Logical input devices and interaction.

R. van Liere, P. J. W. ten Hagen

Department of Interactive Systems
Center for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The *logical input device* model, as is adopted in the standardized graphics packages GKS, has been an accepted basis for producing device-independent graphics systems. However, when used in highly *interactive* graphical applications, the logical input device model does not provide sufficient support for a number of fundamental issues inherent to interaction. This paper reopens a discussion which questions the functionality provided by the logical input device model when brought in conjunction with interaction. In particular, the logical input device model does not support the notion of input/output symmetry.

CR Categories: I.3.4 **[Computer graphics]**: Graphics utilities- *graphics packages*; I.3.6 **[Computer graphics]**: Methodology and techniques- *device independence, interaction techniques*

Key Words & Phrases: Logical input devices, dialogue systems, dialogue programming.

1. Introduction.

The logical input device model, as is adopted by the standardized graphics packages GKS ([. GKS Information processing system .]), is now well known in the computer graphics community. Its main achievement is that it provides an application program with a model in which physical input devices are addressed as abstract data types. This is accomplished by defining six different input classes in which each class is characterized only by a predefined data type. The operational behaviour of each logical device is also be described in a abstract way. It is left as an implementation issue how each logical input device is mapped to a physical device.

More recently, various dialogue systems have been developed that share the property of being built "on top of" a device independent graphics package. A dialogue specification can be seen as the definition of an input process in terms of sequences upon input data structures and the corresponding (syntactical as well as semantical) feedback. Technically speaking, one of the most salient issues that any successful dialogue system will have to cope with is the support it offers to precisely specify the real-time behaviour of the input process (in particular, to what level of detail can the input process of the dialogue be specified). On the one hand, abstraction mechanisms are desirable while on the other, it should be possible to address very low level (even representation dependent) features of the input process. Typical problems encountered with low level dialogue programming include : (i) the structuring of input devices, (ii) the specification of the syntactical and semantical feedback and (iii) the interfacing

with the application program.

The approach taken by most dialogue systems is to provide an additional library of so-called interaction techniques which can be called from the executing dialogue (cf. [. pfaff operator interfaces based on logical input devices .], [. pfaff user interface management systems .], [. thomas graphical input techniques workshop .]). It is, however, very clear that the quality of the dialogue can be measured by the way these interactive techniques present themselves to the user. For instance, does the technique allow the user to perform a task in a natural way? One of the most important aspects that will influence the presentation of an interactive technique is its ability to react to new situations in the dialogue. This is in particular the case in dialogues in which more than one task is being executed by the user. Is the interactive technique used in one task able to take the changes into account which were caused by the other task?

Even before it was proposed in its final form, the logical input device model has attracted severe criticism (cf. [. guedj interaction .], [. rosenthal pfaff semantics input devices .]). The most disputed issues concentrated on the fact that (i) there exits a lack of uniformity among the different logical device classes as to the details of their behaviour and (ii) the difficulty of relating "output" concepts to logical input devices. If this critique is justified, then the corresponding issues in question will also be apparent when used in the context of interaction. This paper illustrates these issues by presenting some typical interactive scenarios in which the input device model is inadequate.

This paper is organized as follows. We assume that the reader is familiar with the basic properties of the logical input device model. † First, in order to set a context for our criticism, in section 2 we discuss a simplified version of the logical input device model. The model is a simplified in the sense that only relevant (for interaction) aspects are included and numerous irrelevant details are left out. Since interaction is the interplay between input and output, we must also give a simplified output model. We chose the GKS output model as a basis for this. Moreover, we discuss some support which GKS provides that interrelate the two models and point out a number of shortcomings of this support when used in interactive systems. Section 3 places these shortcomings and drawbacks in a more general context. Suggestions for what functionality the input model must provide in order to overcome these drawbacks are also made. It is, however, beyond the scope of this paper to propose an alternative approach (see [. hagen liere model interaction .] for such a proposal).

We describe both the input and output models using a semi-formal notation. Although the notation is somewhat inconsistent and its correctness may be questioned, it does add to the precision of the argumentation brought forward during the paper.

---

† See [. enderle computer graphics programming .] or [. duce hopgood introduction .] for a general introduction to logical input devices. Appendix 2 gives a summary of the functionality of logical input devices.

2. GKS-like systems.

2.1. GKS output model.

This section introduces a simplified model describing the output functionality of a GKS-like system. It is not our intention to encapsulate the complete GKS output functionality. However, the model does include the mechanisms that GKS provides for dynamically manipulating pictures, i.e. those pictures that can change during interaction. In GKS, only pictures created in segments can be dynamically manipulated. The model therefore considers only segments and not non-retainable data. Moreover, minor details such as the precise appearance of a particular output primitive (for instance, what it means when a primitive is *red*, has *thick* lines, etc.) will not be taken into account. These details, although indispensable in a practical implementation of GKS, are of no importance for dynamic aspects of picture manipulation.

GKS has a fairly complex three step segment creation mechanism. First, a segment must be opened. This results in a unique segment header which contains a set of segment attributes. Then, output primitives together with their attributes are entered into segments resulting in a segment body. Finally, the segment must be closed. After creation of a segment, the output primitives within the segment body cannot be altered. Only the attributes in the segment header can be given new values. Due to the static nature of a segment body, the output model simplifies the three-step segment creation process to a one-step creation of complete segments.

The simplified output model considers four categories of functions which control the appearance of a picture. The function categories included are:

(i) functions for the creation / deletion of segments
(ii) functions for changing segment attributes
(iii) functions for changing workstation attributes
(iv) functions for changing the update state.

The second and third categories are considered dynamic because they can alter an already existing picture rather than add or delete something.

GKS also maintains a graphical output state which stores the results of the output functions. The graphical output state is denoted as the triple $G_O = \langle S, W, U \rangle$ in which: †

- $S = \{ \langle S_{h_i}, S_{b_i} \rangle \mid i \in N_S \}$ is a set of segments. Each segment is uniquely identified with an segment name from the index set N_S. Every segment, with name i, consists of a header and a body, denoted as S_{h_i} and S_{b_i}. A segment header, denoted as $S_{h_i} = \langle TRANS, PRIO, VIS, DET, HIL \rangle$, consists of a set of attributes which include the transformation, priority, visibility, detectability and highlighting of the segment. We will not elaborate on the contents of a segment body other than that it consists of a set of output primitives and output attributes.

† A glossary of types and operator symbols used in this paper is given in Appendix 1.

GKS states that only the segment header of each segment can be changed dynamically. Once a segment body has been created it cannot be changed. This implies that "editing" an output primitive within a segment body is not possible.

- $W = < \{ Tr_i \mid i \in N_T \}, \{ B_{t,j} \mid t \in P \, , \, j \in N_B \} >$ is the workstation state, consisting of a set of normalization transformations and a set of attribute bundles. Normalization transformations and bundles are identified by unique names which are simply indices within corresponding index sets, denoted as N_T and N_B respectively. P is the set of output primitive types, consisting of *{polyline, polymarker, fillarea, text, cellarray}*.

- $U = <D, R, WS>$ is the update state consisting of two value fields, D and R, denoting the deferral state and regeneration mode respectively. Furthermore, there is the field, denoted as WS, which contains a set of segment names, bundle indices or normalization transformations which do not yet appear on the workstation; i.e WS can act as a buffer for graphical output. If $G_O[\, U\, [WS]] = \emptyset$, then the actual picture is completely "up to date".

  $D \in \{ASAP, BNIG, BNIL, ASTI\}$ and $R \in \{SUPPRESSED, ALLOWED\}$ are values which indicate in what way the contents of $G_O\, [U\, [WS]]$ can be controlled, i.e. D and R provide a means to delay the actual state of the picture as defined by the application program. For instance, by defining the deferral state $G_O\, [U\, [D]]$ to be *BNIL*, GKS will ensure that the actions necessary to achieve the visual effect of each output function are initiated before the next interaction local to the workstation.

The four function categories are described as:

- S-functions: New (S_i), Del (S_i)
  for creating and deleting respectively, a segment

- H-functions: New (S_{h_i})
  for changing (part of) the segment header of S_i.

- W-functions: New (Tr_i), New ($B_{t,j}$)
  for changing the various components of the workstation state.

- U-functions: New (D), New (R), Update ()
  for changing the deferral state, respectively the regeneration mode. The function Update() is used to explicitly ensure that $WS = \emptyset$, i.e. to explicitly execute all deferred actions.

2.2. GKS input model.

This section introduces a simplified model describing the input functionality of a GKS-like system.

GKS defines a number of logical input device classes. Each input device is characterized *only* by the data type it returns. How logical input devices are mapped onto physical devices is of no concern to GKS. It is hidden behind the device class, and thus is outside the scope of the input model.

The operational behaviour of a logical input device can be controlled at device initialization and activation time. Device initialization is achieved by setting various predefined logical input device attributes. Examples of these input device attributes are the echo area, the prompt echo type and the initial value. Analogously to output primitives in segments, the value of attributes settings cannot be changed after the initialization of the input device. A device is activated in one of three modes, each of which determines the way an input value becomes available to the application program:

- in **request**-mode a single input value is taken from an input register. Reading this value implicitly terminates the activation of the input device. In request mode reading implies that the application is suspended if no input value is available for that particular device.
- in **sample**-mode a single input value is also taken from an input register. However, the value will change when the device is operated. As a result, the value taken is the most recently produced one. In sample mode, reading does not terminate the activation of the input device.
- in **event**-mode the input value is taken from a FIFO (first in-first out) queue of input registers. Reading does not terminate the activation of the input device. In event mode, reading implies that the application program can be suspended if no input value is available in the queue. In this case, the amount of time that the application is suspended can be be controlled with a timeout parameter.

Due to the static nature of input device attribute setting, the model will consider device initialization and activation to be combined in one operation.

The GKS logical input device model can be summarized by describing a graphical input state by a tuple, $G_I = <L, M>$ in which:

- $L = \{L_{c,\,i} \mid c \in I_C,\ i \in N_c\}$ is a set of activated logical input devices. Each device belongs to an input class which characterizes the type of the result value. The six possible input classes are denoted as *CHOICE*, *LOCATOR*, *PICK*, *STRING*, *STROKE* and *VALUATOR*. Furthermore, each device is uniquely identifiable by a name from an index set N_c (i.e. $N_c \in \{N_{cho}, N_{loc}, N_{pic}, N_{str}, N_{ske}, N_{val}\}$).
- $M = \{M_{c,\,i} \mid c \in I_C,\ i \in N_c\}$ is a set of logical input device modes, including their initial state. $M_{c,\,i}$ denotes the activation mode of the corresponding logical input device $L_{c,\,i}$.

The two function categories which operate on the input state G_I are :

- M-function: New ($M_{c,\,i}$)
  for initializing and activating a logical input device.
- L-function: Read ($L_{c,\,i}$)
  for request, sample or read-event depending on the mode $M_{c,\,i}$.

2.3. GKS support for interaction.

A basic mistake when designing an interactive program is to regard the input and output functionality as two independent concepts. Rather than input functions, the application program should define interactive functions. These are functions that relate concepts of input to those

of output. In this section we show what support is offered by GKS to relate the output functionality (i.e. G_O) with G_I. We will focus on only three (generic) relations since it is not our intention to give a complete list. These relations are summarized as:

- The first type of relation considers the synchronization between input and output. This can be expressed as :

  foreach $G_I\ [L_{c,\ i}]$ **with** $c \in I_C \wedge i \in N_c$
  if ((is_reading ($G_I\ [L_{c,\ i}]$)) **and** ($G_O\ [U\ [D\ ]] = BNIL$))
  then $G_O\ [U\ [WS\ ]] = \emptyset$

  which states that for every logical input device, $L_{c,\ i}$, if (i) a Read is being done from that particular device and (ii) the deferral mode is BNIL then the workstation buffer (containing the not effectuated transformations settings, segment names and bundle indices) will be empty. †

  This is the only relation supported by GKS which ensures that the picture on the screen is "up to date" whenever an input value is to be read. The importance of providing a very fine control over the deferral state is that it allows an interactive function to present itself to the user without delay. Of course, the tradeoff between a continuously "up to date" picture and the number of updates to achieve that must be taken into account.

  Unfortunately, this relation does not provide sufficient control over the update state which seems to be necessary in real-time interaction. This is because the relation only states that, given certain conditions, the *complete* output state, G_O, is updated. It would be more appropriate to provide a mechanism that would allow a partitioning of $G_O\ [S\ ]$ *and* $G_O\ [U\ [WS\ ]]$ so that the interactive function would influence only a small portion of the outdated output state. This mechanism would allow the application program, by associating a segment partition to a particular interaction task, to ensure that the corresponding feedback is displayed very rapidly and, hence, defer less important graphical output.

- The second type of relation considers input values which refer to output elements. In our simplified GKS model, this is done with the pick device which returns a segment name. † This is expressed as:

  foreach $G_I\ [L_{pic,\ i}]$ **with** $i \in N_{pic}$
  Read ($G_I\ [L_{pic,\ i}]$) $\in \{\ x \mid (x \in N_S) \wedge (S_x \in G_O\ [S]) \wedge (S_{h_x}\ [VIS\ ]\ ) \wedge (S_{h_x}\ [DET\ ]\ )\ \}$

  which states that the return value from a pick device is the segment name of an existing segment which is both visible and detectable. In the case that no segment is picked,

---

† The GKS equivalent of this relation is actually a bit more complex since it takes the (possibly non-empty) event queue belonging to that particular input class into consideration.
† In GKS the pick mechanism returns a pick identifier as well as a segment name.

Read ($L_{pic,\ i}$) will return a null value. This is the only relation supported by GKS which allows a picture element in $G_O\ [S]$ to be a result value of a logical input device.

When used in an application program the pick mechanism reveals two severe deficiencies. Both are due to the fact that the pick device cannot influence $G_O\ [S]$ during during the time that it is active :

- ♦ The feedback given by the pick device when picking a segment is not in $G_O\ [S]$ although the resulting segment name is. For instance, one would expect that the segment picked is somehow highlighted by, say, placing a polymarker on the segment. The polymarker is, however, neither in G_O nor in G_I.

  In general, this is true for all prompts and echos given by *any* logical input device. The feedback produced by logical input devices is not defined in G_O or G_I. The approach that GKS uses to bypass this is to allow each logical input device to supply its own, implementation dependent, set of prompt/echo types. This approach is inadequate in the sense that it does not provide the application program with any mechanism to (re)define low level feedback given by the particular device.

- ♦ GKS does not define how the picking process obtains its measure value. For instance, what happens to the measure value of a pick device if the segment is deleted ?

  The point being made here is that the pick mechanism, which is defined within $G_I\ [L]$, operates independently of $G_O\ [S]$. It is an implementation dependent issue if changes within $G_O\ [S]$ are propagated towards the pick mechanism.

- The third type of relation is concerned with how attribute values from the output state are used to determine the operational behaviour of the input device. As an example, consider how a locator device returns a point and which transformation was used to get the result. This is expressed as :

  foreach $G_I\ [L_{loc,\ i}]$ **with** $i \in N_{loc}$
  if Read $(G_I\ [L_{loc,\ i}]) \in \{ <ntr_{maxp},\ pos> \mid ntr_{maxp} = maxp\ (G_O\ [W\ [\{Tr_i\}]],\ pos)\ \}$

  which states that the locator position is transformed by the transformation with highest priority whose clipping rectangle contains the position. Here *maxp*() is a function that, given a set of transformations and a particular position, will return the transformation with the highest priority which contains the position in its clipping area.

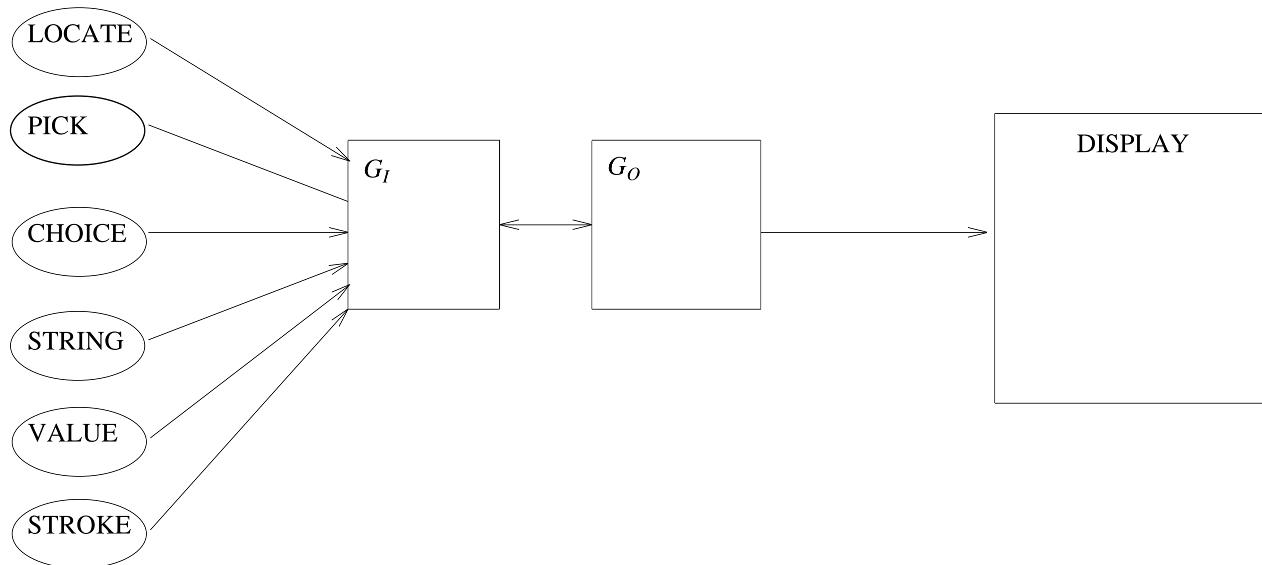
  The locator input mechanism does reveal a further deficiency in the input model. For instance, it is important that the set of normalization transformations $\{Tr_i\}$ is properly set-up (done through New (Tr_i)-functions) prior to a locator device activation. If two locator devices require different orderings for the same Tr_i *and* these settings are in conflict, then the locator devices cannot be simultaneously active.

  In general, this is true for all attributes in $G_O\ [W]$. Whenever a change is made in

$G_O\ [W]$, there is no mechanism within GKS to ensure that these changes are propagated to $G_I\ [M]$. Hence, the attribute values shared by $G_O\ [W]$ and $G_I\ [M]$ can be inconsistent.

3. A closer look.

The critique given in the previous section can be generalized by saying that a GKS-like graphical system can be defined as two state machines; each having their own states, G_I and G_O respectively. These two machines can communicate with each other only by various *ad hoc* relations. The integrity of these relations is guaranteed only at certain predefined moments in time which, on the one hand, can be influenced by setting values within the update state $G_O\ [U]$ and, on the other, at device initialization time in $G_I\ [M]$. We denote such a graphics system as $G = <G_O, G_I>$ with an additional set of predefined relations R. Such a graphics system is depicted in the following figure. G_I and G_O are logical states which describe the state of the physical input devices and the display.



$G = <G_O, G_I>$ plus an additional set of relations R.

We will now take a closer look at the critical issues by placing them in a more general context. We first illustrate in what ways G_O is related with G_I and then, vice versa, show how G_I is related with G_O.

- $G_O \rightarrow G_I$
  Suppose an input device, say $L_{c,\,i}$, is activated during the time interval :

  $$a,\ t_0,\ r_0,\ t_1,\ r_1,\ \cdots\ t_n,\ r_n,\ d$$

  in which a (respectively d) denote the activation (respectively deactivation) moment of

$L_{c,\,i}$. Each t_i (respectively r_i) with $i = 0 \cdots n$ denote moments in which *any* input device is triggered (respectively read). Note that $t_i < r_i$ is always true, whereas $r_i < t_{i+j}$ with $j > 0$ does not have to be true.

The first point is concerned with the consistency between $G_O\,[W]$ and $G_I\,[M]$ during the time interval $[a,\, d]$. The semantics of the device initialization and activation is such that device $L_{c,\,i}$ can be initialized with attribute values taken from $G_O\,[W]$ and (possibly) segment names taken from $G_O\,[S]$. However, during the time interval $[a,\, d]$, $L_{c,\,i}$ will be immune to any state changes within G_O. A more appropriate approach would be to allow the definition of arbitrary (application defined) relationships between any attribute within $G_O\,[W]$ and $M_{c,\,i}$. The graphics system would be responsible for ensuring the integrity of these relationships. In this way each $M_{c,\,i}$ will contain attribute values which are consistent with those found in $G_O\,[W]$ and $G_O\,[S]$.

The second point is concerned with the synchronization between $G_O\,[U\,[WS]]$ and $G_I\,[L]$. As was shown in the previous section, given that various conditions hold, GKS guarantees that $G_O\,[U\,[WS]] = \emptyset$ before each r_i. This, however, is unsatisfactory for real-time interactive applications. What is actually needed is a relation that guarantees that $G_O\,[U\,[WS]] = \emptyset$ before each t_i, since t_i (and not r_i) is the moment when the user triggers a device. †

- $G_O \leftarrow G_I$
  Alternatively, the logical input device model does not allow $G_I\,[L]$ to influence the output states $G_O\,[S]$, $G_O\,[W]$ and $G_O\,[U]$. For instance, the feedback given by any logical device does not reside in $G_O\,[S]$. Each $L_{c,\,i}$ has a private set of implementation dependent segments which the application program can chose from to be used as feedback.

  Ideally, any $L_{c,\,i}$ should be able to execute every output function. For instance, the displacement of the cursor would invoke a New (S_{h_i}) to indicate a new segment transformation. Similarly, highlighting could be done either by a New (S_i) or a New (S_{h_i}) call. It is essential, however, that the application program will have the ability to define new segments which in turn can also be manipulated by $L_{c,\,i}$.

What we are suggesting is a graphics system, denoted as $G^+ = \langle G_O,\, G_I,\, R \rangle$, in which R is a set of application defined relationships that relate arbitrary notions of G_O to those within G_I. These relationships can be augmented with pre- and postconditions which ensure that a state context is satisfied. It will be the responsibility of G^+ to ensure the integrity of these relationships. We say that such a graphics system supports the concept of *input / output-symmetry*, since both the input process as well as the output process can mutually influence each others graphical state.

The advantages of i/o-symmetry can summarized as:

† To be fair, it must be said that this suggestion is strictly theoretical since a system cannot anticipate the moment when the user triggers a device.

- Low-level feedback given by the logical input devices can be influenced by the dynamic aspects of a picture. This allows an interactive program to provide the user with application dependent feedback.
- Logical input devices can influence the dynamic aspects of a picture. This allows the user to "directly manipulate" the picture under construction.

It can be argued that some of the suggested functionality belongs in the dialogue system rather than in the graphics system. This can only be justified if the graphics system provides enough hooks to support the issues mentioned above. In the particular case of logical input devices, we have shown that this support is not sufficient.

4. Conclusions.

In this paper we have questioned the functionality provided by the logical input device model when used within the context of interaction. The issues brought forth were mainly caused by the fact that the input model is completely separated from the corresponding graphical output model. We have shown that the *ad hoc* relationships provided by GKS that interrelate these two models are insufficient and made some suggestions, which are based on the notion of input / output-symmetry, to improve the input model.

Appendix 1. Glossary of symbols.

Symbol	Meaning
$G_O = <S, W, U>$	output state
S	segment state consisting of a set of segments
$S_i = <S_{i_h}, S_{i_b}>$	segment, with name i, consisting of a header and body
W	workstation state consisting of a set of normalization transformation and a set of bundles
Tr_i	normalization transformation with name i
$B_{t, i}$	attribute bundle belonging to output primitive type t with name i
$U = <D, R, WS>$	update state
$G_I = <L, M>$	input state
$L = \{L_{c, i}\}$	set of logical input devices
$M = \{M_{c, i}\}$	set of logical input device modes
$L_{c, i}$	logical input device belonging to input class c
P	set of output primitive types
I_C	set of logical input classes
N_B	index set of attribute bundle names
N_S	index set of segment names
N_T	index set of transformation names
N_{cho}	index set of choice device names
N_{loc}	index set of locator device names
N_{pic}	index set of pick device names
N_{ske}	index set of stroke device names
N_{str}	index set of string device names
N_{val}	index set of valuator device names

Symbol	Meaning
$<x_1, \cdots, x_n>$	n-tuple consisting of components x_i
$\{ x \mid P \}$	set of x such that P is true
$X [x_i]$	selection of component x_i from tuple X
$\in$	set membership
$\wedge$	logical conjunction
$\vee$	logical disjunction

Appendix 2. Logical input devices.

The GKS input model is based on the concept of logical input devices. Logical input devices provide the application program with an interface which abstracts physical input devices from a particular hardware configuration.

A logical input device consists of:

- a *class*.
  The class of a logical input device defines the type of the input value which is returned. The six different classes are given in the following table :

device	returntype
locator	Wc, Ntran
choice	Choice
pick	Pickid
valuator	Value
string	String
stroke	Wc [1, · · · , n], Ntran

The GKS logical input classes.

  The actual number of logical devices in each class is workstation dependent. Each individual logical input device within a class is distinguished by a unique number.

- a *mode*.
  The activation mode indicates how the input value is obtained from the logical input device. Conceptually, there are always two processes running for each active logical input device; these are the so-called *measure process* and *trigger process*. A particular measure value of a logical input device is defined to be the (eventually transformed to world coordinates) value of the physical input device.
  The measure process will always contain the current measure value of the logical input device. Usually, the measure value is echoed in some way on the screen, (for instance, by echoing a cursor shape in the position that corresponds with the measure value).
  A trigger process is an independent, active process that, when *triggered* by the user, sends a message to the measure process. Triggering a logical input device indicates that the current measure value must be returned to the application.

  How the measure value is mapped onto a value returned by a logical input device is defined differently for every input class. For the *locator* device the mapping rules are:

  ♦ Transform the measure value (given in device coordinates) back to normalized device coordinates using the inverse of the current workstation transformation.

  ♦ Select the normalization transformation with the highest viewport input priority in whose viewport the normalized coordinate lies. The selection of a normalization transformation will always succeed since there is a default normalization transformation which covers the complete normalized device coordinate space.

- transform the normalized coordinate back to a world coordinate using the inverse of the selected normalized transformation.
- return the world coordinate and the number of the selected normalization transformation to the application program.

There are three different activation modes:

- *request*
  In the case of request mode, the application program will wait until the trigger process sends a message to the measure process. The value of the measure process at the moment of triggering will then (after the necessary transformations) be passed to the application program.
- *sample*
  In the case of sample mode, the value of the measure process will, at the moment of sampling, be passed to the application program. No triggering is involved when a logical device is sampled so that the application program will immediately continue after issuing a sample call.
- *event*.
  In the case of event mode, the application program will not wait until the trigger process sends a message to the measure process. However, when the logical input device is triggered the value of the measure process at the moment of triggering is put in an input queue. The contents of the queue can be acquired by the application program by issuing calls that query and get the queue elements.

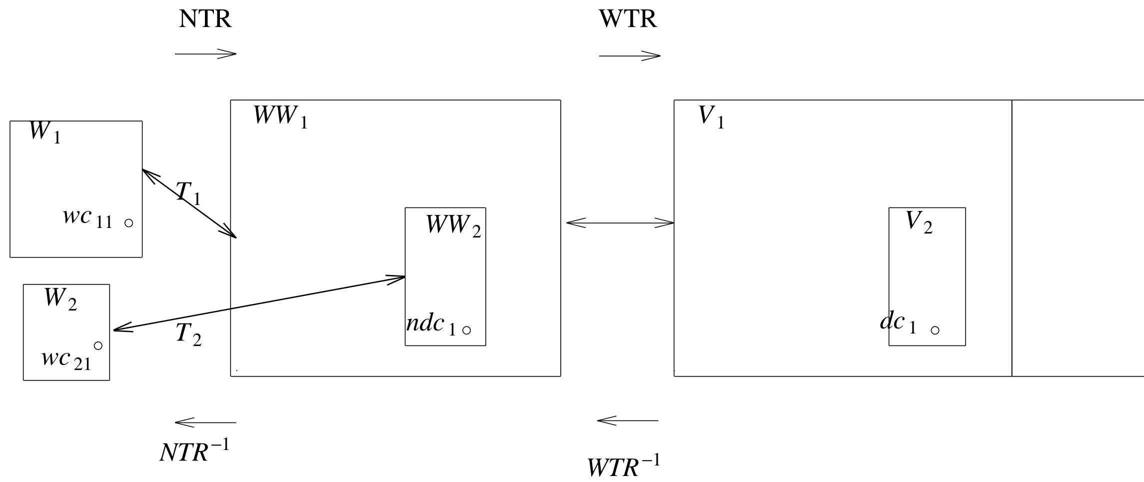
- *attributes*.
  Attributes are used to parameterize the initialization of a logical input device. Most attributes have to do with how and where input devices produce echoes on the screen. Attributes include initial values, prompt / echo types, activation modes and echo areas. Data records provide the application program a means to parameterize the logical input device in a device dependent manner. The layout of a data record must be precisely specified in the installation guide of a particular implementation. For instance, an entry in the data record could specify which mouse button will be used to trigger a locator device.

Example

This example illustrates how a value from *locator* input device is transformed back to world coordinates. Suppose an application program has defined two window / viewport transformations (T_i from W_i to WW_i) and uses the default workstation transformation at the moment the *locator* device is triggered. The following figure illustrates the different coordinate spaces and the relevant transformations. †

---

† For simplicity reasons, the default window / viewport transformation is not shown.



Input transformations.

If the physical locator device triggers at the point dc_1 then, in accordance with rule 1, the inverse of the current workstation transformation is used to calculate point ndc_1. There are now two cases which must be distinguished:

1. viewport input priority (T_1) > viewport input priority (T_2)
   Rule 2 selects T_1. The inverse of this transformation will calculate the point wc_{11}. Finally, wc_{11} and T_1 are returned to the application.
2. viewport input priority (T_1) < viewport input priority (T_2)
   Rule 2 selects T_2. The inverse of this transformation will calculate the point wc_{21}. Finally, wc_{21} and T_2 are returned to the application.

End of Example

References.