1991

D. Soede, F. Arbab, I. Herman, P.J.W. ten Hagen

The GKS input model in Manifold

Computer Science/Department of Interactive Systems Report CS-R9127 May

CWI, nationaal instituut voor onderzoek op het gebied van wiskunde en informatica

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Copyright © Stichting Mathematisch Centrum, Amsterdam

The GKS Input Model in MANIFOLD

Dirk Soede
Farhad Arbab
Ivan Herman
Paul J. W. ten Hagen

CWI
Department of Interactive Systems
P. O. Box 4079, 1009 AB Amsterdam
The Netherlands

ABSTRACT

This paper describes the specification of the GKS input model in MANIFOLD. The aim of the work reported in this paper was two-fold: first, to review the communication patterns implied by the GKS input model, and second, to evaluate the suitability of the MANIFOLD language as a tool for defining complex dynamic interaction patterns that are common in non-trivial user interfaces.

The GKS input model is also adopted by all more recent ISO graphics standard documents. A more formal scrutiny of the inter-communication of the components of this model, excluding the implementation details of their functionality, is instructive in itself. It can reveal directions for improvement of its shortcomings and for generalization of its strengths for the ongoing effort to define the functionality of future graphics packages.

MANIFOLD is a language for describing inter-process communications. Processes in MANIFOLD communicate by means of buffered streams and by reacting to events raised asynchronously by other processes. Our experience shows that MANIFOLD is a promising tool for describing systems of cooperating parallel processes. Our MANIFOLD specification of the GKS input model offers a very flexible way to structure user defined logical input devices. Furthermore, it is simple and modular enough to allow easy extensions to include more functionality by local modifications. As such, it can serve as a basis for possible extensions and enhancements envisioned for future graphics packages.

1987 CR Categories: C.1.2, C.1.3, C.2.m, D.1.3, F.1.2, I.1.3, I.3.6, I.3.4
1885 Mathematical Subject Classification: 68N99, 68Q10, 68U05
Keywords and Phrases: logical input devices, computer graphics standards, parallel programming languages.

1. Background

The first ISO (International Standards Organization) graphics standard, the *Graphical Kernel System* (commonly abbreviated as *GKS*)[13] was published in 1985. This specification was the outcome of a significant amount of work performed by a group of experts of different nationalities. This standard, as well as its followers like CGI[14], GKS-3D[12], PHIGS[15], and PHIGS PLUS[16], have played an important role in computer graphics ever since, both from an industrial as well as an educational point of view (see for example Arnold and Duce[4] or Howard et al[11] for a good overview of all these graphics standards).

However, it is widely recognized that the advances in hardware since 1985 (high performance graphics workstations, low-cost laser printers, the predominance of raster technology both in display and hardcopy devices, etc.) have changed the environment of graphics systems to one which is very different than that envisioned for GKS or even its successors. Meanwhile, many new application areas have also come to

the fore, for example, multi-media systems, embedded systems, cooperative working, and scientific visualization. The advent of low-cost, powerful graphics workstations has also led to the widespread use of visual interfaces to application programs. All these new areas demand graphics support beyond the scope of that provided by the current ISO/IEC standards for 2D and 3D graphics (GKS, GKS-3D and PHIGS). Consequently, several institutions (including ISO) and expert groups have already started to work on the specification of a successor for GKS.

Such a renewal of effort towards the specification of a new standard has several aspects; one of them, which is our concern in this paper, is to take a very critical look at the existing specifications to understand both their strengths and their deficiencies. To this end, among other approaches, the use of more formal specification methods to just simply *describe* the model defined by GKS is important by itself. This is so because the GKS model has been adopted virtually unaltered, by all other ISO graphics standard documents mentioned above, as well. Indeed, the official ISO documents on GKS contain a fairly informal (albeit quite precise) specification of the GKS model. This informality has been the source of a number of misunderstandings and misinterpretations in the past in implementations of the GKS standard. The use of more formal methods may also help in defining a much more advanced model for computer graphics to satisfy contemporary requirements such as user extensibility and the use of advanced parallel hardware. Such formal methods have already been used in an attempt to describe the graphics output pipeline, or at least a significant portion of it (see for example Duce[6]).

2. Introduction

The *input model* of GKS is a relatively independent entity within the description of the standard itself. Although it offers a fairly complex model (which has been reused with only very minor changes in PHIGS and PHIGS PLUS as well), it has been one of the widely criticized chapters of the standard for its relative rigidity, and for the difficulties involved in its adaptation to support new types of input devices.

An interesting aspect of this input model is that it attempts to give a consistent picture of input handling, using the terminology of concurrent processes. This is a natural consequence of the complex nature of input handling in interactive graphics applications, and the fact that it is physically possible for a workstation operator to simultaneously use more than one of its input devices. However, this is the only point in the standard where parallelism is alluded to at all. Unfortunately, the terminology of concurrent processes is used in the standard documents in a fairly imprecise way: the very notion of a process, when it is created and when it dies, etc., are not defined at all; what these and other ''process-related'' notions effectively mean is therefore ''implementation dependent.'' For instance, going through the exercise presented in the sequel, we had different ''correct'' interpretations of these concepts in mind which we had to reconcile among ourselves first. (This is significant, considering that some of the authors of this paper have been involved with the definition of the GKS standard, two independent commercial GKS implementations, and previous papers giving a formal description of the GKS input model using CSP.)

Apart from its input handling aspect, the very notion of parallelism makes the input model of GKS very interesting for future improvement of graphics standards. Indeed, it is widely recognized that concurrency must somehow be included in any future graphics package specification to improve its user-extensibility and its compatibility with more advanced hardware. However, there is no common agreement on which of the different formalisms for parallel processing are the most suitable for the needs of such specifications. To find suitable tools for describing future graphics standards as systems of cooperating parallel processes, it seems inevitable at this point that different approaches must be tried out and tested using realistic examples. The GKS input model seems to be a good first candidate for such studies. Some of our colleagues have already produced a formal description of this model using Hoare's CSP[9] (see for example Duce, van Liere, and ten Hagen 1989[5] or Duce, van Liere, and ten Hagen 1990[7]). Their work provides a good reference point against which the usefulness of other tools and formalisms, such as MANIFOLD, can be measured. Compared to CSP, we believe MANIFOLD allows more flexibility and modularity which seem to be essential for describing the entire functionality (not just the input handling part) of emerging graphics standards.

The rest of this paper is organized as follows. Section 3 is a brief overview of the MANIFOLD language. MANIFOLD is a language for describing complex communications among large numbers of concurrent autonomous agents that comprise a dynamic parallel system. Graphics environments and complex user

interfaces are among the areas that can benefit from MANIFOLD. However, MANIFOLD is useful in a much broader context for programming of parallel systems. The length of Section 3 may seem discouraging to readers with no particular interest in parallel programming. This section can be easily skipped by those readers and used only as a reference in case they wish to understand the details of the MANIFOLD programs presented later in the paper.

Section 4 describes the GKS input model. The notion of processes in MANIFOLD and the ease and the flexibility with which they can be organized into cooperating agents help the construction of an implementation of the GKS input model in MANIFOLD, described in Section 5. This implementation is based on a few reusable components. It is modular and flexible enough to easily allow a range of changes to its functionality and behavior. A few examples of these are discussed in Section 6, together with their implications on the MANIFOLD program of Section 5. Some such changes are small variations of the GKS standard specification. Others go further beyond the intentions of the GKS standard and show some of the extended functionality that is desired to have in future graphics environments. Section 7 is the conclusion of the paper, and a more detailed version of the MANIFOLD programs discussed in the paper appear in the appendices.

3. The MANIFOLD Language

In this section we give a brief and informal overview of the MANIFOLD language. The sole purpose of the MANIFOLD language is to describe and manage complex communications and interconnections among independent, concurrent processes. A detailed description of the syntax and the semantics of the MANIFOLD language and its underlying model is given elsewhere[3]. Other reports contain more examples of the use of the MANIFOLD language[1,2].

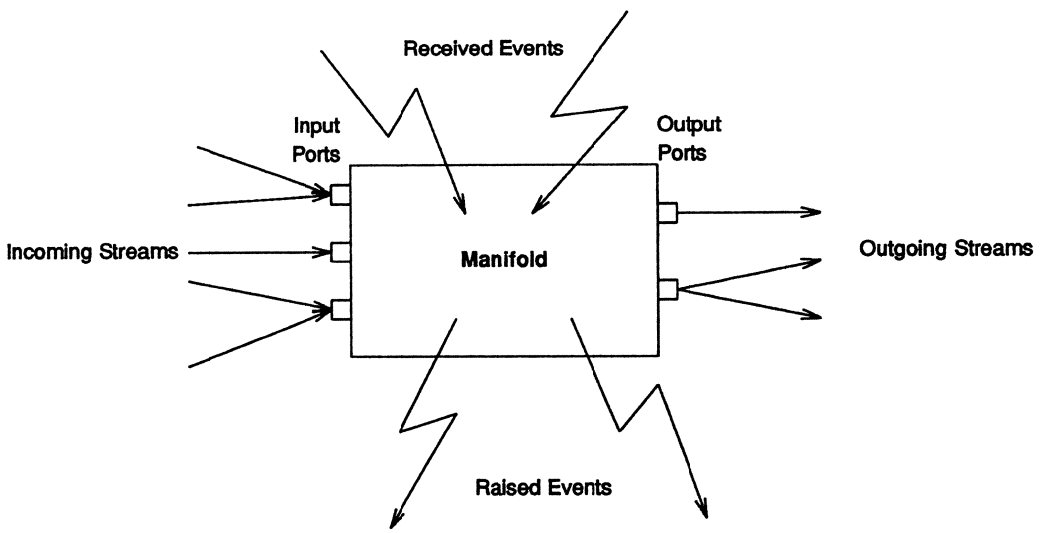

Figure 1 - The model of a process in MANIFOLD

The basic components in the MANIFOLD model of computation are processes, events, ports, and streams. A process is a *black box* with well defined ports of connection through which it exchanges *units* of information with the other processes in its environment. The internal operation of some of these black boxes are indeed written in the MANIFOLD language, which makes it possible to open them up, and describe their internal behavior using the MANIFOLD model. These processes are called manifolds. Other processes may in reality be pieces of hardware, programs written in other programming languages, or human beings. These processes are called *atomic processes* in MANIFOLD. In general, a process in MANIFOLD does not, and need not, know the identity of the processes with which it exchanges information. Figure 1 shows an abstract representation of a MANIFOLD process.

The interconnections between the ports of processes are made with streams. A *stream* represents a flow of a sequence of units between two ports. Streams are constructed and removed dynamically between ports of the processes that are to exchange some information. The constructor of a stream need not be the sender or the receiver of the information to be exchanged: any third party manifold process can define a connection between the ports of a producer process and a consumer process. Furthermore, stream definitions in MANIFOLD are generally additive. Thus a port can simultaneously be connected to many different ports through different streams. The flows of units of information in streams are automatically replicated and merged at outgoing and incoming port junctions, as necessary. The units of information exchanged through ports and streams, are *passive* pieces of information that are synchronously produced and synchronously consumed at the two ends of a stream, with their relative order preserved.

Orthogonal to the stream mechanism, there is an event mechanism for information exchange in MANIFOLD. Contrary to units in streams, events are atomic and *active* pieces of information that are broadcast by their sources in the environment. In principle, *any* process in the environment can pick up such a broadcast event. In practice, usually only a few processes pick up occurrences of each event, because only they are *tuned in* to their sources. Occurrences of the same event from the same source can override each other from the point of view of some observer processes, depending on the difference between the speed of the source and the reaction time of an observer.

Events are generally raised by their sources and dissipate through the environment. They are active pieces of information in the sense that in general, they are observed asynchronously and once picked up, they preemptively cause a change of state in the observer. Events are the primary control mechanism in MANIFOLD. Each state in a manifold defines a pattern of connections among the ports of some processes. The corresponding streams implementing these connections are created as soon as a manifold makes a state transition (caused by an event) to a new state, and are deleted as soon as it makes a transition from this state to another one. In general, the set of sources whose events are honored by an observer manifold, as well as the set of specific events which are honored, are both state dependent.

3.1. Manifold Definition

A manifold definition consists of a header, public declarations, and a body. The header of a manifold definition contains its name and the list of its formal parameters. The public declarations of a manifold are the statements that define its links to its environment. It gives the types of its formal parameters and the names of events and ports through which it communicates with other processes. The body of a manifold may also contain additional declarative statements, defining *private* entities. A manifold body primarily consists of a number of *event handler blocks*, representing its different execution-time states. For an example of a manifold, see Listing 1.

Conceptually, each activated instance of a manifold definition – a manifold for short – is an independent process with its own virtual processor. A manifold processor is capable of performing a limited set of actions. This includes a set of *primitive actions*, plus the primary action of setting up *pipelines*.

Each event handler block describes a set of actions in the form of a *group* construct. The actions specified in a group are executed in some non-deterministic order. Often, these actions lead to setting up *pipelines* between various ports of different processes. A *group* is a comma-separated list of members enclosed in a pair of parentheses. In the degenerate case of a singleton group (which contains only one member) the parentheses may be deleted. Members of a group are either primitive actions, pipelines, or groups. The setting up of pipelines within a group is simultaneous and atomic. No units flow through any of the streams inside a group before all of its pipelines are set up. Once set up, all pipelines in a group operate in parallel with each other.

A *pipeline* is an expression defining a tandem of streams, represented as a sequence of one or more groups, processes, or ports, separated by right arrows. It defines a set of simultaneous connections among the ports of the specified groups and processes. If the initial (final) name in such a sequence is omitted, the initial (final) connection is made to the current input (output) port. Inside a group, the current input and output ports are the input and output ports of the group. Elsewhere, the current input and output ports are **input** and **output**, i.e., the executing manifold's standard input and output ports.

In its degenerate form, a pipeline consists of the name of a single port or process. Defining no useful

connections, this degenerate form is nevertheless sometimes useful in event handler blocks because it has the effect of defining the named port or process as an active source of events.

An event handler block may also describe sequential execution of a series of (sets of) actions, by specifying a list of pipelines and groups, separated by the semicolon (;) operator.† In reaction to a recognized event, a manifold processor finds its appropriate event handler block and executes the list of sequential sets of actions specified therein. Once the manifold processor is through with the sequence in its current block, it terminates.

3.2. Event Handling

Event handling in MANIFOLD refers to a preemptive change of state in a manifold that observes an event of interest. This is done by its manifold processor which locates a proper event handler for the observed event. An event handler is a labeled block of actions in a manifold. The manifold processor makes a transition to an appropriate block (which is determined by its current state, the observed event and its source), and starts executing the actions specified in that block. The block is said to *capture* the observed event (occurrence). The name of the event that causes a transfer to a handling block, and the name of its source, are available in each block through the pseudonyms **event_name** and **event_source**, respectively.

In addition to the event handling blocks explicitly defined in a manifold, a number of default handlers are also included by the MANIFOLD compiler in all manifolds, to deal with a set of predefined system events.

The manifold processor finds the appropriate handler block for an observed event *e* raised by the source *s*, by performing a circular search in the list of block labels of the manifold. The list of block labels contains the labels of all blocks in a manifold in the sequential order of their appearance. The circular search starts with the labels of the current block in the list, scans to the end of the list, continues from the top of the list, and ends with the labels of the block preceding the current block in the list.

The manifold processor in a given manifold is sensitive to (i.e., interested in) only those events for which the manifold has a handler. All other events are to be ignored. Thus, events that do not match any label in this search do not affect the manifold in any way. Similarly, if the appropriate block found for an event is the keyword **ignore**, the observed event is ignored. Normally, events handled by the current block are also ignored.

The concept of an event in MANIFOLD is different than the concepts with the same name in most other systems, notably simulation languages, and CSP. Occurrence of an event in MANIFOLD is analogous to a flag that is raised by its source (process or port), *irrespective* of any communication links among processes. The source of an event continues immediately after it raises its flag, independent of any potential observers. This raised flag can potentially be seen by any process in the environment of its source. Indeed, it can be seen by any process to which the source of the event is *visible*. However, there are no guarantees that a raised flag will be observed by anyone, or that if observed, it will make the observer react.

3.3. Block Labels

Event handler block labels are patterns designating the set of events captured by their blocks, separated by colons (:) from each other and from their blocks. Blocks can have multiple labels and the same label may appear more than once marking different blocks. Block labels are filters for the events that a manifold will react to. The filtering is done based on the event names and their sources. Event sources in MANIFOLD are either ports or processes.

The most specific form of a block label is a dotted pair *e.s*, designating event *e* from the source (port or process) *s*. The wild-card character * can be replaced for either *e*, or *s*, or both, in a block label. The form *e* is a short-hand for *e.** and captures event *e* coming from any source. The form **.s* captures any event from source *s*. Finally, the least specific block label is *.* (or *, for short) which captures any event coming from any source.

---

† In fact, the semicolon operator is only an infix *manner call* (see §3.5), rather than an independent concept in MANIFOLD. However, for our purposes, we can assume it to be the equivalent of the sequential composition operator in a language like Pascal.

3.4. Visibility of Event Sources

In each block, the manifold processor can react to only those events coming from sources (processes or ports) that are *visible* in that block. The visibility rule states that only those sources whose names appear in a block are visible from that block. This means that, while the manifold processor is in a block, except for the manifold itself, no process or port other than the ones named in that block can be the source of events to which it reacts. There are other rules for the visibility of parameters, operands of certain primitive actions, and it is also possible to define certain processes as permanent sources of events that are visible in all blocks. A manifold can always internally raise an event that is visible only to itself via the **do** primitive action.

Once the manifold processor enters a block, it is immune to any of the events handled by that block, except if the event is raised by a **do** action in the block itself. This temporary immunity remains in effect until the manifold processor leaves the block.

3.5. Manners

The state of a manifold is defined in terms of the events it is sensitive to, its visible event sources, and the way in which it reacts to an observed event. Such states collectively define the behavior of a manifold. It is often helpful to abstract and parameterize some specific behavior of a manifold in a subroutine-like module, so that it can be invoked in different places within the same or different manifolds. Such modules are called *manners* in MANIFOLD.

A *manner* is a construct that is syntactically and semantically very similar to a manifold. Syntactically, the differences between a manner definition and a manifold definition are:

1- The keyword **manner** appears in the header of a manner definition, before its name.

2- Manner definitions cannot have their own port definitions.

Semantically, there are two major differences between a manner and a manifold. First, manners have no ports of their own and therefore cannot be connected to streams. Second, a manner invocation never creates a new processor. A manifold activation always creates a new processor to "execute" the new instance of the manifold. To invoke a manner, however, the invoking processor itself "enters and executes" the manner.

The distinction between manners and manifolds is similar to the distinction between procedures and tasks (or processes) in other programming languages. The term *manner* is indicative of the fact that by its invocation, a manifold processor changes its own context in such a way as to behave in a different manner in response to events.

Manner invocations are dynamically nested. References to all non-local names in a manner are left unresolved until its invocation time. Such references are resolved by following the dynamic chain of manner invocations in a last-in-first-out order, terminating with the environment of the manifold to which the executing processor belongs.

4. The GKS Input Model

The purpose of the GKS input model is to supply a general interface to a host of different physical input devices[8, 10, 12, 13]. In this model there is a fixed number of logical input device classes (six, to be precise), which are characterized by the type of values they produce. For instance, there is a *locator device class* which returns a point in the world coordinate system (a user-configurable virtual coordinate system).

An actual realization of a logical input device is a member of its corresponding class whose behavior can be further specialized in a number ways by specifying prompt and echo type indicators; e.g., it is possible to have a locator with a rubber-band or one with a cross-hair, etc.

An important characteristic of the GKS input model is its *operating mode* facility. Each logical input device, regardless of its class, can operate in three different modes:

request mode

In the request mode, the application asks for a value from a logical input device and blocks until the value is produced. An operator must indicate explicitly when the device produces a

value by means of a trigger (e.g., a mouse button). The interval during which the operator may use the trigger starts when the application asks for a value and ends when the operator uses the trigger for the first time after that.

sample mode

In the sample mode, the application can at any time read the current value of a logical input device instantaneously, without blocking. In this mode there is no need for the operator to trigger the device.

event mode†

Here, as in the request mode, an operator must trigger the device, but the produced value(s) are gathered in a central queue. (There is only a single central queue in GKS for all devices operating in the event mode.) The application gets its input values from this queue, so it is not necessarily blocked during an input operation. The interval during which an operator may use the trigger for a logical input device in this mode starts when the event mode is entered for that device and lasts until an explicit departure from the event mode for the device. The trigger may be used as many times as the operator wishes in this interval. This mode allows much more flexibility in the order of triggering different input values by the operator.

Every GKS logical input device has a *measure* process and a *trigger* process, and its behavior is described in terms of these processes. Both processes can monitor several lower level (sometimes physical) devices.

A measure process keeps track of the current state of a logical device by mapping values obtained from a low level input device onto its own state value. A separate process is responsible for producing an echo on the screen. The prompt and echo type parameters of a GKS logical input device determine which of a predefined set of echo processes is to be used. Thus echoing is a hidden activity of a logical input device which can be influenced by an application program only to a limited extent and in a prescribed manner. (For instance, it is not possible for an application program to specify that changes to the current position of a locator device are to be echoed on the screen using an animated cat, rather than the static shape of a humble cross-hair.)

A trigger process must give the operator the possibility to indicate a particular instant in time. For example, pressing a mouse button by itself is merely a timing signal which does not have any other contents.

With these concepts the operating modes can be described as follows. Whenever a logical input device is active, there is also an active measure process supplying it with its state value. In the request mode there is also an active trigger process. The moment the trigger process produces its signal, the device returns its current state value. After producing one value the processes are stopped. In the sample mode the trigger process is not used. When a device is sampled, it produces its current state value as its result. Finally, in the event mode, both measure and trigger processes are used and operate as in the request mode, except that now multiple values may be produced. These values are delivered to a central event queue of GKS.

All processes are activated upon a read request in request mode, and at mode setting for other modes. The control structure imposed on processes in the GKS input model is thus non-trivial. However, note that the exact semantics of terms like *stopped* as used in the GKS standard documents is undefined: it can be interpreted to mean *suspended*, *deactivated*, or *killed*. Similarly, *activation* of a process can be interpreted as *resumption* or *creation* of a new instance.

---

† Caveat: The term *event* is used both in GKS and in MANIFOLD, designating completely different concepts. In this paper, we refrain from using the term *event* to denote its GKS meaning, except in *event mode* and *event queue*. Thus, throughout this paper, *event* is always a MANIFOLD event, but *event mode* and *event queue* are GKS concepts.

5. A MANIFOLD Implementation of the GKS Input Model

In the previous section we outlined the official GKS description of a logical input device. Our implementation of the GKS logical input device contains MANIFOLD processes named after those mentioned in the GKS standard. However, since we explicitly implement the communications that are verbally described in the standard, we introduce a few extra processes as well. For simplicity, we do not explicitly deal with such GKS concepts as input classes and prompt-echo types, etc. These features can easily be added, but they are not relevant to our discussion.

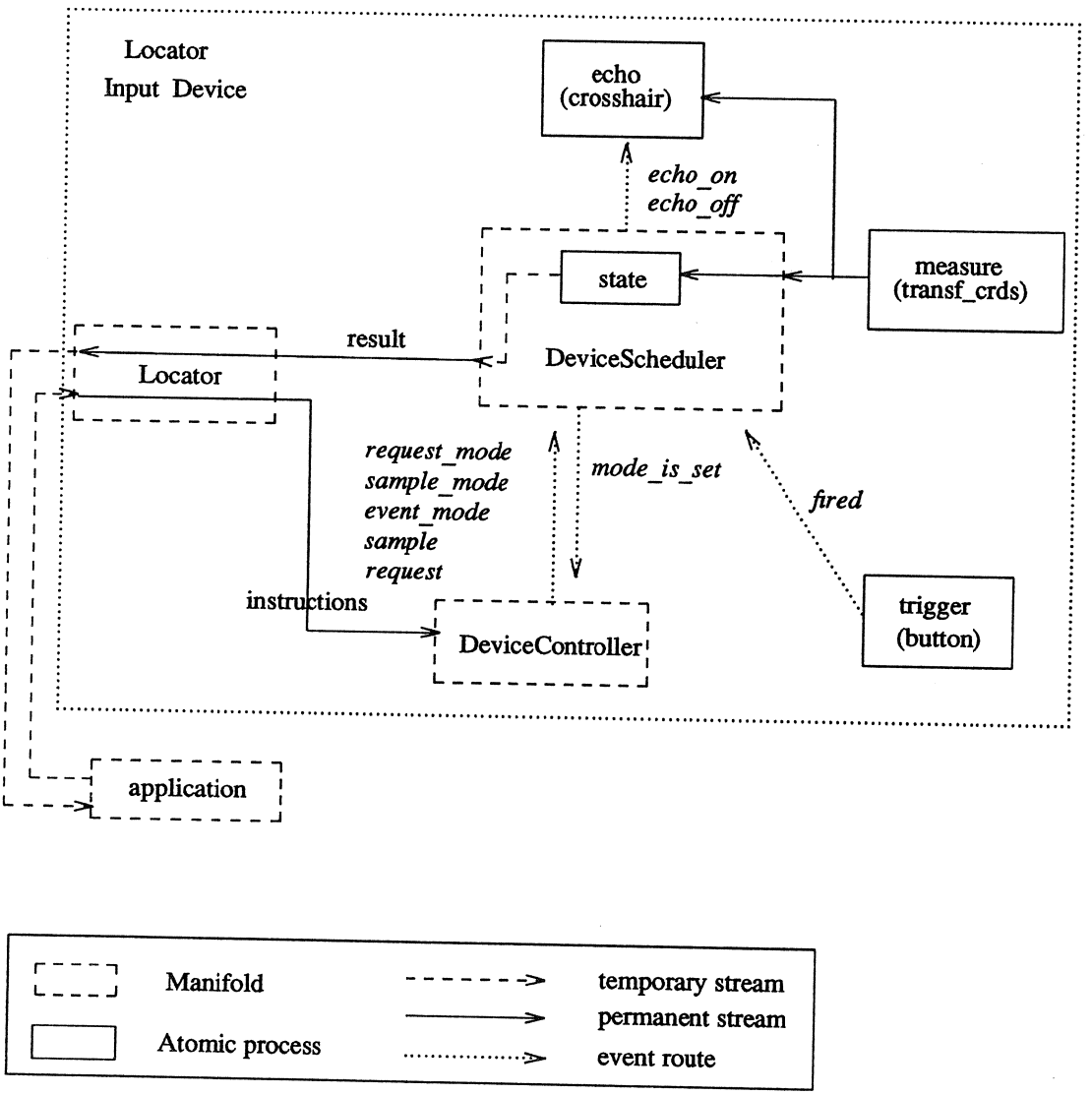

Figure 2 - The Process Structure

In Section 5.1, we first present our implementation model of a GKS logical input device. To be specific, we use a locator device as an example. A GKS locator device is a reasonable choice here because all other GKS input devices involve either equivalent or a subset of the components and connections used for a locator. The structure and most of the components presented in Section 5.1 are generic and can be used for other GKS logical input devices as well. Section 5.2 describes the details of one of the major generic components, the device scheduler manifold. The details of how various specific and generic processes

are put together to produce the functionality of a locator device is explained in Section 5.3.

5.1. General Structure of an Input Device

Figure 2 gives an overview of the relations among processes used for a GKS locator input device with a specific prompt-echo type (we will come back to this example later). There are three types of communications depicted in Figure 2: two types of streams and the so-called event routes. Temporary streams are normally used in this example for transfer of only one unit, after which they are removed. Permanent streams exist for as long as a logical input device exists. Note that this difference between the two types of streams is only a practically useful distinction made in this example; there is no difference between the two that MANIFOLD is aware of. The event routes show the intended trajectories of certain events. Whether or not an event is noticed by a receiver depends on its state: in some cases it might ignore them. To indicate this, the lines for event routes are not fully connected to the receiver. Note that event trajectories are also meaningless in MANIFOLD: they too are only a practically useful concept in the context of this application.

Two types of processes are shown in the Figure 2: manifolds and atomic processes. MANIFOLD views an atomic process as a black box. This simply means that the internals of an atomic process are irrelevant to the application. Atomic processes may represent pieces of hardware, processes written in other languages, etc. It may indeed be desirable to replace some of the atomic processes in our example with manifolds. Nevertheless, we refrain from doing so in this paper because such substitutions are mostly irrelevant for our purposes.

We make another distinction between generic processes and device specific processes. Generic processes are used in the construction of every logical input device. Specific processes are substituted for others in order to create the required variety of logical input device classes and prompt-echo types. Clearly the process `Locator` is specific: it is a specific type of a GKS locator. The same is true for the specific echo process (`crosshair`), the measure process (`transf_crds`), and the trigger process (`button`) used in `Locator`. The generic processes are `state`, `DeviceController`, and `DeviceScheduler`.

The intention for most of the connections in Figure 2 is rather obvious. For instance, the output of the measure process should be directed both to the echo process and to the state process (which stores its last received value). The presence and absence of echoing is controlled by the manifold `DeviceScheduler` through raising events `echo_on` and `echo_off`. The trigger process raises the event `fired` whenever an operator pushes the proper mouse button.

The manifold `DeviceController` controls the operation of the logical input device by transforming instructions that come in from the application program into their corresponding events. The generic process `DeviceScheduler` expects to detect logical input device mode setting requests, as proper events raised by `DeviceController`. The manifold `DeviceScheduler` is the most essential part of the logical input device. It coordinates all actions especially those related to the operating modes. We examine its details more closely in the next section.

5.2. The Device Scheduler Manifold

The manifold `DeviceScheduler` describes the interaction behavior of the processes which make up a logical input device. There are four major stages in this manifold. The first one is when no specific operating mode has yet been chosen for a device. The other three correspond to the GKS logical input device operating modes.

Listing 1 contains the code for the main level of `DeviceScheduler`. There are quite a few things to note here. The parameters in the header refer to processes created elsewhere. The process `controller` is the source of events which direct the behavior of this manifold. `trigger` corresponds the earlier mentioned GKS concept. The values from the measure process are via the input port `measures` available inside the manifold. The event declaration statements indicate what sort of events this manifold will accept or send.

Next is the body of the `DeviceScheduler` manifold, enclosed in a pair of braces. It starts with some private declarations, followed by its event handling blocks. In its private declarations, first we see a definition of a process named `state` which is an instance of `variable`. A variable is a MANIFOLD

```
DeviceScheduler(controller, trigger)
process  controller, trigger.
port in  measures.
event request_mode, sample_mode, event_mode.
event echo_on, echo_off.
{
      process state  is variable.
      permanent  controller.
      permanent (measures -> state).

      start:
            activate state;
            idle.
      request_mode:
            RequestMode(state, trigger).
      sample_mode:
            SampleMode(state).
      event_mode:
            EventMode(state, trigger).
      end:
            deactivate state.
}
```

Listing 1 - Input Device Manifold

process that remembers the last value it receives on its input port, and copies it on its output port whenever it is connected to a "reader" process. The first permanent statement declares the process named controller to be a permanent source of events through the lifetime of this manifold. The second permanent statement sets up a pipeline to store the last value produced by the measure process into process `state`. This is accomplished by defining a stream (→) between port `measures` and the input port of `state`. We assume here that the measure process will later be connected to the input port `measures`.

The rest of the code defines different manifold blocks, each preceded by labels which denote events that cause a transition to that block. The event `start` is always raised automatically after a manifold is activated. In this case, the `start` block activates `state` and subsequently waits until another event arrives. The candidates for incoming events are the three mode setting events (the `end` event will be raised automatically inside the manifold just before it dies, e.g., in response to a terminate event).

Assume, for instance, that the event `request_mode` is received. Its associated block specifies that a manner named `RequestMode` must be called. A manner is a MANIFOLD construct which introduces sub-blocks into a manifold program. A manner call effectively changes the context of the calling processor, and thus its behavior, by pushing the previous context into a stack and entering a new one – that described by the manner. The previous context is restored upon exit from a manner.

Looking at the manner `RequestMode` in Listing 2 we note that it has the same syntactic structure as a manifold (there are minor differences). Note the two mode-specific events that are declared only for this manner. After the manner is entered the event `start` is raised automatically. This first raises the event `echo_off` (switches off the echo possibly left on by previous modes), and then raises the event `mode_is_set` which is needed for synchronization with the `controller` manifold (see Appendix B). The manner then waits for an event to arrive.

At this point the next event may be a `request` event. The corresponding block causes the process `trigger` to be declared as an active source of events by just naming it. The `trigger` process is expected to produce the event `fired` at the proper time indicated by the logical input device operator.

```
manner RequestMode(state, trigger)
action  state, trigger.
event request, fired.
{
        start:
                raise echo_off;
                raise mode_is_set;
                idle.
        request:
                raise echo_on;
                trigger.
        fired:
                raise echo_off;
                state -> pass1() -> output;
                idle.
}
```

Listing 2 - RequestMode Manner

Once the trigger process has fired, a value coming from `state` must be transferred to the output port of the manifold. To accomplish this, a (temporary) pipeline is created from the output port of `state` to the output port of the manifold. A pipeline is a set of consecutive streams which belong together. If one of the processes engaged in the pipeline dies, all the streams in the pipeline will be removed. By putting an implicitly activated instance of the process `pass1` – which dies after passing exactly one unit from its input to its output port – in between, the pipeline will transfer only one unit. Then the manifold waits until a next event is received.

Dynamic nesting of manner calls means that while a manifold processor is inside the `RequestMode` manner (e.g., in block `request` waiting for an event from the trigger), a mode change event will still be responded to properly. For instance, assume that at such a point in time a `sample_mode` event is raised. Because its event source (`controller`) is declared to be permanent in the caller of the `RequestMode` manner, it is allowed to preempt the current block (e.g., `request` in `RequestMode`). Since there is no handling block for `sample_mode` in `RequestMode`, the manner call terminates and the processor returns to its previous (i.e., the calling) context. Popping of the stacked contexts of manner calls continues until either a handling block is found for the outstanding event, or the context of the manifold is reached. In case of our example, we find a handling block for `sample_mode` at the manifold level. Thus, the `DeviceScheduler` manifold exits `RequestMode` and enters `SampleMode`, as specified in the handler block for `sample_mode` in Listing 1.

We will not discuss the other two operating modes in detail here, but only mention their differences with the request mode. The actual code for these modes can be found in Appendix A. In the sample mode, there is no trigger process. Instead, the moment for returning a state value is determined by the arrival of the event `sample` raised by the controller process. In the event mode, the trigger process remains a permanent source of the `fired` events, and the value of `state` is directed to a process representing the global event queue of GKS (`gks_event_queue`).

5.3. A Locator Device

The manifold and manners described in the previous section are more concerned with the generic part of logical input devices. As an example of how an actual input device may be implemented, we take the simplified locator device which was depicted earlier in Figure 2. A device of class locator returns a point in a virtual coordinate system called the world coordinate system. The official GKS model allows several world coordinate systems but for simplicity we assume only one.

```
process button  is ButtonDevice.        /* hardware button device */
process pointer is PointerDevice.       /* hardware pointer device */

Locator()
port in instruct.
{
      process scheduler    is DeviceScheduler.
      process crosshair    is Crosshair.
      process transf_crds  is TransformCoordinates.
      process controller   is DeviceController.

      permanent instruct -> controller.
      permanent pointer -> transf_crds ->
                  (-> crosshair, -> scheduler.measures).
      permanent scheduler -> output.

      start:
            activate controller(scheduler).
            activate scheduler(controller, transf_crds, button);
            activate crosshair(scheduler);
            activate transf_crds();
            idle.
      end:
            deactivate scheduler, crosshair, transf_crds.
}
```

Listing 3 - Locator Manifold

The manifold `Locator` is shown in Listing 3. This manifold serves as the interface for the application. The application puts instructions in the form of strings on `Locator`'s input port `instruct` (which are subsequently forwarded to the `DeviceController`). Results (coming from the process `DeviceScheduler`) are sent back to the application through the standard output port of `Locator`. The proper process structure for our locator device is defined by `Locator`. It first declares the processes that are required for the device. The echo process `crosshair` takes care of the echoing for the device. The process `transf_crds` transforms device coordinates into world coordinates and serves as the measure process for our locator. The atomic processes `button` and `pointer` are supposed to be already active. They represent the actual (e.g., hardware) trigger and the lower-level (e.g., hardware) position or displacement register, respectively.

Next a number of streams are defined, of which most can be traced back to the connections shown in Figure 2. The stream connecting `pointer` to `transf_crds` is the only stream not shown in Figure 2. It supplies the measure process of our locator with its raw input.

6. Extensions and Modifications

One of the advantages of using MANIFOLD is that the resulting programs embody very flexible structures. In our case at hand, new logical input devices can be defined very easily by reuse of the generic manifolds. To create a new input device, it suffices to define a few processes and some stream connections as in `Locator`. The ease of definition of new devices is significant because it allows programmers to construct their own (hierarchical) devices. In the following sections we give a number of examples of how the original implementation can be modified to add extra features or change its behavior.

We distinguish between three types of modifications. First, through a simple example, we show how

intercepting the information flow (in this case out of the measure process) can be used to modify behavior. Then some variations on device echoing are discussed. These two types of modifications are in an abstract sense, "minor" changes. The last set of examples goes into the core of the operation of the logical input device and deals with concepts such as unbuffered information transfer.

6.1. Information Filters

Filtering the information flow that comes out of the measure process can be useful in a number of situations. For instance, to make certain items of a choice device not selectable can be achieved by mapping their corresponding numbers to a special number that indicates no-selection.

In the context of our locator device, enforcing that the return values of a locator device must always lie on some grid in the world coordinate system is another example where filtering is useful. In this case, not only the device must return the properly rounded values, the echo process must also show the same correct values. This can be accomplished by replacing the group in `Locator` by the following line:

```
(pointer -> transfer_crds -> grid ->
  (->crosshair, ->scheduler.measures))
```

The process `grid` is the filter that rounds off a point in the world coordinate system to its closest point on a grid. Both the scheduler and the echo process `crosshair` receive the same rounded points and thus echoing is done as desired.

6.2. Reaction Modifications

The examples in the previous section intercepted and modified the information coming out of the measure process. The examples in this section involve changes to the way in which a logical input device must react to some information, for instance to support a more complex echoing. The following structure can be used to get several simultaneous echoes:

```
(pointer -> transfer_crds -> ( ->echo1, ->echo2, ->echo3 ... ,
  ->scheduler.measures) )
```

Each echo process, `echo`*i*, receives a copy of the values coming from the measure process. An echo process might also be preceded by a process that computes some predicate based on the incoming value. Depending on the value of a predicate `pred`*i*, its corresponding echo process can be turned on or off:

```
(pointer -> transfer_crds -> (-> pred1 -> echo1, -> pred2 -> echo2,
  ... ->scheduler.measures) )
```

The notification of when to turn an echo on or off is supposed to be sent over the stream by its corresponding predicate process.

We may wish to dynamically decide about which one on a set of echo processes is to be used. This is very useful, for example, when a display screen is divided into a number of subareas, each with its own associated cursor shape. Of course, this can be done by the above scheme, using a predicate process for each echo process. In that case, we must make sure that all predicate processes comply with a common switching protocol. It is practically more convenient to encapsulate such a switching protocol in a single process. This can be done by changing the line to:

```
(pointer -> transfer_crds -> (->switch, ->scheduler.measures),
  switch.ch1 -> echo1, switch.ch2 -> echo2, ...)
```

The process `switch` decides which of the several echo processes is to receive the measure values. Turning echo on and off could be done in the same way as in the previous example, i.e., the echo processes can be identical for the two examples. The `switch` process sends the measure values only to the process that is responsible for the cursor in the current area.

6.3. Operational Modifications

The following examples are concerned with the operation of the generic part of the logical input device. There are certain aspects of the implementation model that may cause undesirable effects in practice. In the real world everything takes time: transmission of information, processing of information, and even, simply becoming aware of an event. In MANIFOLD, only the relative speed of the transmission of units and events is guaranteed; there are no restrictions on the absolute delays of data transmission. Even more uncertain is the execution time of atomic processes, such as an echo process.

Our MANIFOLD implementation of a logical input device, as presented earlier, demonstrates an ambiguity in the GKS standard that can lead to annoying consequences. When the measure process of a logical input device produces values in a much higher rate than its echo process is able to handle, the connecting stream buffers them and the echo process will lag behind. Meanwhile, the scheduler actually gets the most recent measure value from the measure process. Thus, firing a trigger will cause a value to be returned that does *not* correspond to that of the echo. In principle, this may or may not be a problem: in some cases it may be desirable for the state of the logical input device to be synchronized with the echo, whereas in other cases, it may be more advantageous to have it synchronized with the measure, even if the echo lags behind. The most strict case where the state of the logical input device must be synchronized with both measure and echo may also be necessary in some applications. The GKS standard does not address this issue explicitly, and thus, can sanction either of the three alternatives.

To avoid the discrepancy between the measure and the echo processes, we can flush intermediate results when a newer measure value is available. To do this we construct a process called `flush` (shown in Listing 4) that provides the latest value received via its input port on its output port whenever it receives the event `new_value`.

The `flush` process has two manners, corresponding to whether or not `flush` has received a new value since the last time it sent out a value. When no new value has arrived and if the receiving process is ready to receive another value, a temporary stream is made directly from input to output. Otherwise the latest value is stored and later retrieved when a value is requested. Process `flush` is now inserted between the measure and the echo process:

```
(pointer -> transfer_crds ->
  ( -> flush -> crosshair, -> scheduler.measures) )
```

The process which decides when the next value is to be sent out (in this case `crosshair`), does so by raising the `value_request` event.

In the above example, the state of the logical input device is synchronized with the measure process. However, we can easily accommodate the situation where the state of the logical input device must be the same as the echo, simply by substituting the line:

```
(pointer -> transfer_crds -> flush ->
  (-> crosshair, -> scheduler.measures) )
```

Now the state of the device is updated at the the same moment as `flush` sends out a value (still on request of `crosshair`). If we are concerned with the performance we can also put `flush` earlier in the pipeline just before `transfer_crds`. This avoids unnecessary computations of `transfer_crds`.

7. Conclusion

This paper reflects our study of the GKS input model from an implementation point of view. Since the model was conceived in late seventies to early eighties, it is not particularly amenable to changes necessitated by recent advances in graphics. For instance, it lacks the flexibility to allow programmers and end-users configure their own customized logical input devices, and to take advantage of the special hardware facilities available on modern workstations.

Our MANIFOLD implementation of the GKS input model is simple and modular. It is smaller than its previous formalizations that have appeared in the literature, and yet, it covers more of the operational details of the model. It clearly shows some of the shortcomings and the ambiguities in the GKS standard

```
flush(echo)
process echo.
{
        process store is variable.
        permanent echo.

        start:
                do unchanged.
        unchanged:
                UnChanged().
        changed:
                Changed().
}

manner UnChanged.
{
        start:
                input -> pass1() -> store;
                do changed.
        value_request:
                input -> pass1() -> output;
                do unchanged.
}

manner Changed().
{
        start:
                input -> store.
        value_request:
                store -> pass1() -> output;
                do unchanged.
}
```

Listing 4 - Process Flush

documents, and we have pointed out the ways in which they can be resolved. One interesting observation is that our MANIFOLD implementation is flexible enough to accommodate all changes and extensions discussed in this paper with minimal effort, and with very little localized impact on the actual source code.

This exercise shows that the MANIFOLD language is useful for describing the complex interactions among a set of cooperating concurrent processes that comprise non-trivial user interfaces. It also suggests that MANIFOLD may be a suitable framework for defining the future graphics packages that must accommodate user reconfigurability. Of course, both of these issues need to be examined more elaborately as we gain more practical experience with the MANIFOLD language and use it in more applications.

Appendix A: Manifold: DeviceScheduler

```
process gks_event_queue is Queue.        /* Global event queue */

DeviceScheduler(controller, trigger)
process  controller, trigger.
port in  measures.
event request_mode, sample_mode, event_mode.
event echo_on, echo_off.
{
    process state  is variable.
    permanent  controller.
    permanent (measures -> state).

    start:
        activate state;
        idle.
    request_mode:
        RequestMode(state, trigger).
    sample_mode:
        SampleMode(state).
    event_mode:
        EventMode(state, trigger).
    end:
        deactivate state.
}


manner RequestMode(state, trigger)
action  state, trigger.
event request, fired.
{
    start:
        raise echo_off;
        raise mode_is_set;
        idle.
    request:
        raise echo_on;
        trigger.
    fired:
        raise echo_off;
        state -> pass1() -> output;
        idle.
}


manner SampleMode(state)
action  state.
event sample.
{
    start:
        raise echo_on;
        raise mode_is_set;
```

```
        idle.
    sample:
        state -> pass1() -> output;
        idle.
}


manner EventMode(state, trigger)
action  state, trigger.
event fired.
{
    permanent trigger.

    start:
        raise echo_on;
        raise mode_is_set;
        idle.
    fired:
        state -> pass1() -> gks_event_queue;
        idle.
}
```

Appendix B: Manifold: DeviceController

```
DeviceController(scheduler)
process scheduler.
{
    event request_mode, sample_mode, event_mode.
    event request, sample.

    start:
        case( $getunit(input),
            "request mode", set_new_mode(scheduler, request_mode),
            "sample mode",  set_new_mode(scheduler, sample_mode),
            "event mode",   set_new_mode(scheduler, event_mode),
            "request",      raise request,
            "sample",       raise sample,
            "stop",         do end);
        do start.
    terminate:
        ignore.
    end:
        deactivate parent.

}


manner set_new_mode(scheduler, mode_event)
action scheduler.
event  mode_event.
event  mode_is_set.
{
    start:
        (scheduler, raise mode_event).
    mode_is_set:
        return.
}
```

Appendix C: Manifold: Locator

```
process button  is ButtonDevice.        /* hardware button device */
process pointer is PointerDevice.       /* hardware pointer device */

Locator()
port in instruct.
{
    process scheduler    is DeviceScheduler.
    process crosshair    is Crosshair.
    process transf_crds  is TransformCoordinates.
    process controller   is DeviceController.

    permanent instruct -> controller.
    permanent pointer -> transf_crds ->
              (-> crosshair, -> scheduler.measures).
    permanent scheduler -> output.

    start:
        activate controller(scheduler).
        activate scheduler(controller, transf_crds, button);
        activate crosshair(scheduler);
        activate transf_crds();
        idle.
    end:
        deactivate scheduler, crosshair, transf_crds.
}
```

References

1. ARBAB, F. AND I. HERMAN, "Examples in Manifold," Technical Report CS-R9066, Centre for Mathematics and Computer Science (CWI), Amsterdam (1990).
2. ARBAB, F. AND I. HERMAN, "Manifold: A Language for Specification of Inter-process Communication," Technical Report CS-R9065, Centre for Mathematics and Computer Science (CWI), Amsterdam (1990).
3. ARBAB, F., "Specification of Manifold," Technical Report (in preparation), Centre for Mathematics and Computer Science (CWI), Amsterdam (1991).
4. ARNOLD, D.B. AND D.A. DUCE, *ISO Standards for Computer Graphics: The First Generation*, Butterworths, London (1990).
5. DUCE, D.A., R. VAN LIERE, AND P.J.W. TEN HAGEN, "Components, Frameworks and GKS Input," in *Eurographics'89 Conference Proceedings*, ed. W. Hansmann, F.R.A. Hopgood and W. Straßer, North Holland, Amsterdam (1989).
6. DUCE, D.A., "GKS, Structures and Formal Specification," in *Eurographics'89 Conference Proceedings*, ed. W. Hansmann, F.R.A. Hopgood and W. Straßer, North Holland, Amsterdam (1989).
7. DUCE, D.A., R. VAN LIERE, AND P.J.W. TEN HAGEN, "An Approach to Hierarchical Input Devices," *Computer Graphics Forum* 9, pp. 15-26 (1990).
8. ENDERLE, G., K. KANSY, AND G. PFAFF, *Computer Graphics Programming - GKS (Second Edition)*, Springer Verlag, Berlin - Heidelberg - New York - Tokyo (1987).
9. HOARE, C.A.R., *Communicating Sequential Processes*, Prentice–Hall , New Jersey (1985).
10. HOPGOOD, F.R.A., D.A. DUCE, J.R. GALLOP, AND D.C. SUTCLIFFE, *Introduction to the Graphical Kernel System GKS*, Academic Press, London - New York (1983).
11. HOWARD, T. L. J., W. T. HEWITT, R. J. HUBBOLD, AND K. M. WYRWAS, *A Practical Introduction to PHIGS and PHIGS PLUS*, Addison-Wesley, Wokingham-Reading (1991).
12. ISO,, "Information processing systems - Computer graphics - Graphical Kernel System for Three Dimensions (GKS-3D) functional description," ISO8805 (1988).
13. ISO,, "Information processing systems - Computer graphics - Graphical Kernel System (GKS) functional description," ISO IS7942 (1985).
14. ISO,, "Information processing systems - Computer graphics - Interfacing techniques for dialogues with graphical devices (CGI) functional description," ISO DP 9636/1-6 (1988).
15. ISO,, "Information processing systems - Computer graphics, Programmer's Hierarchical Interactive Graphics System (PHIGS) - Part 1, Functional description," ISO/IEC9592-1 (1988).
16. ISO,, "Information processing systems - Computer graphics, Programmer's Hierarchical Interactive Graphics System (PHIGS) - Part 4, Plus Lumière und Surfaces (PHIGS PLUS)," ISO/IEC9592-4, rev. 3 (1989).