is translated into the Virtual Device Interface (VDI) format and sent onto the workstation device driver.

5.2 Data Structures

The WIS and the SEGSTORE are identical data structures. Refer to Appendix C for the data declarations. Both are implemented as simple non-partitioned arrays. SEGSTORE and WIS are declared as one dimensional arrays of primitive records. The GKS State List saves a pointer to the top of each array for easy access. This simplistic approach has the advantage of allowing variable sized segments composed of variable primitive records with no storage penalty. It was also advantageous to duplicate the WIS data structure for the SEGSTORE. Segment

primitive records and non-segment primitive records are identical. Segmentation is transparent to the routines that build them and to those that process them.

In addition to the SEGSTORE structure, a SEGTBL structure was declared as an array of fixed sized segment header records. The WISS actually consists of the SEGSTORE together with the SEGTBL. A segment header contains the segment state list which includes the segment's name, the current values of the segment's dynamic attributes, and a beginning and ending pointer (array index) to the primitive records for that segment in SEGSTORE. Figure 8, WISS Data Structure, shows the relationship between the components of the WISS, i.e., SEGSTORE and SEGTBL.

The management of both data structures is quite straight forward. New records are always appended to the top of the array using the index in the GKS State List. When a record is deleted, the hole is immediately compressed. There is no storage fragmentation or garbage collection.

5.3 Segment Manipulation Functions

It was not always feasible to implement the segment manipulation functions as defined in the GKS document. Those functions whose implementation deviated from the specification are discussed below. Refer to Appendix D for the various

segment functions

The CLOSE function did not specify a trap for null segments. A check was added to ensure that the segment had at least one primitive. If not, then the function backtracked and it was as if the segment had never been opened. Because segments cannot be edited, a null segment would have no purpose.

The DELETE_FROM_WS function that was implemented allows segments to exist in the SEGSTORE even when there is no explicit association with a workstation. This approach lends itself to the idea of a library of stored segments.

The COPY_TO_WS function was not implemented as such. It was felt that this approach resulted in duplicate object definitions and thus wasted valuable storage. Instead, a DRAW function was implemented. It is similar to a CALL_SEGMENT function. The DRAW request is treated as a primitive and stored in the WIS (or in the SEGSTORE as part of another segment) as another primitive record. DRAW segment calls are allowed to appear as a part of other segments, but a segment cannot call itself with DRAW. The DRAW is resolved by the DISTRIBUTOR at display time. When the call appears in either the WIS or SEGSTORE, the DISTRIBUTOR recursively calls itself and continues processing the primitives of the named segment.

5.4 Project Status

The GKS implementation at KSU was studied in its entirety before integrating segmentation capabilities into the existing system. The basic segment functions and the supporting data structures were implemented as described in preceding sections.

All of the segment capabilities in the GKS document were implemented and tested as an aggregate system. Due to the limited amount of time, the segmentation features were not tested in a fully integrated configuration. That is, segment capabilities were not tested in conjunction with output device drivers, text capabilities, or metafile capabilities.

```
;procedure wiline (lineindex:index; nolines:linerange
                  ; var araypoints:upoints);
  var
    i: integer;
    npoints : pointsrange;
  begin
    npoints := nolines + 1;
    if gksstate.openseg = 0 then
      begin
        if wistop < maxwielements then
          begin
            wistop := wistop +1;
            with wis[wistop] do begin
              opcode := lline;
              inx  := lineindex;
                n  := nolines;
              p0.ix := araypoints[1].ix;
              p0.iy := araypoints[1].iy;
              p1.ix := araypoints[2].ix;
              p1.iy := araypoints[2].iy
            end;  (* with wis[wistop] *)
          if ((npoints - 2) mod 3) <> 0
          then while (npoints + 1 < maxpoints) and ((npoints - 2) mod 3 <> 0)
            do begin (* fill araypoint to totally fill lcont records *)
               npoints := succ(npoints)
            ;  araypoints[npoints].ix := 0
            ;  araypoints[npoints].iy := 0
            end; (* fill araypoint to totally fill lcont records *)

            npoints := npoints - 2;
            i := 3;

            while ( (wistop< maxwielements) and (npoints>0) and
                    (i< (maxpoints-3)) ) do
              begin
                wistop := wistop +1;
                with wis[wistop]
                do begin
                  opcode := lcont;
                ; cp1.ix := araypoints[i].ix
                ; cp1.iy := araypoints[i].iy
                ; cp2.ix := araypoints[i+1].ix
                ; cp2.iy := araypoints[i+1].iy
                ; cp3.ix := araypoints[i+2].ix
                ; cp3.iy := araypoints[i+2].iy
                ; npoints := npoints -3
                ; i:= i+3
                end  (* with wis[wistop] *)
              end; (* while *)
            if wistop = maxwielements then begin end;
            if i = maxpoints then begin end
          end  (* if wistop < maxwielements *)
          else begin end
      end (* gksstate.openseg *)
```

```
  else
    begin
      if segtable[gksstate.openseg].segend < maxwielements then
        begin
          segtable[gksstate.openseg].segend := segtable[gksstate.openseg].segend +1;
          with segstore[segtable[gksstate.openseg].segend] do begin
            opcode := lline;
            inx  := lineindex;
              n  := nolines;
            p0.ix := araypoints[1].ix;
            p0.iy := araypoints[1].iy;
            p1.ix := araypoints[2].ix;
            p1.iy := araypoints[2].iy
          end;  (* with segstore[segtable[gksstate.openseg].segend] *)

          if ((npoints - 2) mod 3) <> 0
          then while (npoints + 1 < maxpoints) and ((npoints - 2) mod 3 <> 0)
             do begin (* fill araypoint to totally fill lcont records *)
                npoints := succ(npoints)
             ;  araypoints[npoints].ix := 0
             ;  araypoints[npoints].iy := 0
             end; (* fill araypoint to totally fill lcont records *)

          npoints := npoints - 2;
          i := 3;

          while ( (segtable[gksstate.openseg].segend< maxwielements) and (npoints>0) and
                  (i< (maxpoints-3)) ) do
            begin
              segtable[gksstate.openseg].segend := segtable[gksstate.openseg].segend +1;
              with segstore[segtable[gksstate.openseg].segend]
              do begin
                opcode := lcont;
              ; cp1.ix := araypoints[i].ix
              ; cp1.iy := araypoints[i].iy
              ; cp2.ix := araypoints[i+1].ix
              ; cp2.iy := araypoints[i+1].iy
              ; cp3.ix := araypoints[i+2].ix
              ; cp3.iy := araypoints[i+2].iy
              ; npoints := npoints -3
              ; i:= i+3
              end  (* with segstore[segtable[gksstate.openseg].segend] *)
            end; (* while *)
          if segtable[gksstate.openseg].segend = maxwielements then begin end;
          if i = maxpoints then begin end
        end  (* if segtable[gksstate.openseg].segend < maxwielements *)
        else begin end
    end (* else segment open *)
end  (* wiline *)
```

```
; procedure  distributor ( name   : drivers
                         ; update : updateset)

(* Purpose:  This routine calls drawprims to output primitives to the
         distdriver.  It calls drawprims for the prims in the Wis and
         for each segment that is associated with a particular work
         station.
*)


  ; var
       wsnum       : sindex      (* number of workstation *)
    ;  wisindex    : integer
    ;  distvdirec  : drvelement  (* record for distributor *)
    ;  distnum     : integer
    ;  sidx        : 0..maxseg
    ;  segnum      : 1..maxseg

  ;  begin (** procedure distributor **)
       for wsnum := 1 to minindex do        (* for all workstations *)
         if (wstable[wsnum].state = active) and    (* check ws *)
            ((ord(name) = wsnum) or (name = all))
          then begin
             if update = redraw  (* redraw completely or add to existing *)
               then begin (* lclear, start at top *)
                 distdriver(wsnum,vdiclear,distnum,distvdirec)
              ;  wisindex := 1
                    end
               else (* concat *)
                 wisindex := wstable[wsnum].wisptr  (* start at last position *)
    ;  drawprims (wis,wisindex,wistop,wsnum,idmatrix)

    ;  for sidx := 1 to wstable[wsnum].segsetidx do
       begin
          segnum := wstable[wsnum].wssetofstoredsegs[sidx]
       ;  if segtable[segnum].visible = vis then
             drawprims (segstore,segtable[segnum].segbegin,segtable[segnum].segend,
                    wsnum,segtable[segnum].transmatrix)
       end
    ;  distdriver (wsnum,vdisegend,distnum,distvdirec)
    ;  wstable[wsnum].wisptr := wistop + 1
    end     (* for each workstation *)
end        (* procedure distributor *)
    .
```

```
; procedure  drawprims ( prims         : wisegment
                       ; start, stop   : srange
                       ; wsnum         : sindex
                       ; transmatr     : matrixtype)
(* purpose:  this routine extracts primitives from the input wisegement
          (either wis or segstore) and information from wslinetable,
          wsmarkertable, wstesttable and gkstesttable.
          it then converts or packs the information if necessary and
           sends this information to the wsdriver.  The format
            of the call to the wsdriver is the workstation
            number, vdi command to respective workstation driver,
            an integer to contain the possible number of occurances
            (i.e number of lines) or the type of something (i.e
            colortype), and a record to contain the coordinates
            of a line or the coordinates of a textstring and
            the textstring.                                        *)

    ; var
        wisindex    : integer
      ; vdirecindx  : pointsrange
      ; vditrecindx : textcharrange
      ; distloc     : integer
      ; contwhat    : vdicmdtype
      ; distvdirec  : drvelement  (* record for distributor *)
      ; distnum     : integer
      ; segidx      : 0..maxseg
      ; trprim      : wielement

    ; begin (** procedure drawprims **)
        for wisindex := start to stop do begin (* wisindex loop *)

          trprim := prims[wisindex]
        ; transprim(trprim, transmatr)
        ; case trprim.opcode   (* send prim info by opcode type *)
        ......              ......

          ; lcall
            : begin  (* calling a segment *)
                segtblidx (trprim.segment, segidx)
              ; if segidx = 0 then
                  errorhandler (error, 122, 'drawprims           ')
                else
                  drawprims (segstore,segtable[segidx].segbegin,
                             segtable[segidx].segend, wsnum,
                             trprim.transmatrix)
              end  (* lcall case *)
          end  (* case of prim interpretation and sending to distdriver  *)
        end    (* for loop of wisindex through the prim records *)
    end  (* procedure drawprims *)
```

```
(*-----------------------------------------------
 beginning of graphics kernel const declarations
-----------------------------------------------*)
   ;  maxindex = 10
   ;  maxpoints = 50
   ;  maxlines =  51 (* one less than maxpoints*)
   ;  maxwielements = 200
   ;  maxstring = 41
   ;  maxunits = 32000
   ;  minindex = 5
   ;  namelen = 12  (* length of character name *)
   ;  maxseg = 10  (* maximum number of segments *)
   ;  emptystring = '                '

(*-----------------------------------------------
 end of graphics kernel const declarations
-----------------------------------------------*)
```

```
(*-------------------------------------
 graphics kernel type declarations
-------------------------------------*)
   ;  index  = 1..maxindex
   ;  sindex = 1..minindex
   ;  matrixtype = array [1..2, 1..3] of real (* transformation matrix *)
   ;  srange = 1 .. maxwielements
   ;  linerange     = 1..maxlines
   ;  namerng = 1..namelen
   ;  unit = 0..maxunits
   ;  regenset = ( suppressed, allowed)
   ;  frameset = ( no, yes )
   ;  visset = ( vis, invis )
   ;  highset = ( high, norm )
   ;  detectset = ( detect, undetect )
   ;  gkslevset = ( gkcl, gkop, wsop, wsac, sgop )
   ;  name = array [namerng] of char
   ;  wsrec =
         record
            defined  : boolean
         ;  state    : wsstateset
         ;  wisptr   : integer
         ;  kind     : wskindset
         ;  devices  : array [sindex] of devrec
         ;  devmode  : wsdefset
            (* set of seegments associated with WS *)
         ;  wssetofstoredsegs  : array [1..maxseg] of 1..maxseg
         ;  segsetidx  : 0..maxseg  (* index to above array *)
         ;  regenmode  : regenset
         end (* wsrec *)

   ;  tstring = array [1..maxstring] of char
```

```
  ;  wielement = record
                   case opcode: optype of
                     ltop :   (a: integer)

                    ;lend :   (b: integer)
                     (* drawseg *)
                    ;lcall:   (segment : name
                               ;transmatrix : matrixtype)
                    ;lline:   (inx    : index
                               ;n     : linerange
                               ;p0,
                                p1    : ipoint)
                    ;lmarker: (minx   : index
                               ;mn    : markerrange
                               ;mp0   : ipoint)
                    ;ltext:   (tinx   : integer
                               ;tcn   : textcharrange
                               ;tp0   : ipoint
                               ;tstr  : tstring )
                    ;lclear:. (g      : integer)
                    ;lcont:   (cp1,
                               cp2,
                               cp3    : ipoint)
                  end  (* wielement *)

  ;  wisegment = array [1..maxwielements] of wielement

     (* segment state list *)
  ;  segrec =
        record
           segname        : name
        ;  assocws        : array [sindex] of sindex
        ;  wssetidx       : 0..minindex
        ;  transmatrix    : matrixtype
        ;  visible        : visset
        ;  highlight      : highset
        ;  segpriority    : real
        ;  detectable     : detectset
        ;  segbegin       : srange
        ;  segend         : 0..maxwielements
        end (* segrec *)

     (* GKS state list *)
  ;  gksrec =
        record
           openws      : array [sindex] of drivers
        ;  activews    : array [sindex] of drivers
        ;  openseg     : 0..maxseg
        ;  levelgks    : gkslevset
        end (* gksrec *)
(* ----------------------------------------
   end of graphics kernel type declarations
   ---------------------------------------- *)
```

```
(*-------------------------------------
 graphics kernel var declarations
 -------------------------------------*)
   ;  curlineindex   : index
   ;  curmarkerindex : index
   ;  curtextindex   : index
   ;  curwinindex    : index
   ;  gkstexttable : tgkstexttable
   ;  gkswintable  : tgkswintable
   ;  clipind : cliptype
   ;  wis     : wisegment (* non-segment, WS-dependent storage *)
   ;  wistop  : integer  (* pointer to last entry in WIS *)
   ;  disttop : integer
   ;  wisindex : integer
   ;  wslinetable   : twslinetable
   ;  wsmarkertable : twsmarkertable
   ;  wstexttable   : twstexttable
   ;  wsviewtable   : twsviewtable
   ;  wstable       : twstable
   ;  textstring  : tstring
   ;  gksstate : gksrec  (* GKS state list *)
   ;  segtable : array [1..maxseg] of segrec (* segment state lists *)
   ;  segstore : wisegment (* segment storage *)
   ;  idmatrix : matrixtype (* identity matrix for transformations *)

(* ---------------------------------------
   end of graphics kernel var declarations
   --------------------------------------- *)
```

```
(*  segtblidx - translates the segment name into the
       corresponding index into segtable.
       if no such named segment exists segtblidx returns 0.
*)
;procedure segtblidx (sname : name; var segidx : integer)
;var count1 : integer
;    ofile : text
;begin
segidx := 0
;for count1 := 1 to maxseg do
     begin with segtable [count1] do
          if segname = sname then segidx := count1
     end
end


(*  settrans - stores the given transformation matrix
        in the segment record for the named segment.
*)
;procedure settrans (sname:name;matr:matrixtype)
;var sidx : integer
;    ofile : text
;begin
segtblidx(sname, sidx)
;if sidx =0 then errorhandler(ofile, 122,'settrans')
           else segtable[sidx].transmatrix := matr
end


(*   setseghgh - set the segment highlight attribute.
*)
;procedure setseghgh(sname:name;hattr:highset)
;var sidx : integer
;    ofile : text
;begin
segtblidx(sname, sidx)
;if sidx = 0 then errorhandler(ofile, 122,'setseghgh')
             else segtable[sidx].highlight := hattr
end
```

```
(* setsegd - set segment detectable attribute.
*)
;procedure setsegd (sname : name; dattr : detectset)
;var sidx : integer
;    ofile : text
;begin
segtblidx(sname, sidx)
;if sidx = 0 then errorhandler(ofile, 122,'setsegd')
            else segtable[sidx].detectable := dattr
end


(*  closeseg - close the currently open segment.
        discard if segment is null.
*)
;procedure closeseg
;var count1 : integer
;    ofile : text
;begin
if gksstate.levelgks = sgop
    then begin
         if segtable[gksstate.openseg].segend < segtable[gksstate.openseg].segbegin
             then begin
                  for count1 := 1 to 10 do
                      segtable[gksstate.openseg].segname[count1]:=' '
                  end
          ;gksstate.levelgks := wsac
          ;gksstate.openseg := 0
         end
    else begin
         errorhandler(ofile, 4,'closeseg')
         end
end
```

```
(*  drawseg - put display_segment request in wis.
       segment is transformed just before display.
       a request for an undefined segment is allowed at this point.
       segment must be defined before displaying.
*)
;procedure drawseg (sname : name; matr : matrixtype)
;var sidx : integer
;    ofile : text
;begin
if not (wistop < maxwielements)
   then errorhandler(ofile, 302, 'drawseg')
   else begin
        wistop := wistop+1
       ;with wis[wistop]
          do begin
             opcode := lcall
            ;segment := sname
            ;transmatrix := matr
             end
       ;segtblidx(sname, sidx)
       ;if sidx = 0 then errorhandler(ofile, 124, 'drawseg')
        end
end


(* copyseg - copy named segment to wis.
*)
;procedure copyseg (sstart, sstop, tto : integer ; var arr : wisegment)
;var i, top : integer
;    ofile : text
;begin
  top := tto
 ;for i := sstart to sstop
      do begin
         top := top + 1
        ;arr[top] := segstore[i]
         end
end
```

```
(*  wsdeleteseg - disassociate named segment from specified ws.
      remove segment name from wssetof storedsegs.
      remove ws id from assocws.
*)
;procedure wsdeleteseg (wsid : drivers; sname : name)
;var here, i, sidx : integer
;    ofile : text
;    wsnum : sindex
;begin
wsnum :=3
;segtblidx (sname, sidx)
;if sidx = 0 then errorhandler (ofile, 122,'wsdeleteseg')
             else begin
                  here := 0
                  ;with wstable[wsnum]
                    do begin
                       for i := 1 to segsetidx do
                         if wssetofstoredsegs [i] = sidx then here := i
                      ; if not (here = 0)
                           then begin
                            if not (here = segsetidx)
                              then
                               for i := here to segsetidx -1
                                do wssetofstoredsegs[i]:=wssetofstoredsegs[i+1]
                              else begin end
                            ;segsetidx := segsetidx -1
                           end
                           else begin end
                    end
                 end
end
```

```
(* insrtseg - if open segment, then append named segment
      to current segment else append named segment to wis.
*)
;procedure insrtseg(sname : name ; matr : matrixtype)
;var sidx : integer
;    sstart, sstop, tto : integer
;    ofile : text
;begin
 segtblidx(sname, sidx)
; if sidx = gksstate.openseg
    then errorhandler(ofile, 125,'insrtseg')
    else begin
   ; if sidx = 0
     then errorhandler(ofile, 124,'insrtseg')
     else
          begin
          sstart := segtable[sidx].segbegin
         ;sstop := segtable[sidx].segend
         ;if gksstate.levelgks = sgop
             then begin
                  tto := segtable[gksstate.openseg].segend
                 ;copyseg(sstart, sstop, tto, segstore)
                  end
             else begin
                  tto := wistop
                 ;copyseg(sstart, sstop, tto, wis)
                  end
          end
     end
end
```

```
(* The followimg functions were inplemented by D. A. Bernadis.
   The documentation for these routines appear in her report. *)

; procedure createseg ( segname : name )
  ; begin end

; procedure renameseg ( osegname : name
                      ; nsegname : name )
  ; begin end

; procedure deleteseg ( segname : name )
  ; begin end

; procedure assocseg ( wsname : drivers
                     ; segname : name )
  ; begin end

; procedure setsegvis ( segname : name
                      ; vattr : visset )
  ; begin end

; procedure setsegprty ( segname : name
                       ; pattr : real )
  ; begin end
```

REFERENCES

Bail83 Baily, C., "Graphics standards are emerging - slowly but surely." , Electronics Design , Vol 31, No.20 (January 1983), 103-110.

Berg78 Bergeron, R.D., Bono, P.R., and Folly, J.D., "Graphics Programming using the Core System." , ACM Computing Surveys , Vol.10, No.4 (December 1978), 389-443.

Bono82 Bono, P.R., et.al., "GKS - The First Graphics Standard." , Computer Graphics and Applications , Vol.2, No.5 (July 1982), 9-23.

Butt81 Buttuls, P., "Some Criticisms of the Graphical Kernal System (GKS)." , Computer Graphics , Vol.15, No.4 (December 1981), 302-305.

Cahn79 Cahn, Deborah, et.al., "A Response to the 1977 GSPC Core Graphics System.", Computer Graphics, Vol.13, No.2, (August 1979), 57-62.

Enca80 Encarnacao, J., et.al., "The Workstation concept of GKS and the resulting conceptual differences to the GSPC Core System." , Computer Graphics , Vol.14, No.2 (July 1980), 226-229.

Ewal78 Ewald, R.H. and Fryer, R., "Final Report of the GSPC State-of-the-Art Subcommittee.", Computer Graphics, Vol.12, Nos.1-2, (June 1978).

Fich83 Fichera, Richard, "Graphics Compatibility.", Mini-Micro Systems, (July 1983), 189-194.

Fole82 Foley, J.D. and VanDam, Andries, Fundamentals of Interactive Computer Graphics , Addison-Wesley Publishing Co., 1982.

Gilo78 Giloi, Wolfgang K., Interactive Computer Graphics Data Structures, Algorithms, Languages , Printice-Hall, Inc., 1978.

Hatf82 Hatfield, Lansing and Herzog, Bertram, "Graphics Software - From Techniques to Principles.", IEEE Computer Graphics and Applications, (January 1982), 59-80.

Magn81 Magnenat-Thalmann, Nadia and Thalmann, Daniel, "A Graphical Pascal Extension Based on Graphical Types.", Software: Practice and Experience, Vol.11, (January 1981), 53-62.

Newm79 Newman, W.M. and Sproull, Robert F., Principles of Interactive Computer Graphics , Second Edition, Mcgraw-Hill Book Co., 1979.

Nico81 Nicol, C.J. and Kilgore, A.C. "A Pascal Implementation of the GSPC Core Graphics Package." , Computer Graphics , Vol.15, No.4 (December 1981), 327-335.

Orr82 Orr, Dr. Joel N., "How Graphics Systems Get the Picture." , Mini-Micro Systems, (July 1982), 195-198.

Perr83 Perry, Burt, "Graphics Frees Itself From Device-Dependence.", Electronic Design, Vol.31, (January 20, 1983), 167-173.

Prin71 Prince, David M., Interactive Graphics for Computer-Aided Design , Addison-Wesley Publishing Co., 1971.

Shre82 Shrehlo, Kevin B., "ANSI graphics standards coalesce around international Kernel.", Mini-Micro Systems, (November 1982), 175-186.

Rose83 Rosenthal, S.H., "Managing Graphical Resources.", Computer Graphics, Vol.17, No.1, (January 1983), 38-45.

VanT82 VanTilborg, A.M., "Executing Large Graphic Programs with Small Computers." , Software: Practice and Experience, Vol.12, (October 1982), 915-927.

Warn82 Warner, Jim, "Device Independence and Intelligence in Graphics Software.", Computer Technology Review, Vol.2, No.2, (Summer 1982), 117-120.

Warr83 Warren, Carl, "ANSI answering concerns over proposed graphics communication standards.", Electronic Design, (September 2, 1983), 55-56.

Draf83 "Draft International Standard ISO/DIS 7942.", Information Processing Graphical Kernal System (GKS) Functional Description, (January 17, 1983).

Stat77 "Status Report of the Graphic Standards Planning Committee of ACM/SIGGRAPH.", Computer Graphics Quarterly Report of ACM-SIGGRAPH, Vol.11, No.3, (Fall 1977), Part I.

IMPLEMENTATION OF DATA SEGMENTATION FOR A GKS BASED GRAPHICS SYSTEM

by

REBECCA EDWARDS MAY

B.S. Appalachian State University, 1978

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

ABSTRACT

The paper begins with an overview of the GKS specification to give the reader an idea as to the type of functions described by GKS. A discussion of data segmentation, the idea of logically grouping graphical primitives, preceeds a detailed discussion on the GKS requirements for data segmentation. The GSPC Core design for segments is compared to the GKS design.

Several concepts on segmentation that are not well defined or reasonably justified in the GKS specification are discussed in more depth here: binding of primitive attributes, and permanent segment libraries. Finally, a specific implementation of GKS is given as an example. The design of that system is discussed, focusing on the segmentation design and the issues that lead to that design.