Design and Implementation of a Graphical Front-End

Brück, Dag M.

1987

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT–7367)/1-023/(1987)

Design and Implementation of a Graphical Front-End

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
August 1987

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
August 1987

Document Number
CODEN: LUTFD2/(TFRT-7367)/1-023/(1987)

Author(s)
Dag M. Brück

Supervisor

Sponsoring organisation
The National Swedish Board of Technical Development (STU)

Title and subtitle
Design and Implementation of a Graphical Front-End

Abstract

This report describes the design and implementation of a graphical front-end, i.e., a program that handles all input and output operations associated with the user for an application program. The front-end was designed to be general and easily portable.

In the report, special attention is paid to the design of segment handling, picking, rubberband shapes and menus. The front-end was implemented on a Silicon Graphics IRIS 2400 workstation, and problems with the window manager, the input queue and process communication are discussed. Lastly, a full description of available operations is given.

Key words
Computer graphics, Graphics standards, Window management, Segment handling, Picking, IRIS workstation

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
23

Recipient's notes

Security classification

1. Introduction

This report describes the design and implementation of a graphical front-end for computer aided control engineering (CACE). The current version runs on an IRIS 2400 workstation from Silicon Graphics, Inc. The implementation is machine specific, but the design is supposed to be general and easily portable.

There are four major sections. Firstly, an overview of the system and the objectives is given. Secondly, some of the basic concepts and operations for man-machine interaction are discussed in some depth. Thirdly, the problems and requirements of window management are discussed. Fourthly, the resulting system decomposition and the actual implementation is described. There is also a summary and an appendix describing available operations in detail.

Related work at the Department of Automatic Control has been reported by Brück [1986] and Mattsson et al. [1986]. The reader is assumed to have a working knowledge of computer graphics; otherwise a good textbook is recommended, for example, Hearn and Baker [1986].

2. Overview

The man-machine interface should be well-defined and easy to comprehend. In the current version, there are 34 operations divided into eight groups: drawing primitives, text, coordinate systems, segment handling, window management, picking, rubberband shapes and menu handling; input from the user can be presented to the application in four different ways.

Most operations are rather low-level; complex operations have been included only when the graphical manipulation could be implemented without any knowledge of the meaning in the application.

Portability and implementation speed was a major concern during the design. Existing software packages have been used as much as possible; on the IRIS this includes the window manager (mex) and the IRIS Graphics Library.

Output operations

A minimum of two output devices should be supported: the workstation screen and PostScript printers for hardcopy output. Only very simple shapes such as lines, rectangles and circles are used for drawing block diagrams. Simple graphics operations are usually portable from one computer to another, but for example segment handling can be difficult to implement.

Key features are color, highlighting (visual feedback), multiple local coordinate systems which may be nested inside each other, and hierarchical segments with simple editing. Multiple overlapping windows are provided through the existing window manager.

Input operations

Two forms of input are handled. Keyboard input of text strings and commands, and a pointing device (mouse). The mouse is used for many high-level operations, such as menu selection, picking graphical objects on the screen, drawing rubberband shapes and reshaping windows. Input from stored files is also required in the future in order to repeat complex sequences. This implies that each input operation must have a textual equivalent.

3. Basic operations

This section describes a number of important operations. The problems are often implementation dependent, but the operations are fundamental for interactive graphics software. Issues related to window management are covered in the next section.

Coordinate systems

Multiple local coordinate systems may be used. Transformation matrices are pushed and popped on a stack by the user; scaling and translation may be applied to the top element of the stack only.

Transformation changes are local to a segment. The stack is automaticly pushed before the first scale or translate (unless the user has done so explicitly), and popped to its original state when the segment is left. This is done by counting the number of push/pop operations and making sure that every push is matched with a corresponding pop, even if the user has made an error.

Transformation changes are of course effective when other segments are called, i.e., at "lower" levels in the segment tree.

Segments

Hierarchical segments, in combination with hierarchical coordinate systems were considered essential for our application, in particular when building objects from smaller components, or when presenting multiple views of an object.

The current design has hierarchical segments, which are supported by PHIGS [Brown, 1985] and the IRIS Graphics Library [Silicon Graphics, 1986], but not by GKS [Hopgood et al., 1983]. Implementing hierarchical segments without support from the graphics package requires a separate data-structure, and a mapping from the user's segment identity to multiple segments stored in the graphics package. Hierarchical coordinate systems may also become more difficult to implement.

Another problem area is segment editing. Editing can be very useful, but may be difficult and very inefficient to implement without support from the graphics package; this design only supports appending of new graphics primitives to existing segments, which should be fairly easy to implement on any computer. A typical example is plotting of a trend curve, where a complete segment regeneration for each displayed point would be extremely time-consuming.

Segment attributes

It is natural to associate some attributes with a segment, for example, current color, or if segment should be highlighted. The possible scope of an attribute change can be described with three different models: Firstly, attributes can be viewed as global states. In this case, a change of color will effect any segment which is executed after the current one, including segments at "higher" levels. This model is easy to implement, but not very predicatable because the execution order of the segments is not user-specified.

Secondly, attributes may be strictly local to a segment, affecting no other segments. This model is safe and easily comprehendable, but requires a large number of possibly redundant attribute settings in each segment. This model unnecessarily restricts highlighting (see below).

Thirdly, downward propagation can be specified, i.e., attribute changes apply to the current segment and all segment called by this segment, directly or indirectly. In this case, changes affect all segments at "lower" levels in the tree, so the execution order is insignificant.

This model also has the property that the internal structure of a segment is insignificant. For example, if a segment is highlighted, it may call several "component" segments, all of which will be highlighted as well. This is not possible when attributes are strictly local. One drawback is that some redundant segments are needed. Structurally equivalent but conceptually different objects must be represented by separate segments, so that highlighting can be performed individually.

The last model has been chosen in the current design. It offers the best compromise between flexibility and safety; it is also very similar to the way the transformation stack in handled. A common problem arises if the underlying graphics library does not support a stack of attributes or transformation matrices: then it will be necessary to split a single segment (as seen by the user) into several smaller parts, and to save the attributes/matrices by some other means. Apart from execution speed, program complexity and memory requirements may be severely affected.

Picking

By picking, the user tells the application program that he/she is interested in a particular object on the screen. This is done by placing the cursor over the object and then pressing one of the mouse buttons. The problem is to make the "backward translation" from the graphical shape on the screen to an internal representation in the program.

The first problem is to determine what the user has picked. To make picking easier, a hot-zone (or picking region) of about 10×10 pixels is defined around the cursor tip; any object inside the hot-zone yields a hit. There is a risk that many objects may cause a hit, for example, near a terminal the system box, the terminal box and the connection may all cause hits; it is not clear what the user really intended. This design specifies that every object in the hot-zone causes a hit; it is left to the application to resolve any conflicts.

It is common to associate picking with segments, i.e., a hit returns a segment identity. The second problem arises when two conceptually different objects are represented using a common set of segments because they are graphically similar. If only a segment identity was returned, the different objects could not be distinguished.

In order to solve this problem, the path to an object must be traced, i.e., every enclosing object must be listed in addition to the segment that caused the hit. If there is more than one occurrence of a segment in the enclosing object, every occurrence must be counted. This process can of course be automated, but the mapping back to the internal data structures in the program is not much easier.

An alternative approach, which has been chosen here, is to put the result of a pick operation under direct program control. The application program can push and pop so called pick markers on a special-purpose stack. The push/pop operations can be put anywhere in a segment, interspersed with ordinary drawing primitives. When a hit occurs, the complete stack is returned to the program for interpretation. The only action performed automatically is to empty the stack before picking starts.

The push marker stack is very flexible and can easily emulate the two other strategies: single-segment hit and segment-path hit. More powerful encodings of the stack contents are also possible. Regrettably, it is not yet clear if the stack can be readily implemented on other workstations than the IRIS, or using graphics packages like PHIGS or GKS. A complete redesign may be called for, with subsequent changes in the application program.

Rubberband shapes

The rubberband line is a convenient but resource consuming way of interacting with the user. A rubberband line has one fixed end-point, while the other end-point is tied to the cursor. The line is redrawn as quickly as possible when the user moves the cursor, giving almost instant feedback. A common extension is to be able to draw a rectangle between two end-points, as well as a line.

In the current design, the rubberband line has been generalized to a polyline, i.e., a number of points connected with straight line segments. The following shapes can be formed:

Variable length polyline. The first and intermediate points are marked by pressing the middle mouse button; the last point is marked by pressing the right mouse button.

Straight line. The application program requests input of a polyline with only one line segment. Input of the line is automaticly finished when two points have been marked.

Rectangle. Similar to the straight line, but rather than drawing a line, a rectangle is formed between the two end-points. The cursor glyph can be changed to emphasize the rectangle. A polyrectangle is theoretically possible, but would probably look funny.

Single point. A degenerate polyline with only one point. Used for example to mark the position of an input or an output.

Window indentity. Input of a rubberband shape also carries a window identity that tells which coordinate system the points refer to. Requesting a polyline with zero points returns the window identity of the currently attached window, without any user action.

The rubberband shape is controlled by three parameters. Firstly, the maximum number of points; for a "variable length" polyline, any large number will do. Secondly, a flag that indicates that a polyline can be finished prematurely; otherwise, all points specified above must be marked. Thirdly, a graphical shape, i.e., whether a line or a rectangle should be drawn between two points.

Menus

Menu selection is the principal way to give commands to the system. The IRIS Graphics Library has a simple but flexible menu handler; the following basic interaction ideas have been used:

- There is always one main menu which pops up when the right mouse button is pressed.
- Menus pop up at the current cursor position. The first alternative is under the cursor.
- Menus may roll over to either side, displaying any number of submenus.

Features that for simplicity reasons have been left out are icons (menu alternatives represented by pictures, not by text strings) and forms (input of text in the menus). The menus are programmable by the application program, and any text string can be returned for a menu selection, but it is not possible to parameterize selections.

Text editor

Some sort of text editing facility is required, for example, to edit equations. This can be done in three ways: Firstly, an existing editor can be executed separately (taken off the shelf, so to speak). The main advantages are the simple implementation, and that the user is allowed to use his/her favorite editor. The main disadvantage is the lack of integration; start-up will be rather slow, and the user must manually read and write files containing equations.

Secondly, a well-known standard editor (such as Emacs) can be adapted to our particular needs. For example, Emacs can be programmed with a special "equation-mode," or to understand program-generated error messages. Another advantage is that start-up of the editor can be done automaticly, and relevant files transferred without the user's help. The disadvantages are the increase in implementation effort, and that the user is tied to a single (albeit programmable) editor.

The third approach is to write a special-purpose editor which can be completely integrated with the CACE package. In addition, start-up will be fast, and more important, graphics can be used to advantage. On the other hand, the user will be confined to yet another, rather restricted, text editor, and much work must be devoted to ordinary program development. An alternative is to turn the problem outside-in, and implement the CACE system in Emacs.

The current opinion advocates approach number two, the adapted standard editor. Some sort of integration is preferable, but the required effort should be kept to a minimum. It should be possible to begin with a minimum of integration, e.g., automatic start-up and equation-text transfer.

4. Window manager interaction

Window management is one of the least developed and standardized areas of man-machine interaction. A good discussion is presented by Hopgood et al. [1986].

The window manager on the IRIS, called mex (multiple exposure), is quite simple and easy to use, but does not provide any high-level operations except menus. All input from keyboard and mouse goes to mex, and is then redistributed to the process that owns the currently attached window (the user attaches himself to the window he/she is interested in). Moving, reshaping and selecting windows is done by the user and mex, with only a limited cooperation from the affected processes.

There is always at least one text window on the screen, which is associated with "standard input," "standard output" and "error output" (i.e., used by read and write in Pascal). In addition, processes that use graphics normally have one or more graphics windows. Regrettably, it is not possible to print directly to a graphics window. Each process can have multiple graphics windows, but each graphics window can only be owned by one process; the text

window is normally shared by all processes. A process can only use windows it owns.

Mex is different from many other window managers because it cannot redraw a window by itself, for example, when a window has been moved; instead, mex will request every process to redraw those windows that have been affected. This strategy has probably been adopted because the IRIS is often used to display very complex images, which may not be efficiently stored in an application-independent format.

The input queue

The IRIS Graphics Library manages an input queue where events such as mouse movements and key presses are recorded. A typical program enables specific events and then spends it time reading the input queue, performing some action on every event.

The window manager uses this queue too; when mex detects that a window must be redrawn, a REDRAW token is put on the input queue, together with an indication of which window must be redrawn. The program must then redraw its window completely, taking into account possible changes in window size, aspect ratio, etc. The INPUTCHANGE token informs that that the user has attached or detached a window.

This strategy for screen regeneration has two implications. Firstly, the program must run in an endless loop, reading tokens off the input queue and redrawing windows on request. In order to get acceptable interaction with the user, this must be done quite fast (no absolute measures can be given yet). It should be noted that the windows must be redrawn in a certain order because of window overlap, so several processes may be affected.

Secondly, the REDRAW token is an asynchronous event over which the program has no control. This means that the program must be prepared to redraw any of its windows at any time, i.e., there must always exist a valid representation of the window contents. For this reason, there is always a segment associated with each window. This is the root in a tree of segments which defines the window contents. The program may of course ignore the REDRAW token or delay any actions, but the effect on user interaction is hardly acceptable.

Double buffering

A simple way to get subjectively faster graphics is to use two buffers; one is displayed and the other updated. Because the buffers can be instantly swapped, the user does not have to watch the image being drawn. In practice, two buffers are slightly mode complicated to handle than one, in particular when doing incremental updates.

Mex can handle programs that use both single buffered mode and double buffered mode. All processes share one display memory, so some sort of synchronization is needed. The User's Guide says:

> *"Each double buffered program is blocked until all other double buffered programs are ready to swap. Then all programs running in double buffered mode swap at the same time."*

For double buffered programs, this means that every program must wait for the slowest program to finish drawing. There is no possibility to do anything useful while the program hangs, and it is not possible to detect that the program will hang. Single buffered programs are not directly affected; they

are requested to redraw for each swap, so the total amount of redrawing may increase considerably.

5. Realization

This section covers three topics: the decomposition of the system into functionally different parts, the implementation of the front-end, and a discussion of future developments and the use of PHIGS.

System decomposition

In the current design, the complete system has been decomposed into three functionally different parts:

- An **application** written in Common Lisp which understands system representation and operates on systems.
- A **command decoder** which reads user input, either directly or from a command file; it also creates command files and log files. A future expert system can be associated with the command decoder.
- A **front-end** which handles physical devices and interaction with window manager.

The implementation uses two Unix processes: one process that runs the application and the command decoder, and one process for the graphical front-end. The front-end is written in C, the rest in Common Lisp.

The main reasons for using two separate processes are the requirements imposed by the window manager. Writing a program that gracefully interacts with mex (e.g., frequently reads the input queue) requires a program structure which makes non-graphical parts more difficult to write. Although double buffering is not yet used, future use of double buffering would make this problem even more difficult. There are also some operations, such as rubberband shapes, that require maximum performance; this is easier to achieve in C. In addition, some window manager commands have no Common Lisp bindings.

The last reason is portability. The current front-end is quite IRIS specific with close ties to the IRIS Graphics Library; a separate program (process) is easier to rewrite and port to other computers. Decomposition into separate processes requires a well-defined protocol. This puts am emphasis on specification and documentation of the front-end.

Process communication

Communication between the application and the front-end is done with Unix messages. There is one message for every graphical operation (output), and one message for every form of input. Messages are fully asynchronous, and the application cannot assume that a certain type of input will be made by the user.

Output messages are binary, but input messages are sent as textlines (any numbers are converted to textual form). Binary messages are more efficient by a factor of 4–5, but this doesn't matter for input. Input messages are constructed to look like lists in Lisp; consequently, text messages are normally evaluated by the Lisp system, so the full power of Lisp is still directly available. Binary input messages would have required the construction of an internal Lisp object, which would have been difficult in C.

Transmitting a message requires two system calls and copying, but this is not unreasonable; Unix messages are as efficient as sockets or pipes. On the other hand, using two processes causes some competition for CPU resources and additional context switches.

Front-end implementation

The front-end runs in an endless loop, responding to stimuli from two sources: messages from the application and tokens on the input queue from the user (or mex). It is not possible to wait for input from two sources simultaneously. The first version of the front-end used busy-wait, i.e., continuously polled both sources. Regrettably, this consumed about 20% of the CPU resources when the front-end was "idle." The current version normally hangs on the input queue; a timer puts tokens on the queue ten times per second, causing the front-end to poll the message queue. It uses approximately 3% of the CPU when idle, and 30–35% when driven by a fast Lisp program.

All operations are regarded as atomic, which is almost true. All operations are allowed to finish before any other input is honored. This means, for example, that it is impossible to move or reshape a window when picking or rubberband drawing has been requested.

Another problem arises when the program is constructing a segment, i.e., storing graphics primitives in an internal datastructure. If a REDRAW token forces a redraw of a window using the segment under construction, the segment must be saved temporarily and whatever there is used for drawing. What will really be drawn, and if the image is consistent, is not clear. The current implementation has a serious bug though. The REDRAW token will not redraw the window, but the operations to redraw a window will be inserted into the currently open segment; disaster is inevitable. The combination of segment editing and window changes is luckily not so common.

An exception handling package for C is used to handle bad messages, to start polling the messages queue after a time-out, and to abort execution when the BREAK key has been pressed. It is powerful and easy to use.

Future developments

The front-end is in its present state not fully robust. Illegal operations (e.g., bad message parameters), and perhaps some legal operations, may cause the front-end to crash. Diagnostics are bad; illegal messages are ignored, unless tracing is turned on.

The most serious problem is the misuse of the REDRAW token when a segment is open for editing (see above). There are operations which are not meaningful when editing segments (e.g., requesting a rubberband shape), and operations which are meaningful only when editing a segment (e.g., closing a segment). Every operation should be categorized and tests made appropriately. It is possible to draw directly to the screen without using a segment, but the application cannot control which window will be used, or what coordinate system applies. This feature would be valuable for testing purposes.

Windows are at present shaped by the user. In some cases the application could preset the size, and perhaps even the position, of a newly created window. In many window management systems it is possible to shrink a window to an icon; when the window is expanded again, it assumes its original size and position.

Two additional types of input are immediately useful: forms and dragging.

A form can be regarded as a more general menu facility; the user can type text into one or more predefined slots. The completed form is then sent to the application.

Dragging means that you have an existing shape and want to move it around the screen, for example, to position an icon. The application program could define a segment, which is tied to the cursor. After the final position has been marked, the new coordinates are returned. It would also be possible to change an existing shape. One point of the shape is tied to the cursor, and the shape changes as the user moves the mouse. Final position of the point is returned to the application. Two-button stretching in Hibliz [Mattsson et al., 1986] is a combination where the user can move a shape or change the position of two points individually.

Some features of the front-end which today are welded-in should be controlled from the application. Examples are: tracing of messages, window background color, use of a grid to make rubberband shapes easier to draw, and color definitions. An initialization file could provide default parameters.

Highlevel plotting routines for drawing nicely scaled axes in Simnon have been converted to C. Axes generation should be handled by the application program, not the front-end.

Implementation using PHIGS

The proposed graphics standard PHIGS (Programmers Hierarchical Interactive Graphics Standard) is specificly aimed at high-speed interactive modelling of 3D objects [SIS, 1985]. One of the key concepts is the centralized segment storage; multiple segment trees can be maintained simultaneously, and individually tied to views, the windows in PHIGS.

Segments (called structures in PHIGS) have a number of attributes, such as the ability to be picked, highlighted or invisible. With respect to picking, any segment can be uniquely identified, including all enclosing segments. Picking in PHIGS is not as flexible as the scheme outlined in section 2, but fully adequate for our purposes.

The graphical front-end can probably be easily implemented using PHIGS, but no practical tests have been conducted. It would also be interesting to see if windowing can be implemented using multiple views in PHIGS, and if this would remove some of the restrictions imposed by the window manager. There exists today a single implementation of PHIGS for the IRIS workstation, and more may become available in the near future.

6. Summary

Key features

The man-machine interface should

- Be well-defined.
- Handle input from keyboard, pointing device (mouse) and stored files.
- Handle multiple graphical output devices; minimum is the IRIS screen, Postscript devices, and some sort of logging facility.
- Use existing software packages as much as possible, e.g., the window manager (mex) and the graphics library on the IRIS.

Basic output operations include

- Draw simple shapes like lines, rectangles and circles. Fill areas using multiple colors.
- Multiple local (hierarchical) coordinate systems.
- Highlighting, or some other visual feedback.
- Multiple overlapping windows.
- Segment handling in order to improve efficiency and aid structuring of graphical information.

Basic input operations include

- Text input.
- Menu selection, with equivalent textual command forms.
- Picking of graphical objects.
- Rubberband shapes, e.g., lines, rectangles and polylines.
- Reshaping and moving windows.

Some operations are quite powerful, for example, there are hierarchical coordinate systems and hierarchical segments. Segment editing is limited to appending graphics primitives to an existing segment, and picking conflicts must be resolved by the application. The rubberband concept has been generalized; parameters controlling a rubberband polyline provide points, lines, rectangles and polylines.

System decomposition

The current design suggests a system decomposed into three separate parts:

- An **application** written in Common Lisp which understands system representation, and operates on systems.
- A **command decoder** which reads user input, either directly or from a command file. Command files are generated by the command decoder. A future expert system interface will be associated with the command decoder.
- A **front-end** which handles physical devices and interaction with the window manager.

The following issues influenced the design

- The window manager imposes certain requirements on all processes using windows. Any process owning a window must be able to redraw its windows on demand, for example, when a window is reshaped.
- The window manager puts a special *REDRAW* token on the same input queue which is used for mouse input. This means that the program must run in an endless loop, reading the input queue as quickly as possible.
- In doublebuffered mode, all processes owning windows are blocked until every process has redrawn its windows.
- Future extensions in the area of expert system interfaces tie naturally into the command decoder, i.e., after the physical input, but before the application.
- A quick implementation of the man-machine interface has lead to a program which has close ties with the IRIS Graphics Library. A separate program (process) is easier to redesign and port to other computers.

- Decomposition into separate processes requires well-defined protocols using some sort of message passing. This puts an emphasis on specification and documentation of the man-machine interface.

Portability can be a problem, in particular with the chosen strategy for picking. Other problem areas are segment editing, segment attributes and hierarchical coordinate systems. It is probably quite easy to implement the front-end using PHIGS, but quite difficult using GKS.

7. Acknowledgements

The author is grateful for the feedback and suggestions from Mats Andersson, who used the front-end during its most unstable time. Dr. Sven Erik Mattsson provided helpful comments and suggestions, as usual.

This work was supported by The National Swedish Board of Technichal Development (STU), as part of the Computer Aided Control Engineering project conducted at the Department of Automatic Control, Lund.

8. References

BROWN, M. D. (1985): *Understanding PHIGS*, Template, 9645 Scranton Road, San Diego, CA 92191, USA.

BRÜCK, D. M. (1986): "Implementation of Graphics for HIBLIZ," CODEN: LUTFD2/TFRT-7328, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

HEARN, D. and P. BAKER (1986): *Computer Graphics*, Prentice-Hall International, ISBN 0-13-165598-9.

HOPGOOD, F. R. A., D. A. DUCE, J. R. GALLOP and D. C. SUTCLIFFE (1983): *Introduction to the Graphical Kernel Standard (GKS)*, Academic Press.

HOPGOOD, F. R. A., D. A. DUCE, E. V. C. FIELDING, K. ROBINSON and A. S. WILLIAMS (Eds.) (1986): *Methodology of Window Management*, Proceedings of an Alvey Workshop at Cosener's House, Abingdon, UK, April 1985, Springer-Verlag.

MATTSSON, S. E., H. ELMQVIST and D. M. BRÜCK (1986): "New Forms of Man-Machine Interaction," CODEN: LUTFD2/TFRT-3181, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

SILICON GRAPHICS (1986): *IRIS User's Guide*, Version 2.1, Update 2.4, Silicon Graphics Inc., 2021 Stierlin Road, Mountain View, CA 94043, USA.

SIS (1985): "Datorgrafi—PHIGS, Programmers Hierarchical Interactive Graphics Standard," Technical report no. 306, SIS—Standardiseringskommisionen i Sverige.

Appendix – Front-End Operations

1. Low-level output interface

This section describes the output operations of the interface between the front-end and the command decoder. These operations are issued by the application and executed by the front-end.

Because the window manager (mex) may ask the front-end to redraw a window at any time, everything on the screen must be stored in segments. The general procedure for creating graphics in as follows: Firstly, a set of basic segments is created; these contain shapes that need no changes in the future. Secondly, a set of higher-level segments is created; these also call other segments to produce more complicated shapes. Lastly, one or more windows are defined, each of which call one segment to draw the image in the window.

Drawing primitives

Used to draw graphical objects (shapes) and save or restore local attributes. These operations are 2D only.

```
move(x, y : coord)
```

Sets current position to (x, y).

```
draw(x, y : coord)
```

Draws a line from current position to (x, y), which then becomes the current position.

```
rectangle(x1, y1, x2, y2 : coord)
fillrectangle(x1, y1, x2, y2 : coord)
```

Draws a rectangle or a filled rectangle from $(x1, y1)$ to $(x2, y2)$. Current position becomes $(x1, y1)$.

```
circle(x, y : coord; radius : real)
fillcircle(x, y : coord; radius : real)
```

Draws a circle or a filled circle with center point in (x, y) and specified radius. Current position becomes (x, y).

```
setcolor(col : integer)
```

Shapes will be drawn with the specified color from the color map. The color map is defined by a special file. The default color, when no color has been specified, is blue.

```
setlinewidth(w : integer)
```

Shapes will be drawn with w pixels wide lines. Minimum is one pixel.

Text

Textual output is restricted to simple strings. Formatting of numerical data must be done by the application. Raster or graphical fonts may be supported.

```
drawstring(height : integer; s : string)
```

Draws a string starting at the current position. The height is specified in local user coordinates.

```
requeststring
```

Asks the user to type a string, which is returned to the application. See *Text* in Section 2.

Coordinate systems

There may be multiple, hierarchically organized coordinate systems. The coordinate system is represented by a transformation matrix, which is manipulated by scaling and translation.

```
scale(x, y : real)
```

Changes horizontal and vertical scale by multiplying the current transformation matrix. For $c \in \{x, y\}$ we have

$$\begin{aligned} |c| > 1 &\quad \text{expands objects,} \\ 0 < |c| < 1 &\quad \text{shrinks objects,} \\ c < 0 &\quad \text{mirrors objects.} \end{aligned}$$

```
translate(x, y : coord)
```

Moves the origin of the coordinate system relative its current position in the enclosing coordinate system.

```
pushmatrix
popmatrix
```

Saves and restores the current transformation matrix using a stack.

Segment handling

A segment may contain drawing primitives, operations to manipulate the transformation matrix, calls to other segments, and operations to set the pick marker. These are collectively called *graphics commands.*

It is possible to *call* a segment, almost like subroutines are called in a programming language. When a segment is called, the current state of the caller is saved, and later restored on return. In particular, the transformation matrix and attributes, such as color and linewidth, are saved.

```
open(name : segmentid)
```

Opens a segment which will be ready to accept graphics commands. If the segment did not exist, a new segment is created; otherwise, the graphics commands will be appended to the existing segment. Only one segment may be open at a time.

```
close
```

Closes the currently open segment. No action is taken if no segment is open.

```
delete(name : segmentid)
```

Deletes the specified segment. No action is taken if the segment does not exist. It is possible to delete the open segment.

```
call(name : segmentid)
```

The specified segment will be invoked when the call is executed. Saves and restores transformation matrix and attributes. Calling an undefined segment (either not yet defined, or deleted) is a no-op.

```
highlight(name : segmentid; on : boolean)
```

The specified segment, and all segments that are called by this segment, will be highlighted.

Window management

Windows are rectangular areas on the screen. There is always a single segment associated with each window; this segment is the root of a tree of segments, which will draw an image in the window. The tree may be used at any time to redraw the contents of a window, for example, when the user has resized or moved a window.

When the front-end starts, the user is asked to shape an initial window; this window has identity 0 (zero). The application cannot use more than ten windows simultaneously.

```
create(win : windowid)
```

The user is asked to shape a window on the screen. The window is initially empty.

```
title(win: windowid; t : string);
```

Assigns a title to the specified window. An empty string ("") removes any existing title from the window.

```
bind(win : windowid; seg : segmentid)
```

Binds a segment to a window in order to draw an image in the window. The segment is called every time the window must be redrawn.

```
limits(win : windowid; x1, y1, x2, y2 : coord)
```

Defines the window limits, i.e., a local coordinate system from $(x1, y1)$ to $(x2, y2)$ which will fill the window. The initial values when creating a window are $(0, 0)$ to $(1, 1)$.

```
erase(win : windowid)
```

Removes the specified window from the screen. The associated segment is not destroyed.

```
redrawwindow(win : windowid)
redrawall
```

Forces redraw of a window/all windows. Updating a segment does not update the window automaticly.

Picking

Picking is a form of input, but the application controls the result of picking a certain object, and can also ask the user to pick an object.

The front-end maintains a stack of so-called *pick markers*. Interspersed with other graphics commands, the application pushes and pops markers on the stack. Normally the markers on the stack are not used for drawing, but when a pick-hit occurs, the contents of the stack is returned to the application. The application may then interpret the contents of the stack in any way.

```
pushmarker(m : pickmarker)
popmarker
```

Saves and removes a user specified marker on a stack. The contents of the stack is returned when a pick-hit occurs.

```
requestpick
```

Asks the user to pick an object on the screen. See *Picking* in Section 2.

Rubberband shapes

Rubberbanding is a form of input, but the application has the possibility to prompt the user for a shape. See *Rubberband shapes* in Section 2.

```
requestline
requestrectangle
```

Asks the user to define a line or rectangle using rubberbands.

```
requestshape(n : integer)
```

Asks the user to define a polyline with at most n points, i.e. $n-1$ archs. Certain values of n have special meaning:

$n = 0$ Returns the window identity, but no points.

$n = 1$ Returns a point.

$n = 2$ Returns a line (or possibly a point).

$n > 2$ Returns a polyline.

Defining menus

The application can define menus which are then handled locally in the front-end. There may be multiple menus, each with a variable number of choices. A menu choice may also "roll over" to a submenu. Menus are numbered freely by the application, with the following exception:

Menu 0 Reserved, meaning no menu at all. Normally used in `addtomenu` to indicate that roll-over menu option is not desired.

The application associates a string with every menu choice. This string is sent to the application when the user selects something. If the user pops a menu but does not select anything, the string *NULL* is sent to the application.

`newmenu(m : menuid; heading : string)`

Starts the definition of a new menu. Any existing menu with the same identity will be destroyed.

`addtomenu(m : menuid; ch, answer : string; rover : menuid)`

Adds a choice to the specified menu. If this choice is selected, the string *answer* will be sent to the application. If *rover* is non-zero, the menu choice may roll-over to a previously defined menu.

`requestmenu(m : menuid)`

Presents the specified menu to the user and awaits selection.

`mainmenu(m : menuid)`

Makes *m* the main menu, i.e., the menu the user gets when he presses the right mouse button.

2. Low-level input interface

This section describes the input operations of the interface between the front-end and the command decoder. These operations are issued by the front-end and interpreted by the command decoder; most operations are forwarded to the application without changes.

Many input operations are initiated by the application, i.e., the application sends a request, forcing the user to do some input operation. In the general case however, the user is free to perform any operation, so the application must be able to accept (or reject!) any input at any time. A summary of the proposed user interface:

- Typed input is sent as strings to the application.
- Pressing the *right mouse button* invokes the main menu of the application. It is sometimes used to draw rubberband shapes.
- Pressing the *middle button* picks an object on the screen. It is also used to draw rubberband shapes.
- The *left button* is used to select a window.
- Pressing *NO SCRL* stops output from the front-end.
- Pressing *SET UP* directs input to the window manager (mex).

Text

Keyboard input is sent line-by-line to the application. Exceptions are some keys which will send a Unix signal to the application.

`text(s : string)`

Sends the string typed by the user (excluding the final new-line character) to the application. Double quote (") is preceded by a backslash (\). Every character is echoed to the *console window* as it is typed. Pressing *DELETE* erases the last character.

Rubberband lines and rectangles

Rubberband shapes are normally requested by the application. The user positions the cursor at the starting point and presses the middle mouse button. He/she then moves the cursor to the ending point and presses the middle button a second time.

The returned coordinates are window coordinates, i.e., expressed in terms of the window limits. The coordinates may specify a point outside the window. Because lines and rectangles are specializations of the more general polyline, a count of the number of coordinate pairs is sent to the application (the count is of course 2).

```
rubberline(w : windowid; 2; x1, y1, x2, y2 : coord)
```

Sends the window identity and the end-points of a line drawn by the user.

```
rubberrectangle(w : windowid; 2; x1, y1, x2, y2 : coord)
```

Sends the window identity and two opposing corners of a rectangle drawn by the user.

Rubberband polyline

This is a generalization of the rubberband line. The user defines a number of points by pressing the middle mouse, up to a maximum specified in `requestshape`. The last point is defined by pressing the right mouse button.

```
rubberline(w : windowid; n : integer; a : array of coord)
```

Sends the window identity and an array of coordinate pairs (x_i, y_i) for a polyline drawn by the user. The count n is number of pairs. The coordinate pairs are stored in the order they were defined by the user.

Picking

To pick an object, the user points at a shape on the screen and presses the middle mouse button. The front-end will redraw every window and simultaneously maintain a stack of pick markers (see *Picking* in Section 1). When a pick-hit occurs, the contents of the stack is sent to the application.

```
pick(w : windowid; n : integer; hits : array of pickhit)
```

Sends the window identity, the number of pick hits and an array of pick-hits to the application. Each pick-hit is represented by a count m and the stack of m pick markers at the time of the hit.

If no pick-hit occurs, `pick(0, 0)` is sent to the application. This is also the case if the user picks outside the windows belonging to the front-end, in which case the window identity is meaningless.

Menu selection

The main menu is presented when the user presses the right mouse button. He can then select one of the choices, or roll over to another menu. When the user releases the button, the string associated with the choice (see `addtomenu`) is sent to the application. If the user does not select any of the choices (selects outside the menu), the string `*NULL*` is sent to the application.

`menu(s : string)`

Sends the string associated with a menu choice to the application.

Unix signals

A Unix signal is an asynchronous interrupt, which may be caught by the receiving process. Two signals have been bound to keys:

ESC	Sends *SIGINT* to the application, which will put the user at the Common Lisp top-level. The application can later be resumed.
BREAK	Sends *SIGKILL* to the application, which will be killed.

These signals can probably be caught by some mechanism in the Common Lisp system, but God knows what...

3. Examples

EXAMPLE 1 — Simple segment
This is a very simple segment which defines a diagonal line.

```
open(1)
move(1, 1)
draw(9, 9)
close
```

EXAMPLE 2 — A more complex segment
This segment will draw a box and then call the previous segment to draw a diagonal line inside the box.

```
open(2)
rectangle(0, 0, 10, 10)
call(1)
close
```

EXAMPLE 3 — System segment
This is a segment representing a system. It consists of a surrounding box and two of the boxes above, representing sub-systems.

```
open(3)
rectangle(0, 0, 10, 10)

pushmatrix
translate(2, 4)
scale(0.2, 0.2)
call(2)
popmatrix

pushmatrix
translate(6, 4)
scale(0.2, 0.2)
call(2)
popmatrix
close
```

EXAMPLE 4 — Simple window

Here we create a simple window in order to display a system.

```
create(100)
limits(100, -1, -1, 11, 11)
bind(100, 3)
redrawwindow(100)
```

EXAMPLE 5 — Zooming closer

By changing the window limits, we zoom in on the lower-left corner of the system.

```
limits(100, 0, 0, 5, 5)
redrawwindow(100)
```

EXAMPLE 6 — Information zooming

Here we open a new window and zoom in one of the sub-systems. We do not change the representation though.

```
create(101)
limits(101, 1, 3, 5, 7)
bind(101, 3)
redrawwindow(101)
```

EXAMPLE 7 — Picking an object

In order to illustrate picking, we must adopt some conventions: Firstly, every segment will begin with a push of a marker indicating the *type* of the segment. Secondly, to distinguish between multiple calls of a segment, a *sequence number* will be pushed in front of every call.

First we create a sub-system box and a system with two sub-systems. This is very similar to Example 3.

```
open(1)
pushmarker(SS)
rectangle(0, 0, 10, 10)
popmarker
close

open(2)
pushmarker(S)
rectangle(0, 0, 10, 10)

pushmatrix
translate(2, 4)
scale(0.2, 0.2)
pushmarker(C1)
call(1)
popmarker
popmatrix

pushmatrix
translate(6, 4)
scale(0.2, 0.2)
pushmarker(C2)
call(1)
popmarker
popmatrix
```

```
popmarker
close
```

The application can now send a `requestpick` to the front-end. If the user doesn't pick any object at all, no (zero) pick markers are returned. If the user picks the system above, but not one of the sub-systems, one pick marker is returned:

```
1  S
```

If the user picks the left sub-system, three pick markers are returned, indicating a system, the first call to another segment, and the sub-system:

```
3  S  C1  SS
```

EXAMPLE 8 — Another picking scheme

A segment will return the same pick markers every time it is called. It is not possible to tell which call (instance) of a certain segment caused a pick-hit without more information. In the example above, we could separate the two calls of the sub-system becaused we traced every call in the enclosing system.

Not every segment must use pick markers. It is convenient to use some segments simply as a library of geometrical shapes, glued together with segments that push and pop pick markers. We can probably afford to create a unique segment for every object we want to be able to pick. In this case, the pick markers can be encoded to uniquely identify objects in the application.

```
open(1)
pushmarker(SS1)
rectangle(0, 0, 10, 10)
popmarker
close

open(2)
pushmarker(SS2)
rectangle(0, 0, 10, 10)
popmarker
close

open(3)
pushmarker(S)
rectangle(0, 0, 10, 10)

pushmatrix
translate(2, 4)
scale(0.2, 0.2)
call(1)
popmatrix

pushmatrix
translate(6, 4)
scale(0.2, 0.2)
call(2)
popmatrix

popmarker
close
```

If the user picks the left sub-system, only two pick markers are returned, one for the system and one for the sub-system.

```
2  S  SS1
```