A device-independent graphics package for CAD applications

by R. B. Capelli
G. C. Sax

GSSP (Graphics Support Subroutine Package) is a device-independent two-dimensional graphics package developed by Engineering Design Systems (EDS) to support several major electronic and mechanical computer-aided design applications within IBM. Graphics systems supported range from interactive, high-function, distributed-graphics workstations to passive graphic-output devices. GSSP provides many of the functions usually found in other device-independent graphics packages, with additional support for hierarchically structured display files and distributed graphics systems. The major functions provided by GSSP are described, and an overview of the implementation is presented to show how issues such as interactive performance and human factors are addressed.

Introduction

The Graphics Support Subroutine Package (GSSP) is a device-independent two-dimensional graphics package developed by Engineering Design Systems (EDS) for computer-aided design (CAD) applications within IBM. The applications and GSSP execute on IBM System/370, 4300, 303X, and 308X computers. Applications that use GSSP include integrated circuit layout [1], printed circuit board wiring [2], schematic entry, and mechanical solid modeling.

The environment that has evolved for most CAD applications within IBM is based on large time-shared host computer facilities. Very large designs must be handled, and centralized data must often be shared by several designers or application programs.

It is very important to maximize designer productivity. Fast interactive response time is required, with an objective of 0.5 seconds or less for most responses. Some applications are tailored to particular graphics hardware to optimize human factors. High-performance hardware is justified despite higher costs if it allows a large enough productivity gain for users of an application.

It is also important to be able to exploit new graphics hardware as it becomes available, and to be able to support lower-cost hardware for some uses. It seems unreasonable that major new development should be required *for each application* to support different graphics hardware devices. A device-independent graphics package allows development costs to be minimized since new development to support a device can benefit many applications. The use of a device-independent graphics package within an organization also reduces the need for programmer retraining and encourages the sharing of graphics software among application development groups. These and other advantages of a device-independent graphics package are well known, encouraging proposals and activities leading towards industry and international graphics system standards suitable for a very broad constituency [3, 4].

Certain characteristics of GSSP are important in optimizing interactive performance for the time-shared host environment. These same characteristics also allow potential conflicts between device independence and interactive performance to be avoided.

GSSP uses the concept of a segmented, hierarchically structured display file. Since many CAD applications use a hierarchical database, this allows straightforward image generation with nearly one-to-one correspondence between the structure of the application data and the display file. Similarly, it allows direct correlation of graphic input, in terms of the display file, to the application data. The segmented display file also allows incremental updating of the image, required to provide fast response.

GSSP uses hardware features, whenever possible, to provide graphics package functions. However, graphics package functions are defined by the logical function they perform, rather than in terms of hardware features. This allows a function to be emulated in software, if required, when the necessary hardware features are not available.

Most of the graphics devices used for EDS applications have local intelligence. In order to provide dynamic visual feedback for user input in the time-shared host environment, GSSP exploits this local intelligence with distributed system functions. Dynamic visual feedback techniques, such as rubberbanding, dragging, entity verification, and digital coordinate readouts, improve human factors and increase user productivity by allowing trial-and-error graphic input strategies to be avoided.

This paper presents the history of GSSP development, a summary of the GSSP architecture and the functions it provides, an overview of the implementation of the device driver interface, and some issues of device independence addressed by GSSP.

Development history

GSSP has had two major releases. The first, in use between 1974 and 1982, was implemented to support the IBM 1130-1653 [5] graphic subsystem developed internally for EDS. The IBM 1130 was a 16-bit computer with 64K bytes of storage used as a system controller in this configuration. It provided the local intelligence required to offload interactive functions from the time-shared host, allowing several important dynamic visual feedback techniques to be provided. The capabilities of this configuration have had a lasting influence on EDS applications and the development of GSSP.

Because there was some uncertainty about the hardware it was to support when the functional specifications for GSSP were initially being developed, a device-independent application program interface was a major objective from the start. A prototype version of GSSP was actually implemented for the IBM 2250 Model 3 [6] to allow application program development before the IBM 1130-1653 hardware became available. Painless transition to the newer hardware attests to some early success in meeting that objective.

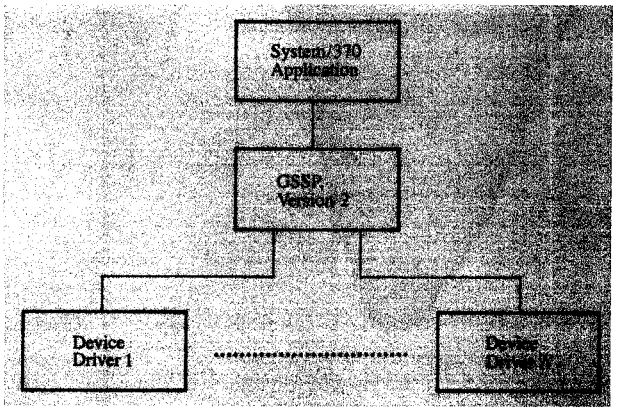

Figure 1

Model of GSSP device-independent graphics system.

The second release of GSSP in early 1982 was heavily influenced by the ACM/SIGGRAPH Graphics Standards Planning Committee (GSPC) proposal for a device-independent Core Graphics System [3]. It was desired to allow applications using GSSP to take advantage of new hardware as it became available, and to take advantage of potential hardware functions that the first release of GSSP did not consider. There were strong similarities between some features of the GSPC Core proposal and the first release of GSSP, so the GSPC Core proposal was a natural model to extend GSSP for functions it did not previously provide.

While the first release of GSSP did provide a device-independent application program interface, its implementation was entirely specific to the IBM 1130-1653 graphic subsystem. The second release of GSSP modularizes the implementation into a device-independent component common to all devices, and into specific device driver modules for each supported device. **Figure 1** illustrates the relationship between the device-independent component and the device drivers.

Device drivers have been developed for several IBM and vendor graphic systems representing several different display technologies and different levels of hardware function. A device driver for the IBM 1130-1653 graphic subsystem, a directed-beam refresh display, provides compatibility to the first release of GSSP. Device drivers have been developed for the IBM 3250 [7] directed-beam refresh display, the IBM 3277 Graphic Attachment [8] using a direct-view storage tube monitor, the IBM 5080 [9] raster display workstation, and microprogrammed vendor directed-beam refresh display

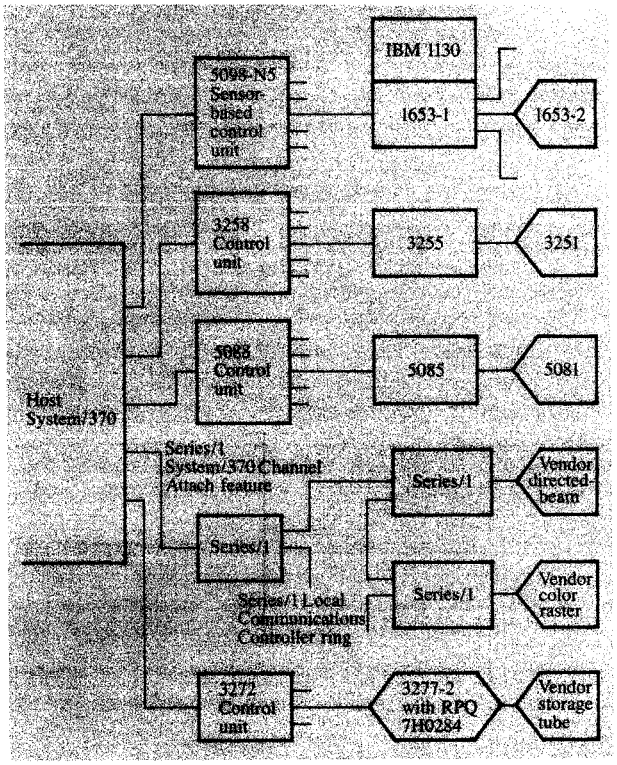

Figure 2

Hardware configurations of interactive graphics systems supported by GSSP.

and raster display systems attached to the IBM Series/1 [10] minicomputer using internally developed interfaces [11]. **Figure 2** summarizes the hardware configurations of these interactive systems. GSSP device drivers also have been or are being developed for several pen, printer, and electrostatic plotters.

GSSP architecture

Graphics applications invoke all GSSP functions using a subroutine call mechanism. GSSP subroutines use standard OS/VS linkage conventions allowing graphics applications to be written using a number of high-level languages.

The application programmer's model of GSSP is built on the following important concepts:

- The *virtual device* represents the ideal device supported by GSSP. The virtual device is mapped onto a physical device to optimally exploit hardware capabilities.
- *View surfaces* represent devices or workstations. There is a one-to-one correspondence between a view surface and a GSSP device driver. More than one view surface may be used concurrently by an application, although graphic input can be obtained from only one primary view surface.
- A *segmented, structured display file* is maintained by each GSSP device driver. The display file maintained by a device driver may be the actual graphic-order list executed by hardware, or it may be a *pseudo-display file* that is interpreted by the device driver software.
- *Logical input devices* are device-independent models of idealized physical graphic-input devices. Graphic applications request input in terms of logical input devices, and GSSP device drivers use available physical devices to obtain the requested input.
- *Distributed system functions* allow capabilities of intelligent graphics workstations to be exploited. The graphic application can request that dynamic visual feedback of certain user input be provided. This immediate feedback is possible only if functions to provide it are implemented at the workstation, rather than on a time-shared mainframe host.

The subroutine calls in GSSP may be divided into five major groups: control functions, graphic-output functions, structured display file functions, graphic-input functions, and distributed system functions.

- *Control*

Control functions initialize and terminate GSSP, define characteristics of the virtual device, initialize, control, and terminate view surfaces, and provide error handling.

Initialization and termination

The subroutines that make up the device-independent component of GSSP are link-edited into a single reentrant, refreshable load module, and accessed using a transfer vector. This allows a single copy of the GSSP load module to be shared in virtual storage by all users on the host system, tending to reduce operating system paging. The application must initialize GSSP before invoking any other GSSP subroutine. During initialization, the transfer vector is initialized, control blocks and tables are allocated and initialized to set various defaults, and the primary view surface is initialized. When an application invokes the GSSP termination routine, all system resources allocated for GSSP are released.

Virtual device characteristics

The default virtual device has a display area with a square aspect ratio and is described by a default Cartesian coordinate system. The application can change this definition of the virtual device by specifying a different aspect ratio and any convenient Cartesian coordinate system. Each device driver maps the virtual device onto a physical device to maintain the specified aspect ratio and to use the largest possible display area.

View surfaces

When a view surface is initialized, the corresponding device driver is loaded into storage and invoked so that it can be initialized. The device driver allocates and initializes its control blocks, tables, and display file. All resources allocated for a view surface are released when a view surface is terminated.

The application may direct graphic output to specific view surfaces by "selecting" and "de-selecting" view surfaces as it segments display files into logical buffers (described in more detail later).

Error handling

Errors can occur because of incorrect programming, such as passing invalid parameters to GSSP subroutines or invoking GSSP subroutines in an invalid sequence. Environmental errors, such as display file buffer overflow or I/O errors, can also occur.

Flexibility is provided by a number of error-handling mechanisms. First, each GSSP function sets a return code indicating any error that was encountered. GSSP also allows the application to obtain a report of the most recent error, if any, that has occurred. Finally, an application exit routine can also be specified to be invoked by GSSP when an error is encountered. The application exit routine can analyze the error and, if desired, attempt to recover.

• *Graphic output*

GSSP provides functions to create two-dimensional graphic-output primitives, to set static attributes affecting graphic output, to specify a window and viewport for a viewing transformation, and to specify image and modeling transformations.

Graphic-output primitives go through a "pipeline" of coordinate processing steps before being displayed. **Figure 3** summarizes the viewing pipeline supported by GSSP.

Output primitives

Lines, polylines, points, characters, circles, arcs, and areas are the basic forms of graphic output in GSSP. Areas are defined by one or more closed shapes composed of line, polyline, circle, or arc primitives. Filled areas can contain holes and islands.

Output primitives are defined with respect to a *current position.* The current position establishes the starting coordinate position at which graphic output is generated; it is subsequently updated to reflect the final coordinate position defined by the graphic output. Coordinate data associated with output primitives may be specified as absolute positions or as relative vector components that modify the current position.

Static attributes

Static attributes are set modally and affect the appearance of subsequent graphic output. GSSP supports linestyle, linewidth, intensity, color, fill pattern, character font, character size, character spacing, and text precision static attributes.

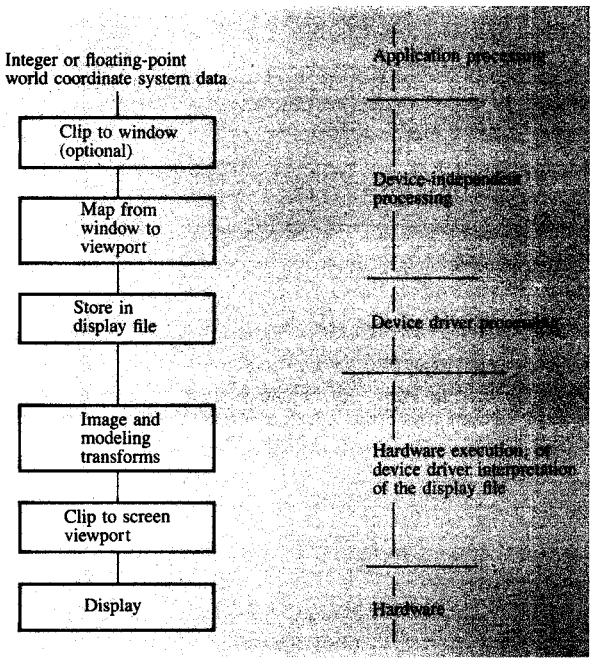

Figure 3

Viewing pipeline supported by GSSP.

Static attributes are meant to take advantage of specific hardware features. If this is not possible for a given view surface, the device driver may emulate the graphic attribute with software, may substitute another static attribute (e.g., intensity for linewidth), or may completely ignore the static attribute (e.g., color on a monochrome view surface). Functions are provided to determine the static attributes supported by hardware so that applications may select static attributes to optimize human factors.

Color is supported using the concept of a color look-up table. Colors in the look-up table may be defined using three common color models [12]: the RGB (red, green, blue) color cube, the HSV (hue, saturation, value) single cone, and the HLS (hue, lightness, saturation) double cone. The application can select a color for subsequent graphic output by specifying the look-up table index static attribute. A convention has been adopted for raster display system device drivers that pixel data are ORed into the frame buffer to generate an image, rather than having new pixel data simply replace data previously stored. (This may be considered a default in the future, should GSSP be enhanced to allow explicit specification of other modes of operation in updating

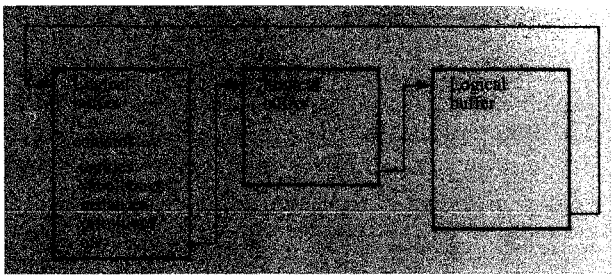

Figure 4

Display file segmented into a linked list of logical buffers.

a raster frame buffer.) For multi-layer images typical of electronic physical-layout design applications, a powerful technique is to assign bit-significant color indexes for data representing a layer. This allows a variety of useful interactive effects, such as transparency and dynamic layer priorities, to be obtained for a color raster display using the look-up table [13]. The color of all graphic output generated with a given look-up table index may be changed by simply defining a new color in the corresponding color look-up table entry.

Applications may specify character font, size, spacing, and text precision for character primitives. Text precision allows application programmers to specify the tradeoff between efficiency and exactness in generating characters. *String precision* allows a hardware character generator to be used on a string-by-string basis for maximum efficiency. Character font, size, spacing, and orientation are approximated as closely as possible. *Character precision* allows a hardware character generator to be used on a character-by-character basis. Characters are spaced exactly as specified. Character font, size, and orientation are approximated as closely as possible. *Stroke precision* requires that characters be generated with the specified font, size, spacing, and orientation, even if this means that each character must be stroked out with line segments.

The viewing transformation

An application can modally specify a window in an arbitrary Cartesian world coordinate system and a viewport in the virtual device coordinate system. Subsequent graphic output defined in the world coordinate system will be optionally clipped to the window and mapped to the viewport on the view surface. The application can specify the data format for the world coordinate system so that coordinate data can be defined as halfword or fullword integers or floating point numbers.

Image and modeling transformations

GSSP functions allow two-dimensional image and modeling transformations to be specified as matrices, to be concatenated into composite transformations, and to be saved and restored using a stack. Using these functions, an application can straightforwardly generate an image from a transformed, hierarchical data structure. The display file can also be updated incrementally to change a previously defined transform matrix, thus causing all graphic output affected by the transform to be changed. Ideally, these functions take advantage of hardware matrix transformation capabilities. GSSP also allows applications to take advantage of hardware capabilities to clip to arbitrary screen viewports.

• *Structured display files*

CAD applications require the ability to manipulate groups of objects as a unit (e.g., a schematic design or an options menu), to manipulate individual objects within a group (e.g., schematic symbols or menu items), and to select a specific object or group of objects as input to an application program. Required, too, is an easy and efficient means to define and replicate common objects (e.g., transistor symbols).

GSSP supports a segmented and hierarchically structured display file that satisfies these requirements. Logical buffers, entities, subpictures, and instances are constructs that segment and structure the display file. These constructs allow the display file to mimic the structure of a nested application database so that an application may efficiently and straightforwardly create an image from the database and process graphic input. When names are assigned to these constructs, they may later be referenced to incrementally update the display file, allowing applications to easily modify, replicate, and delete objects and groups of objects.

Logical buffers

Logical buffers are the highest level of segmentation in the display file. All graphic data must be generated in a logical buffer. Within a logical buffer, graphic data may be generated with or without further structuring. **Figure 4** shows how the display file is conceptually segmented into a linked list of logical buffers.

Logical buffers are generally used to group objects with similar global characteristics. For example, one logical buffer may contain a schematic design, while another logical buffer contains a menu of options for modifying the schematic design. Applications may use logical buffers to modularize the display file to correspond to the modularization of the application program.

When a logical buffer is created, it is defined only for view surfaces currently "selected." Consequently, only these view surfaces reflect the graphic data in the logical buffer. Each logical buffer may be referenced independently after it has been created so that it may be made visible, made invisible,

saved, restored, or deleted without affecting other logical buffers.

Global aspects of graphic data generation, such as a current position, a complete set of static attributes (including defaults), and a viewing transformation (window, viewport, coordinate data format, and clipping status switch), are associated separately with each logical buffer. Any one logical buffer can be designated the "active" buffer. Graphic data are always added to the current active buffer, using the status and attributes maintained for that logical buffer.

Entities

Entities provide the next level of hierarchical structuring within a logical buffer, allowing applications to manipulate local aspects of graphic output such as visibility and positioning. Graphic data generated within an entity can be referenced and modified as a unit, without affecting other graphic data in the logical buffer.

An entity is defined by specifying the beginning and end of a sequence of graphic data. An application can later delete, copy, rename, or reposition the entity, or modify dynamic attributes of the entity.

Only graphic output generated as part of an entity can be detected by a pick device. The name of the selected entity is returned for a pick event. Entities can also be structured to include an "action list" to provide handling of a pick event without direct application intervention (described in more detail later).

The following dynamic attributes of an entity can be modified:

- Visibility—entities may be made visible or invisible.
- Detectability—entities may be made detectable or nondetectable. An entity that is visible and detectable can be selected by a pick device.
- Verifiability—entities may be made verifiable or nonverifiable. An entity that is visible, detectable, and verifiable will provide visual feedback (changed color or intensity level, blinking, etc.) while the pick device is pointing at that entity. When the pick device is not pointing at the entity, the entity returns to its normal state. This feedback is provided without application intervention. Entity verification is practical only if supported by hardware.
- Highlighting—entities may be highlighted or nonhighlighted. Highlighting is some visual effect (changed color or intensity level, blinking, etc.) that causes the graphic output in the entity to be easily distinguished from graphic output that is not highlighted. Unlike verification, highlighting does not depend on the use of the pick device.

Subpictures and subpicture instances

Subpictures can be used to define objects hierarchically in the display file. A subpicture instance invokes a specified subpicture at a specified position. Instances allow copies of an object defined by a subpicture to be easily replicated. Individual subpicture instances may be referenced to change the subpicture invoked by the instance, to alter the instance position, or to make the instance visible or invisible. Ideally, subpictures and instances are implemented by GSSP device drivers using hardware graphic-order subroutine functions.

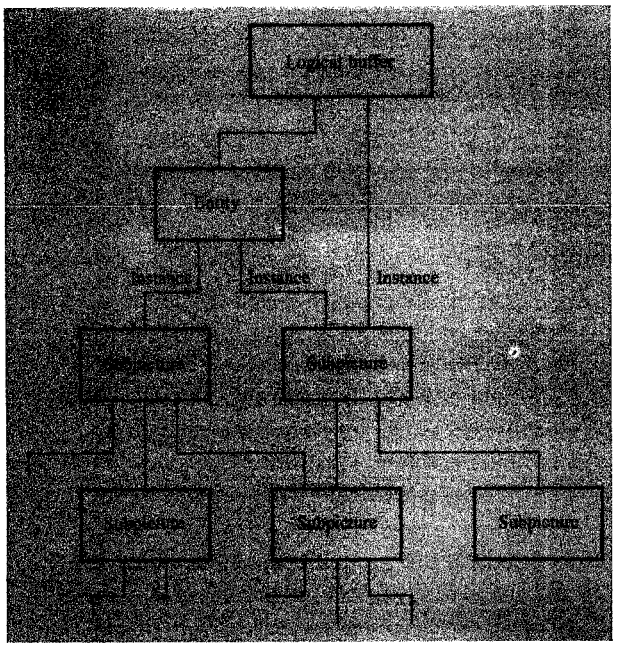

Figure 5

Subpictures and instances allow a hierarchical display file to be built.

The sequential generation of graphic data in the active logical buffer is suspended when a subpicture is started, and is continued when the subpicture is ended. A subpicture is defined in its own local Cartesian coordinate system. Graphic data generated in a subpicture have no relationship to graphic data generated outside of the subpicture until it is invoked by a subpicture instance.

Graphic output in a subpicture inherits the static attributes in effect for each instance of the subpicture, unless static attributes are explicitly set within the subpicture. Any changes to the current position and static attributes in the subpicture, however, do not affect graphic output generated following a subpicture instance.

Subpictures may instance other subpictures to build nested tree structures, as illustrated by **Figure 5.** Such hierarchical structures are often instanced within entities. When a pick event occurs, GSSP can return a trace of the tree structure in a selected entity to assist an application in correlating the selected primitive to the application database.

• *Graphic input*

GSSP provides functions to control logical graphic-input devices and obtain graphic input. These functions support operator interaction with the primary view surface only. No graphic input is available to an application if the primary view surface does not support operator interaction, as in the case of a plotter.

Logical input devices

Five logical classes of graphic-input devices are supported by GSSP: picks, buttons, locators, keyboards, and valuators. Logical input devices can be modally enabled and disabled. When a logical input device is enabled, the device driver for the primary view surface uses any combination of physical input devices and software that will emulate the logical input device and obtain corresponding input.

Picks A logical pick device returns the name of a selected entity. The model for the ideal pick device supported by GSSP is a lightpen with a tipswitch. A detectable entity may be selected while the pick device points at it by closing the (possibly emulated) tipswitch to generate a pick event.

When the hardware can support it, a pick device can cause a detectable, verifiable entity to be dynamically verified. This dynamic visual feedback is accomplished by changing the intensity or color or otherwise distinguishing the entity while the pick device is pointed at the entity with the tipswitch open. This feedback not only indicates whether the pick device is pointing at the intended entity, but also allows the user of a graphics application to determine which entities can be selected and how data are meaningfully grouped in entities. Entity verification greatly enhances human factors for EDS graphics applications, where entities naturally describe circuit elements, wiring nets, etc.

Locators A logical locator device returns a coordinate position. Ideally, a visible echo known as the "tracking symbol" is displayed at the current locator position, which may be initialized by the application. The locator position is continuously updated to track any movement of the locator device. The final locator position may be determined following an event. Among physical devices used by GSSP device drivers to provide a logical locator device are lightpens, joysticks, and tablets.

GSSP maintains a locator window in an application world coordinate system and a locator viewport in the virtual device coordinate system, allowing an application to obtain or set the locator position in the world coordinate system. Using distributed system support functions, which are described later, GSSP provides a number of options for additional support of the locator. The locator position can be constrained to the locator viewport, and to horizontal, vertical, or diagonal directions. Various crosshairs can be attached to the locator position. Bump buttons can be defined to allow the locator position to be moved in what can even be subpixel increments. Dynamic digital readout of locator position information in the world coordinate system can be requested.

Buttons The model for logical buttons supported by GSSP is a lighted program function keyboard. Enabled logical buttons cause an event when they are used. The button number reported for the event can be interpreted by the application as a choice among application-dependent commands. GSSP allows lights or other prompting mechanisms to be set independently of the enabled buttons. Since EDS graphics applications often group buttons in iconic patterns, GSSP provides functions to map logical button numbers to physical button configurations.

Keyboards Alphanumeric text is entered using a keyboard. As described later, text is echoed as it is typed, when possible, directly in editable message fields defined by the application in the display file. The application specifies the initial keyboard cursor position in a message field when the keyboard is enabled. An event is reported when the "enter" key is used.

Valuators Valuators return scalar values in an application-specified range. Dials are the model for logical valuator devices supported by GSSP.

Obtaining graphic input

Graphic input is made available to an application following an event. The application may use GSSP to poll or wait for an event. Events may be generated using an enabled pick, button, or keyboard device. For each event, the logical device type and specific device causing the event are reported.

Information that is available after an event includes the locator position and valuator scalar values. Keyboard input is determined by inquiring the contents of editable message fields. When a pick event occurs, the identifier of the logical buffer containing the selected entity and the selected entity name can be obtained, as well as the coordinate position and the trace of the hierarchical display file structure for the graphic primitive pointed to by the pick device.

• *Distributed system support*

GSSP provides distributed system functions that allow applications executing on the time-shared host system to exploit capabilities of intelligent workstations. Distributing important fixed functions to a processor local to a graphics terminal offloads the time-shared host, and allows dynamic visual feedback for user interaction to be provided that otherwise would not be possible with a time-shared host mainframe computer [14]. Dynamic visual feedback can improve user productivity by allowing trial-and-error graphic

input strategies to be avoided. For example, a rubberband line shows the designer the actual relationships between a new line that is to be created and other design data. Without the dynamic visual feedback of a rubberband line, the designer may need to input and delete several trial lines to iterate to a satisfactory result.

The fixed-function distributed support provided by GSSP is defined in a device-independent way, by associating fields in the display file with logical input devices. When there is no processor local to a graphics terminal that can be programmed to provide the defined capabilities, the fixed-function support can be emulated by the GSSP device driver on the host computer.

The capabilities provided by fixed-function distributed support are editable message fields, locator dragging and rubberbanding, various options for enhanced locator control and feedback, action lists, gates, and data registers.

GSSP provides some basic support for distributed applications. This support allows a part of an application to be distributed to an intelligent workstation, and allows the portion of an application executing on the host system to control and communicate with the portion executing on the distributed processor. This capability allows distributed application functions beyond the fixed-function distributed support provided by GSSP. However, device independence may not be maintained unless the *application* provides equivalent functions for the various supported graphics systems.

Message fields

A message field is a text string in the display file that is assigned an identifier so that the field can be referenced. A message field can be made editable to allow input from a keyboard, and can be updated, cleared, or deleted by the application. A message field text string may be positioned anyplace on the screen, and is always generated with the string precision attribute.

GSSP supports a keyboard cursor that may be initially inserted into any editable message field by the application. Keyboard input is echoed in the message field, as it is entered if possible. The cursor can be moved to any editable message field in the display file using a "jump" key.

After an event, the contents of message fields may be obtained to allow the application to determine keyboard input. The current keyboard cursor position can also be obtained, in terms of the identifier of the message field containing the keyboard cursor, the cursor position in the message field, and the identifier of the logical buffer containing the message field.

Locator dragging and rubberbanding

Graphic data can be "attached" to the locator to allow "dragging" and "rubberbanding." GSSP functions allow the locator to drag any arbitrary graphic primitives, to drag entities, to drag subpicture instances, to rubberband lines, and to rubberband rectangles (by moving one corner with the locator while the diagonally opposite corner is fixed). Any combination of these techniques can be used simultaneously.

Enhanced locator control and feedback

The following distributed-system fixed functions allow special capabilities for dynamic visual feedback of the locator tracking symbol position:

- Area constraint—the locator position is constrained to the locator viewport.
- Axis constraint—the locator position can be constrained to a horizontal or vertical axis, or to either 45-degree diagonal.
- Grid constraint—the locator position can be constrained to intersection points of an application-specified grid overlaying the locator window.
- Crosshairs—axial crosshairs that span the locator viewport may be "attached" to the locator. The crosshairs may be flat (vertical and horizontal) or tilted (positive and negative 45-degree diagonals).
- Digital readout—the locator position and the vector components between the locator position and a specified reference position may be dynamically "read out" into specified message fields. These digital readouts are displayed in an application-specified world coordinate system.
- Bump buttons—an application can specify buttons that move the locator position, in compass directions, by a specified world coordinate system increment, without requiring a button event.

Action lists

Entities can be structured to include action lists to allow pick selections to be handled dynamically. When an entity containing an action list is selected with the pick device, the action list is executed. No pick event is reported to the application. An action list allows limited display file programming to provide dynamic visual feedback and to collect data for graphic input. Action-list functions generate graphic orders in the display file that, when executed by hardware or interpreted by device driver software, can dynamically reposition entities and subpicture instances, open and close gates, and store data in data registers.

Gates

The model for a gate is a conditional branch in the display file. A gate is defined by specifying the start and end of the gated sequence of graphic data in a logical buffer. Gated sequences may be overlapped. If a gate is open, the gated sequence is executed when the image is generated. If the gate is closed, the gated sequence is not executed. The application

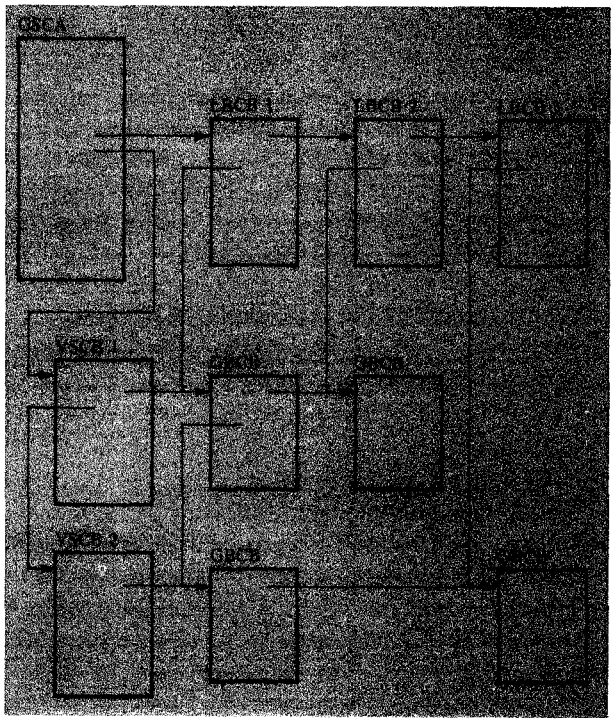

Figure 6

GSSP control block relationships. In this case, an application is using two view surfaces and has created three logical buffers. Logical buffer 1 is associated with both view surfaces, logical buffer 2 is associated only with view surface 1, and logical buffer 3 is associated only with view surface 2.

program specifies whether a gate is initially open or closed, and may later reference a gate to change its state.

Gates can be used to change the visibility and static attributes of graphic output. This can be done dynamically by using action list functions to open and close gates. After an event, an application may determine whether gates have been open or closed.

Data registers

An application may define an array of data registers. Action list functions can dynamically store data constants in specified data registers. An application can obtain the contents of the data registers following an event to determine, for example, which action lists were executed.

Implementation

An overview of the implementation of the GSSP device driver interface can provide some insight into the way various issues of device independence related to interactive performance are resolved.

A key issue in the design of the second release of GSSP was the split of function between the device-independent component and device drivers. It is desirable to make device drivers simple to implement, implying that as much function as possible should be provided by the device-independent component. A conflicting goal, however, is to take advantage of hardware capabilities whenever possible, implying that function should be delegated to device drivers. GSSP resolves these tradeoffs for various cases in different ways, but usually leans towards delegating function to the device driver to allow the best interactive performance to be achieved. While this can make the development of device drivers relatively more difficult, it is expected that subroutine libraries of common services will evolve to assist in developing device drivers for graphics systems having similar characteristics.

Each GSSP device driver maintains its own display file. This may be a pseudo-display file interpreted by the device driver, or the same physical display file used by refresh display hardware. When the device driver is implemented to use a specific hardware display file format, the need to translate between different representations of the same data is eliminated, so that interactive performance is optimized.

A device driver is invoked by the device-independent component of GSSP using a subroutine procedure call whenever a device-dependent action is required. Typically, a device driver is invoked to add to the display file, to modify or inquire into fields already defined in the display file, and to obtain graphic input.

The interface protocol between the device-independent component and device drivers is based on control blocks in which the state of the system is maintained. This control block interface allows the device-independent component and a device driver to share system state information. **Figure 6** shows the relationship between the various major control blocks. The control blocks are defined and organized according to the kind of state information they contain (global to the system, or specific to a logical buffer), and according to the owner of the state information (device-independent component, or a specific device driver).

The graphic system control area (GSCA) is owned by the device-independent component and contains state information global to the entire graphics system, such as the aspect ratio and coordinate system of the virtual device. The GSCA points to a list of view surface control blocks and to a list of logical buffer control blocks.

There is a view surface control block (VSCB) for each initialized device used by a graphics application. Each VSCB is owned by a device driver, and contains state information global to the device driver. The first part of a VSCB has a fixed format that can be accessed by the device-independent component to obtain device characteristics, such as physical device addressability, static attributes supported by hardware, etc., and to obtain event reports for graphic input.

It also points to a list of graphic buffer control blocks. The remainder of the VSCB is called the VSCB extension and contains device driver implementation-dependent information, such as actual display file pointers.

There is a logical buffer control block (LBCB) for each initialized logical buffer. LBCBs are owned by the device-independent component and contain device-independent state information for a logical buffer, such as the current world coordinate position for graphic output, the static attributes in effect, the viewing transformation window and viewport, state information controlling display file segmentation and structure, etc. An LBCB points to a list of graphic buffer control blocks. Each LBCB also points to a graphic field control table (GFCT), which maintains device-independent information about each construct in the logical buffer to which an identifier is assigned.

A graphic buffer control block (GBCB) is owned by a device driver and contains device-dependent information about a logical buffer associated with a specific view surface. The first part of a GBCB has a fixed format that can be accessed by the device-independent component. It contains the current physical device coordinate position and the transformation factors to convert directly from the world coordinate system to physical device coordinates. The remainder of the GBCB, the GBCB extension, contains device driver implementaton-dependent information.

A device driver maintains global pointers to its display file in the VSCB extension and pointers to logical buffers within the display file in the GBCB extension. A device driver usually maintains what we have called a buffer reference pointer table (BRPT) in the GBCB extension to allow quick access to fields defined with an identifier in a logical buffer. The BRPT contains pointers in the display file to entities, message fields, subpictures, instances, gates, and transforms within a logical buffer. BRPT entries correspond to entries in the GFCT maintained by the device-independent component and can be located by using the same table indexes.

There is little extra overhead in maintaining the control blocks to separate device-independent and device-dependent information. Most of the information would be needed in a device-specific implementation anyway. In a number of tests comparing the performance of the device-specific implementation of the first release of GSSP with the second release of GSSP using a device driver for the same IBM 1130-1653 system, we have never observed any execution time difference greater than one percent.

Issues of device independence

We describe GSSP as a device-independent graphics system. Ideally, an application using GSSP should be able to be used on any device which is supported by a GSSP device driver. Is this really the case? What is gained and what is lost by an application using a device-independent graphics subroutine package?

GSSP device drivers for passive-output devices such as plotters do not support graphic-input functions, so interactive applications obviously cannot be used with those devices. The advantages of providing output-only support for plotters with GSSP are that output-only application program modules can be written that are device-independent, and that application programmers need learn how to use only one basic graphics package.

With the obvious exception of output-only systems, we believe that we have been very successful in allowing interactive graphics applications to be run on devices using different display technologies and having different levels of hardware function. Responsibility for device independence, however, must be shared by a graphics application as well as the graphics package. For example, if an application ignores the possibility of buffer overflow errors that can be reported by GSSP, the application may run successfully on storage tube or raster frame buffer display systems, but fail on display systems having relatively limited amounts of display file storage. A more robust application would check for the buffer overflow condition and notify the user and allow a means of displaying a smaller window, for example, if the condition occurred.

Developers of highly interactive graphics applications are often very aware of hardware capabilities and limitations. Such applications tend to be targeted to a specific device, with a good deal of care taken to optimize human factors. The use of a device-independent graphics package need not restrict such an approach. GSSP provides a rich set of functions that directly support hardware features. GSSP also provides inquiry functions that an application can use to determine, and thus tailor its operation for, important device characteristics.

Even if a graphics application is targeted to a particular device for optimized human factors, the use of a device-independent graphics package allows that application to at least run on other supported devices. This benefits the pragmatic end user who would like to use the application but does not have ready access to the first-choice hardware. The human factors of the application do indeed change. The end user, however, can still usually understand the images produced by the application even if capabilities such as intensity, color, linestyle, and entity verification are not provided, and can still interact with the application even if input devices are simulated by the device driver.

Probably the most important benefit of using a device-independent graphics package is the ability of an application to migrate to new hardware that obviously cannot be anticipated by the application developer. Once a device driver is available for a new device, retargeting an application specifically for it usually requires only incremental changes to optimize human factors for device characteristics and to take advantage of new hardware features. Migration is especially simple when the new device

supports the features of the earlier hardware as well as providing new capabilities, which is often the case.

Summary

GSSP provides a rich set of atomic functions that can be combined with a great deal of flexibility to provide a wide range of application techniques. GSSP functions directly support the capabilities of many graphics hardware configurations. However, GSSP functions are defined in a device-independent manner that can be interpreted in a meaningful way by device driver software if necessary.

CAD applications seek to optimize interactive performance and human factors to maximize designer productivity. The segmented, structured display file and distributed system functions supported by GSSP are especially useful for meeting these objectives. GSSP functions allow images to be efficiently generated and incrementally updated from hierarchically structured data typical of many CAD applications. Meaningful dynamic visual feedback can be provided for graphic input, and graphic input can be directly correlated to the application data.

As hardware costs decline, we expect to see more function provided by graphics hardware, and tighter coupling of the graphics hardware to the computer on which the application executes. Such a configuration will provide improved interactivity and more dynamic visual feedback to user input, resulting in higher user productivity. We are confident that the direction we have taken with GSSP will allow us to continue to take advantage of these developments.

Acknowledgments

Major contributors to the design of GSSP were Frank Skinner, Lou Biskup, Al Barone, Phil Carmody, Greg Morrill, and the late Cal Lovejoy. We also wish to acknowledge everyone who has contributed to the implementation and maintenance of GSSP and its device drivers, although the number of these good people has grown too large to list here.

References and notes

1. P. Carmody, A. Barone, J. Morrell, A. Weiner, and J. Hennesy, "An Interactive Graphics System for Custom Design," *Proceedings of the 17th Design Automation Conference*, Minneapolis, MN, June 1980, pp. 430–439.
2. F. D. Skinner, "The Interactive Wiring System," *IEEE Computer Graphics & Applications* **1**, No. 2, 38–51 (April 1981).
3. "Status Report of the Graphic Standards Planning Committee of ACM/SIGGRAPH," *Computer Graphics* **13**, No. 3 (August 1979).
4. "Graphical Kernel System (GKS) Functional Description," Draft International Standard, ISO/DIS 7942 (November 1982).
5. The IBM 1130-1653 graphic subsystem was developed for internal use within IBM and was never marketed. The subsystem was based on the IBM 1130 computer and the IBM 2250 Model 4 display unit, which was marketed. Information about the IBM 2250 Model 4 can be found in *IBM 1130 Computing System Component Description: IBM 2250 Display Unit Model 4*, Order No. GA27-2723, available through IBM branch offices.
6. *IBM System/360 Component Description: IBM 2250 Display Unit Model 3, IBM 2840 Display Control Model 2*, Order No. GA27-2721, available through IBM branch offices.
7. *IBM 3250 Graphics Display System Component Description*, Order No. GA33-3037, available through IBM branch offices.
8. *IBM 3277 Display Station, Graphics Attachment RPQ 7H0284, Custom Feature Description*, Order No. GA33-3039, available through IBM branch offices.
9. *IBM 5080 Graphics System: Principles of Operation*, Order No. GA23-0134, available through IBM branch offices.
10. *Series/1 Digest*, Order No. G360-0061, available through IBM branch offices.
11. These systems were developed for internal use within IBM, and the interfaces for these vendor systems to the IBM Series/1 are not marketed.
12. J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co., Reading, MA, 1982, Chapter 17, pp. 611–620.
13. S. L. Tanimoto, "Color-Mapping Techniques for Computer-Aided Design and Verification of VLSI Systems," *Computers & Graphics* **5**, 103–113 (1980).
14. J. D. Foley, "A Tutorial on Satellite Graphics Systems," *Computer* **9**, No. 8, 14–21 (August 1976).

Received December 7, 1983; revised March 8, 1984

Ronald B. Capelli *IBM Data Systems Division, P.O. Box 390, Poughkeepsie, New York 12602.* Mr. Capelli is an advisory engineer in solid modeling development. He received his B.S. in electrical engineering from the University of Michigan, Ann Arbor, in 1973. He joined IBM in 1973 in East Fishkill, and has worked until 1984 in the Engineering Design Systems area. Mr. Capelli has worked in development of interactive graphics applications, graphics subroutine packages, and the architecture and microprogramming of high-performance graphics systems. He is a member of ACM/SIGGRAPH, the IEEE Computer Society, and Tau Beta Pi.

George C. Sax *IBM General Technology Division, East Fishkill facility, Hopewell Junction, New York 12533.* Mr. Sax is a staff programmer in device support development for Engineering Design Systems. He received his B.S. in computer science from Union College, Schenectady, New York, in 1979. Since joining IBM in 1979 in East Fishkill, he has worked in the EDS area. Mr. Sax has worked in the development of alphanumeric subroutine packages, graphics subroutine packages, and EDS systems support.