AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 205/82 AUGUSTUS

D.S.H. ROSENTHAL

MANAGING GRAPHICAL RESOURCES

Preprint

CR Categories and Subject Descriptors: I.3.4 [Computer Graphics]: Graphics Utilities — graphics packages; I.3.6 [Computer Graphics]: Methodology and Techniques — device independence; interaction techniques

General Terms: Design, Standardization

Managing Graphical Resources

by

David S. H. Rosenthal

ABSTRACT

The development of interactive graphics software has separated into two apparently incompatible streams. The mainstream, exemplified by GKS and the Core, and a “RasterOp” stream, exemplified by Smalltalk and the Carnegie-Mellon CANVAS package. In the mainstream, the important concepts are viewing, segments, output primitives, and virtual input devices. In the RasterOp stream, the important concepts are windows, the refresh hierarchy, and pointing for input.

Although it is easy in the mainstream to provide libraries of output-only routines for general use, it is difficult to do the same for input. Although it is possible to write device-independent interactive applications using mainstream techniques, the quality of the user interface may vary wildly as they are ported. These problems are caused by the mainstream's lack of the mechanisms needed to control the allocation of real graphical resources within the application:

1. No hierarchy of pictures on the view surface. All parts of the application have control over all parts of the view surface.
2. No connection between viewing and segmentation. The viewing parameters used to create a segment are not attributes of the segment.
3. All parts of the application have access to all the input devices. It is up to each part to decide if an input is relevant, or if not, what to do with it.

This was one of the invited topical papers at the ACM-SIGGRAPH Workshop on Graphics Input Interaction at Seattle, Wash. on June 2-4th 1982. It appears in the issue of *Computer Graphics* devoted to the Workshop.

1. Introduction

"Whenever I'm caught between two evils, I take the one I haven't tried before."

Mae West.

The development of interactive graphics software has separated into two apparently incompatible streams. The mainstream, exemplified by GKS[5] and the Core[11], and a "RasterOp" stream, exemplified by Smalltalk[6] and the Carnegie-Mellon CANVAS package[2]. In the mainstream, the important concepts are viewing, segments, output primitives, and virtual input devices[7]. In the RasterOp stream, the important concepts are windows, the refresh hierarchy, and pointing for input.

The differences between the streams may seem to reflect the differences in the hardware they address, but the example of a window-manager graphics system driving storage tube terminals in Edinburgh refutes this[9]. The RasterOp stream is attempting to provide facilities whereby the graphical resources can be *managed*; whereby routines and sub-programs can be provided with controlled access to part of the resources controlled by their caller. The Seillac II workshop[4] stressed the importance of building interactive applications from existing components. It is only in the context of graphical resource management that this can successfully be done, since the invoker of an existing component can be confident that it will only access the resources it has been given.

The mainstream, the RasterOp stream, and the storage-tube window manager will now be described in a common format, and then assessed against the goal of resource management. Finally, the design principles for a package based on resource management will be presented.

If standards proposals are to succeed, they must be based on codifying the "best current practice" and not on radical innovation, no matter how inviting. However, "best current practice" is constantly changing as hardware and software evolve. The critique of the models underlying current standards proposals that follows is not to be interpreted as advocating their rejection, but rather as an attempt to discover the directions in which current practice will develop.

2. The Mainstream

The concepts forming the backbone of the mainstream graphics packages are powerful and seductive abstractions, but are strangely unrelated to one another. At the start of the standards effort, this *orthogonality* was highly prized for the simplicity and structure it gave to the documents, but in the development since then it has been overridden by other criteria.

2.1. Transformations

The concept of viewing was introduced at Seillac I as the fundamental criterion for dividing the functions of a graphics package from those of the rest of the world. Graphics was viewing, everything else was modelling. It was based on the *synthetic camera analogy*, with a scene defined in a single world coordinate space being viewed as if by a camera located at a single point in the space (Fig. 1).

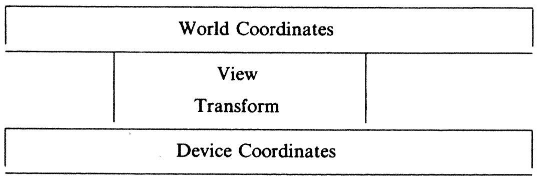


Fig. 1 — Synthetic Camera Analogy

This simple analogy has since broken down in two directions. The two breakdowns have been seen as independent, but have now completely vitiated the power of the analogy.

The stated objective of the Core was to provide program (and programmer) portability between devices. The minimal interpretation of this was that it must be possible for the same program, at different times, to drive different devices. However, the implicit intent of the Core was to make it possible for the same program to drive different devices *simultaneously*. To support this, it was necessary to split the single transformation from world to device coordinates into a two-stage process. Coordinates were first transformed by a single global transformation to a *normalised device coordinate* space, and then by one of possibly many device-specific transformations to device space (Fig. 2).

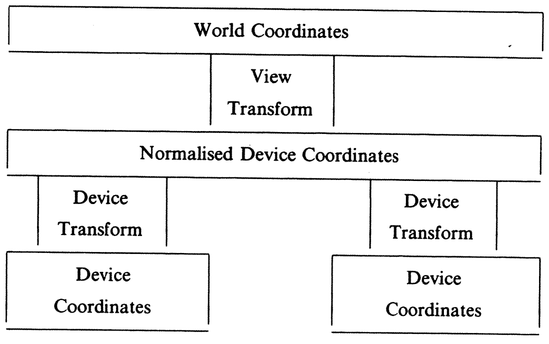


Fig. 2 — CORE Transformations

There are almost no graphical applications in which the only pictures generated consist of a single view of a single object occupying the whole view surface. Thus, pictures will really be generated using several "world" coordinate systems. Consider a drawing system in which part of the screen shows a small view of the whole drawing, part an expanded view of the area being manipulated, part some symbols from a symbol library, part some menu items, and so on. If the system embodies only a single viewing transformation, the application must laboriously re-create the correct viewing transformation for the part of the display it wishes to modify. Further, a single transformation system places the burden of re-transforming locator input coordinates from NDC space to the space used by the application entirely on the application. These requirements led GKS to include multiple normalisation transformations (Fig. 3)[10].

2.2. Segmentation

The concept of segmentation was introduced to avoid the problem of supplying names for individual primitives to identify them for post-generation manipulation. It also addressed the problem of refreshing the picture after changes, storing a description of the

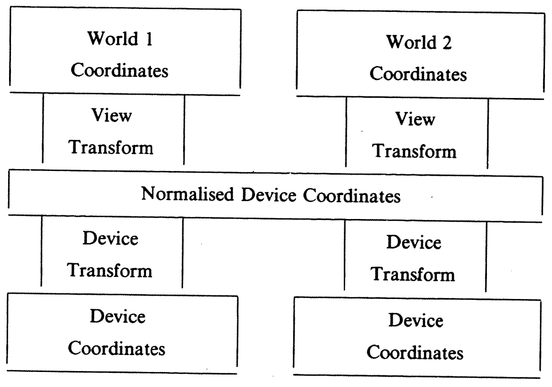


Fig. 3 — GKS Transformations

picture as output primitives in a segmented display file. The range of permitted manipulations was modelled on the capabilities of a typical refresh vector display, highlighting, visibility and detectability changes, and segment transformations.

Segments are created at will, initially in an *open* state. At most one segment may be open. Primitives output while a segment is open become part of it, and are then subject only to changes of the segment attributes until the segment is eventually destroyed.

2.3. Attributes

Mainstream packages support one or more of the following types of attribute for primitives and segments:

1. Global attributes, applicable to appropriate primitives as they are output and not subject to retro-active modification. For example:

   SET LINE STYLE(DOTTED)

2. Segment attributes, applicable to each segment and subject to retro-active modification. For example:

   SET HIGHLIGHTING(SEG,ON)

3. Workstation attributes, generally selected from a table by a global index attribute, applicable to appropriate primitives, and subject to retro-active modification. For example:

   SET COLOUR MAP(WS,INDEX,COLOUR)

2.4. Logical Input Devices

Mainstream input facilities are built round the logical input device concept. Initially, this specified that each workstation had several logical input devices, divided into *classes*, according to the type of value they return (Fig. 4). Typical classes are LOCATOR, returning a position (and, for GKS, a normalisation transformation ID — Fig. 5), VALUATOR, returning a real number, and PICK, returning a segment name and a pick identifier.

	Classes	
LOCATOR	→	Position (WC) + Transformation ID.
STROKE	→	Positions (WC) + Transformation ID.
VALUATOR	→	Real Number
CHOICE	→	Integer
PICK	→	Segment Name + Pick ID.
STRING	→	Characters
	Modes	
REQUEST	→	*REQUEST LOCATOR()*
SAMPLE	→	*SAMPLE LOCATOR()*
EVENT	→	*AWAIT EVENT()* *GET LOCATOR()*
	THEREFORE	
	Eighteen different functions to obtain input.	

Fig. 4 — GKS Input

The concept has been refined in several ways[10], the most obvious of which are:

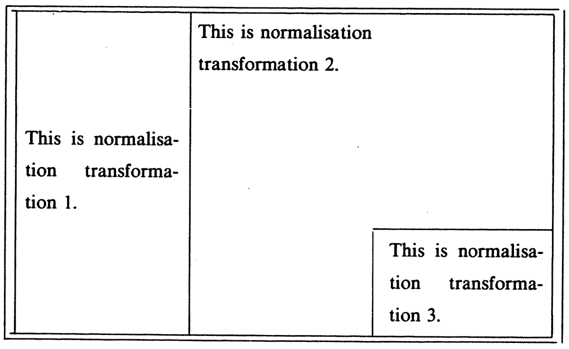

This is normalisation transformation 2.
This is normalisation transformation 1.
This is normalisation transformation 3.

Fig. 5 — GKS Locator Input

1. Each device can operate in different *modes*, specifying how the value is obtained from the device. Typical modes are REQUEST, in which the application is suspended until input is available, SAMPLE, in which the device is polled for its current value, and EVENT, in which the device creates *event records* describing its value and adds them to an input queue.
2. Each device has *attributes*, including parameters for its echo implementation, initial values, restrictions on its value, and so on. These attributes in part provide the link with output, at least at the device level.

3. The RasterOp Stream

The consensus established by the standards effort makes it possible to pontificate about the mainstream with confidence. No such consensus exists in the RasterOp stream; Gene Ball's CANVAS package[2] is used as an example. It was developed at Carnegie-Mellon, as part of the SPICE project, in Pascal. Other systems have been developed in more exotic languages, such as Smalltalk[6, 13], and LISP[12], and are thus less easy to contrast with mainstream packages.

3.1. Transformations

The application using CANVAS creates output in an abstract (integer) coordinate space called a *canvas*. New canvases may be created at will; there is no implementation-defined limit on their number as there is with GKS' normalisation transformations. All output operations must quote a descriptor for a canvas; canvases are always available for output irrespective of whether the output would be visible. Contrast this with GKS' concept of a *current* normalisation transformation, applied to output until another is selected.

A canvas, and the graphics sent to it, become visible as the result of a two-stage process. First, the canvas must be mapped to a *viewport*. Viewports describe rectangular patches of the view surface. Canvas (=world) coordinates are mapped to viewport (=device) coordinates either by scaling or by clipping a specified range of canvas coordinates to the available range of device coordinates. This mapping corresponds to the mainstream's window/viewport transformation.* A viewport may display at most one canvas, but a canvas may be mapped simultaneously to many different viewports. Multiple canvases are analogous to multiple normalisation transformations; multiple viewports for a single canvas are analogous to multiple workstations.

Each viewport provides access to a specific rectangle of display space. These rectangles may overlap, and thus several different canvases may each try to control the state of a particular pixel. CANVAS resolves these conflicts by organising the active viewports into a hierarchy called the *Refresh Tree*. At the root of this tree is a special, system-provided viewport giving access to the whole screen. A viewport in the tree may have any number of children, each of which may lay claim to any part of the display, but they will only have visible effect within the part of the display controlled by their parent. Thus, a primitive output to a canvas C that is mapped to a viewport V will be visible in that viewport if the canvas coordinate P_c is mapped to a viewport coordinate P_v which lies within each of the sequence of viewports:

$$V, parent(V), parent(parent(V)), \cdots RootViewport.$$

If a viewport has more than one child, they may also lay claim to the same pixels. This conflict is resolved by introducing a precedence order among the children of one parent (siblings). Viewports may be either *transparent* or *opaque*. Opaque viewports obscure any siblings of lower precedence they overlap, whereas transparent viewports share access to their pixels with their lower precedence siblings (Fig. 6).

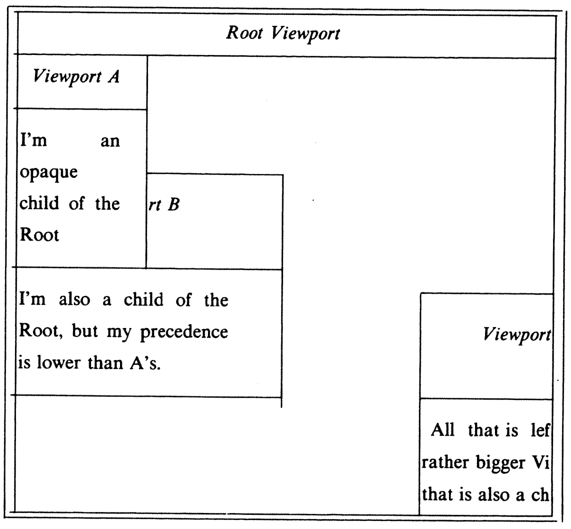

Root Viewport
Viewport A
I'm an opaque child of the Root
rt B
I'm also a child of the Root, but my precedence is lower than A's.
Viewport
All that is lef
rather bigger Vi
that is also a ch

Fig. 6 - The Refresh Tree

Positioning a viewport in this tree requires access to its parent. Thus, control over part of the view surface may safely be delegated, by passing a viewport descriptor. The possessor of this descriptor will be able to create a sub-tree depending from this "root", but cannot affect the root's position in the tree, nor affect any part of the view surface outside the root's parent.

In this way, CANVAS provides a mechanism for controlling the allocation of the scarce view surface resource among the uses that need to share it. Programs may safely call other programs or modules and pass a viewport descriptor, confident that neither a

* Except that the scale/clip choice is made independently for each axis.

passive nor an interactive callee can affect other parts of the view surface.

3.2. Segmentation

In the mainstream, segments serve two purposes; to name parts of the display (groups of primitives) for manipulation, and to store a picture description for future regenerations. CANVAS does not support a segment concept, but in some ways the combination of the canvas and viewport concepts play the same rôle.

Just as segments have visibility and detectability attributes, so viewports may be placed into and removed from the refresh tree. Just as segments may be highlighted, RasterOps may be performed within canvases, for example to invert a region. On the other hand, unlike segments, canvases are always open, and are associated with particular patches of the view surface.

CANVAS does not support a description of the picture as primitives at all. A viewport may (but need not) store the bitmap representation of the parts of other viewports it obscures, and restore them when they are obscured no longer. A viewport may (but need not) store its own complete bitmap representation, and automatically refresh parts no longer obscured. No storage other than bitmaps is provided. At higher levels, the picture is defined *procedurally*; the system arranges, via the input mechanism, for appropriate application code to be invoked when regeneration is required.

3.3. Attributes

CANVAS supports only a single type of attribute, applied to all appropriate primitives sent to a particular canvas. For example:

SetColor(c : Canvas ; ink : Color)

Thus each canvas provides its own attribute context; a callee can be invoked with a canvas descriptor in the knowledge that it cannot affect other canvas' attributes.

Note, however, that the management of the view surface (by viewports) is separate from the management of the attributes (by canvas). These "call-by-value" facilities can be provided independently for either.

3.4. Input

CANVAS provides a single input class, the *KeyEvent*, into which all physical device inputs are mapped.

```
KeyEvent = packed record
        Cmd:  0..255;
        Ch:   char;
        X,Y:  integer
end;
```

Each canvas has an input queue, containing zero or more of these events. When a KeyEvent is generated, it is added to the tail of the queue of the canvas mapped to the *deepest* viewport containing the position (Fig. 7).

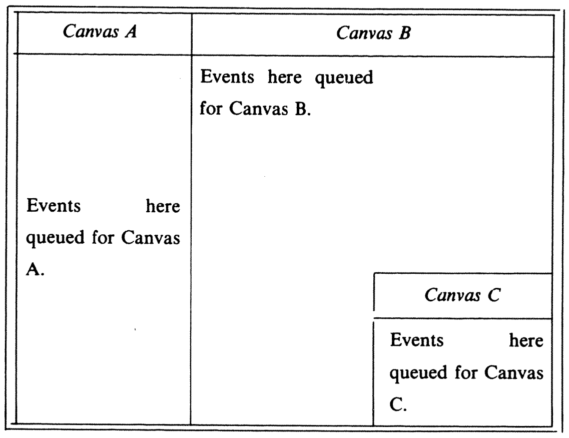

Canvas A
Canvas B
Events here queued for Canvas B.
Events here queued for Canvas A.
Canvas C
Events here queued for Canvas C.

Fig. 7 — Canvas Input

Each canvas also has a *translation table*, loaded from a file at run-time and used to map from the operator's keystrokes and button pushes to application-specific commands. For example, the editor interprets

Cmd = cDELPREVWD

as "delete the previous word". It is unaware that the translation table for its canvas specifies that this command is generated by pressing the <CTRL><SHIFT>W keys.

Thus, the only "device" an application sees is its canvas. The details of how the physical devices generate KeyEvents in its queue are of no interest; indeed it has no way of addressing "devices" at all. Relevant inputs for the application are directed to it by pointing at any of the viewports displaying its canvas; irrelevant inputs point to some other canvas' viewport and are never seen.

CANVAS provides two input modes; EVENT and SAMPLE. It also uses the event queue more generally than its name implies, permitting the application to push events back, and generate events itself, to assist in parsing, and also permitting the system to synthesise events, for example to request picture regeneration. SAMPLE mode might be handled more elegantly in this way too, by allowing the application to trigger the system into queuing an event without operator action.

A canvas may be placed in *forwarding* mode, when all events directed at it will be passed up the refresh tree to its parent. The program pulling an event from a canvas' queue may use this if it decides the request cannot be handled at this level, setting forwarding mode and pushing the event back will send it to the parent.

4. Doing Without RasterOp

It might be thought that the structure of CANVAS was peculiar to those systems with RasterOp, but this is not so. Some time ago I wrote an experimental package in C for UNIX,* called **GiGo**[9]. It was based on my reactions to GKS, and on ideas from DLISP[12] and CURSES[1]. The intention was to produce a graphics system with as few underlying concepts as possible.

* UNIX is a Trademark of Bell Laboratories.

4.1. Transformations

Scanning the list of concepts in GKS, several were obviously superfluous. The first was *normalised device coordinates*. This left only a single level of transformation, mapping between world coordinates, defined by the application:

```
typedef struct {
        float   n_x, n_y;
        } Ncoord;
```

and device coordinates, defined by the hardware:

```
typedef struct {
        int     d_x, d_y;
        } Dcoord;
```

The starting point for the Window concept thus became a transformation between application and device coordinates:

```
typedef struct {
        Dcoord w_lo, w_hi;
        Ncoord w_ll, w_ur;
        } Window;
```

Our Tektronix terminals have slave screens, so the idea that there were multiple view surfaces was natural. A Screen data type was needed, and a way of keeping track of which view surface the viewport was on:

```
typedef struct {
        Dcoord w_lo, w_hi;
        Ncoord w_ll, w_ur;
        Screen *w_scrn;         /* Active on */
        } Window;
```

This represented a window/viewport transformation, and therefore a patch of view surface, so there were going to be many of them. A list of active ones was needed:

```
typedef struct WINDO {
        struct WINDO *w_next;          /* Link */
        Dcoord w_lo, w_hi;
        Ncoord w_ll, w_ur;
        Screen  *w_scrn;
        } Window;
```

4.2. Segments

If a Window corresponded to a patch of view surface, could it not also correspond to the graphics in that patch? The concepts of segments and segment storage vanished, and the Window structure grew a bit:

```
typedef struct WINDO {
        struct WINDO *w_next;
        Dcoord w_lo, w_hi;
        Ncoord w_ll, w_ur;
        Screen  *w_scrn;
        Prim    *w_prim;        /* Display File */
        } Window;
```

Holding display files in the memory of a PDP-11 is simple but impractical for complex pictures. The alternative is to permit the application to define the picture *procedurally*:

```
typedef struct WINDO {
        struct WINDO *w_next;
        Dcoord w_lo, w_hi;
        Ncoord w_ll, w_ur;
        Screen  *w_scrn;
        Prim    *w_prim;
        int     (*w_outp)();    /* Redraw */
        } Window;
```

If *w_outp* is not NULL, it points to a routine capable of re-creating the picture in the Window. If it is NULL, then a list of primitives is stored with the Window, ready to be interpreted by the system to re-create the picture.

This far, Windows provide for the rôles of segments as manipulatable parts of the picture, and as picture stores for regeneration. They can also play the rôle of segment as symbol, a picture component ready to be incorporated in future pictures. All that is needed is to allow Windows not to be active on any Screen, with *w_scrn* being NULL. These Windows represent potential, instead of actual, patches of view surface. An additional primitive is then provided, a *reference* to a window, giving multi-level segmentation (cf. GKS' INSERT SEGMENT).

4.3. Attributes

All **GiGo**'s attributes are Window attributes; both "segment" attributes such as background colour, and "primitive" attributes such as current linestyle. They are all stored in the Window structure, and no manipulation of a Window's attributes can affect the attributes of another Window.

4.4. Input

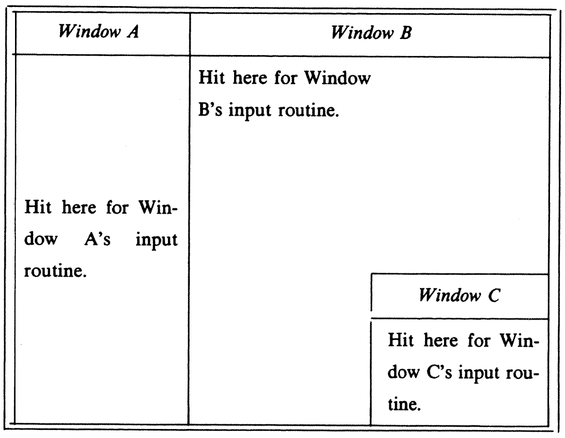
The model of input underlying **GiGo** is a complete break with the device model. The various aspects of this model were rejected as follows:

1. *Device classes* had to go. They were a way of segregating input into different kinds, and experience with operating systems illustrated the advantages of *unifying* all sources of input under a single concept, such as the file[8]. Thus, all physical devices were mapped into a single class, returning a device coordinate and an integer code. A button box would return a valid code and a prestored coordinate. A digitiser might return a valid coordinate and a prestored code.
2. *Input modes* had to go, because they also forced the application to choose between the different ways input might arrive.
3. *Input devices* also had to go. They were either intimately associated with a view surface, as with the Tektronix cursor, or could be described as an input-only screen, as with the digitiser (or the keyboard).

Instead, inputs are treated like interrupts, with an input-handling routine attached to each Window:

```
typedef struct WINDO {
        struct WINDO *w_next;
        Dcoord w_lo, w_hi;
        Ncoord w_ll, w_ur;
        Screen *w_scrn;
        Prim   *w_prim;
        int    (*w_outp)();
        int    (*w_inp)();    /* Input */
        } Window;
```

When the system receives an input, it scans the active Windows on the corresponding view surface in inverse priority order until the position lies within one, transforms the position to the corresponding application space, and invokes the input routine with the position, the code, and the Window as arguments (Fig. 8).



Window A
Window B
Hit here for Window B's input routine.
Hit here for Window A's input routine.
Window C
Hit here for Window C's input routine.

Fig. 8 — GiGo Input

There is (conceptually) no queue; if an application expects that responding to a particular input will take a long time, it is expected to create another process to do so.

Inputs may be synthesised by one input routine calling another directly; they are just routines with known parameter lists. In this way, the push-back facility of CANVAS can be imitated.

4.5. Experience

The first version of **GiGo** took about four weeks to write. It has been available for use in the Architecture department in Edinburgh for more than a year, and three substantial applications have used it. They are:

- A graphics front-end for the logic programming language Prolog[3], providing the user with a window for text interaction, and others for graphics.
- A re-write of a system for maintaining a database describing the accommodation, curriculum, and staff and student numbers of schools in Scotland.
- A ground-modelling system.

The experience of application programmers has been generally favourable. Their comments have led to changes in various areas, particularly in the strategy for deferring updates, and the handling of text I/O.

Using **GiGo** has encouraged them to provide user interface facilities that were previously too much trouble, for example scrolling text windows, and menu items with submenus of common choices. It has proved to be easy for programmers to borrow code implementing particular types of interaction from each other.

Two major differences between **GiGo** and more conventional systems have caused problems of adjustment. They both have positive and negative aspects. The first is that, in **GiGo**, interactions are conceptually all on the top level. That is beneficial, in that applications code written to deal with one interaction sequence need take no special account of the possibility that the operator may break off, perform all or part of another sequence, and then return. On the other hand, the application programmer has to make each input handling routine implementing an interaction sequence into a state machine, since the same code is invoked for every hit in a window.*

* Actually, this need not be so. There is nothing to stop an input routine overwriting its own, or another Window's, *w_inp* with a pointer to another routine. This is in effect maintains the state of the state machine in the Window itself.

The other, related, problem is that the state of any interaction sequence must be maintained in static or global variables, rather than in the local variables of a function invocation (because each hit causes a new function invocation). A "spare" field in the Window structure is often used to point to a structure containing the state of an interaction.

There are some capabilities of **GiGo** which have not yet been heavily used. Although the "reference" primitive allows for multi-level (even recursive) segmentation, in practice only a single level is used. Windows normally clip, and this gives an opportunity no-one has yet used to have symbols that are views of parts of a larger picture.

5. Managing Resources

The hierarchical division of programs in conventional languages into *callers* and *callees* enforces a hierarchical structure on the management of resources. A package must:

a) provide descriptors for graphical resources, and forbid access to such resources without a suitable descriptor being quoted as authorising the access.

b) permit subroutines or subprograms holding descriptors for resources to obtain descriptors for all or part of those resources to pass on to the subroutines or subprograms they themselves invoke.

It should be possible at each level to invoke the same code to allocate the resources available at that level. This code must process requests from the same level for specific resources, e.g:

ActivateWindow(parent, child : Window ; bl, tr : Coord)

asking for the *child* to be mapped to the space between *bl* and *tr* in the parent's coordinate space, requests from below for amounts of resource, e.g:

ReSizeWindow(child : Window ; size : Coord)

asking for the *child* to be made *size.x* by *size.y* big, and warnings from above that its resource allocation has changed. It may itself request more resources from above, but must be prepared to be refused.

5.1. Transformations

The mainstream has no means of preventing a routine accessing any part of the view surface, by altering the current (global) normalisation or workstation transformations.

The mainstream deals with three types of coordinate system, WC, NDC, and DC. Both CANVAS and **GiGo** deal with two, WC and DC. Although CANVAS permits applications to express viewport positions in DC, it supports hierarchical dissection of the screen by clipping away parts of viewports outside their parent.

The logic of resource management suggests that there should be only a single type of coordinate. Each level of the hierarchy is given a descriptor for some part of their parent's coordinate space. This acts as their DC. They may impose their own coordinate system on this, to act as their WC, but by default will use their parent's space. They may pass access to part of their space on to their children. At the root of this hierarchy is a system-provided descriptor for the whole view surface.

5.2. Segmentation

The mainstream has no means of preventing a callee affecting the segmentation environment of its caller, for example by closing the open segment.

The mainstream treats transformations and segments as orthogonal. CANVAS treats "segments" as the units to be transformed. **GiGo** treats transformations as attributes of "segments". The difference between the latter is that CANVAS permits more than one mapping per "segment".

The resource management approach suggests that the natural grouping for manipulation is among primitives sent to the same descriptor. This is reinforced when segments are considered as picture stores. Many devices can erase part of the view surface, and need only to regenerate those "segments" erased.

The mainstream defines its picture store as data, CANVAS defines it as procedures, and **GiGo** gives the application the choice. The choice has proved useful.

5.3. Attributes

The mainstream has no way of preventing a callee from disturbing the attribute context of its caller, for example by changing the global or workstation attributes. Both CANVAS and **GiGo** provide mechanisms whereby a caller can create an attribute context for a callee to operate in.

Note that the resource management approach suggests that *all* graphical resources must be managed. The colour map is such a resource, a window needs to be allocated not merely a patch of device coordinates, but also a part of the colour map.

5.4. Input

The mainstream has no way of preventing a callee intercepting input destined for the caller. All parts of the application can REQUEST or SAMPLE all devices, or remove events from the single queue. By contrast, both CANVAS and **GiGo** ensure that the only input accessible to a routine is that sent to it by the operator pointing at its window.

The resource management approach suggests that devices are not useful concepts; they are unitary and cannot be subdivided to be passed on to children. Input must be directed by the system to appropriate parts of the hierarchy, where the appropriate part is the part the operator is pointing at.

The restriction that input is always processed by the window it was pointing to is too rigid. For example, pointing at a window and saying "grow bigger" is not a command the window can process itself. It requires access to resources (more view surface) belonging to routines further up the tree, and must be processed by them. The window pointed at must be able to "pass-the-buck", handing the input up the tree to more powerful routines. The more powerful routines may also want to preempt their children's handling of input. An "I'll-handle-it" facility is required whereby parents can temporarily remove their children from consideration by the input process.

6. Design Principles

These graphical resource management facilities may be provided by a package in the conventional way, but are more suitable for implementation as a server process, managing the physical view surface for several client processes. The requirements for such a server may be summarised as follows:

1. Hierarchical dissection of graphical resources; each descriptor confers the right to manage that resource in the same way as it is being managed for you.
2. Only a single type of coordinate system; children are positioned in their parent's coordinate space.
3. Only window attributes — each descriptor for some graphical resource carries its own attribute context. Note that this implies that the resource includes a range of entries in the colour map as well as a patch of device space.
4. Window-based input with only a single class, and no modes. Two facilities needed are:
    a) "Pass-the-buck" to let the parent handle something the child cannot.
    b) "I'll-handle-this" to let the parent preempt the child's handling of something.
5. Input events queued by the server for each window, and sent to the clients when they ask. **GiGo**-style input routines are really only suited to single-process systems.
6. Pictures in windows defined either by display file or by procedure, at the application's choice.

REFERENCES

[1] K. C. R. C. Arnold, "Screen Updating and Cursor Movement Optimization: A Library Package," Computer Systems Research Group, Dept. EECS, University of California, Berkeley, California.

[2] J. E. Ball, "Canvas: the Spice Graphics Package," S108, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pennsylvania (October 1981).

[3] W. Clocksin and C. Mellish, *Programming in Prolog,* Springer Verlag, Berlin (1981).

[4] R. A. Guedj and others (eds.), *IFIP Workshop on Methodology of Interaction,* (publishers North-Holland), Seillac, France (May 1979).

[5] ISO, "Graphical Kernel System (GKS) — Functional Description," ISO DP 7942 (January 1982).

[6] D. H. H. Ingalls, "The Smalltalk Graphics Kernel," *BYTE*, pp.168-194 (August 1981).

[7] J. C. Michener and J. D. Foley, "Some Major Issues in the Design of the Core Graphics System," *Computing Surveys* **10**(4), pp.445-463 (December 1978).

[8] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Comm. Assoc. Comput. Mach.* **17**(7), pp.365-375 (July 1974).

[9] D. S. H. Rosenthal, "'Methodology in Computer Graphics' Re-examined," *Computer Graphics* **15**(2), pp.152-162 (July 1981).

[10] D. S. H. Rosenthal, J. C. Michener, G. Pfaff, R. Kessener, and M. Sabin, "The Detailed Semantics of Graphics Input Devices," To be presented at, SIGGRAPH '82, Boston, Mass (July 1982).

[11] SIGGRAPH-ACM (GSPC), "Status Report of the Graphics Standards Planning Committee," *Computer Graphics* **13**(3) (August 1979).

[12] R. F. Sproull, "Raster Graphics for Interactive Programming Environments," CSL-79-6, XEROX PARC, Palo Alto, California (June 1979). (Abridged as *Computer Graphics* **13**(2) August 1979, pp. 83-93)

[13] S. K. Warren and D. Able, "Rosetta Smalltalk: A Conversational Extensible Microcomputer Language," *SIGSMALL Newsletter* **5**(2), pp.36-45 (April 1979).

MC NR

35227