1988

The meta-device: an object based, non-retained graphical layer

Dario Giuse
Carnegie Mellon University

Lynn Baumeister

Follow this and additional works at: http://repository.cmu.edu/compsci
NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.
The Meta-Device: An Object Based, Non-Retained Graphical Layer

Dario Giuse
Lynn Baumeister

April 1988
CMU-CS-88-136(2)

Abstract
The Meta-Device is a graphical interface tool developed as part of the Dante project at Carnegie Mellon University. The Meta-Device provides device- and window system independent access to graphical output operations in a workstation environment. The Meta-Device uses an object-oriented model which supports easily extensible handling of input events for application programs written in Common Lisp. The system represents the lowest graphical level in the Dante system, and therefore it does not support retained objects; rather, it provides efficient access to the primitive graphical operations of the host window system.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.
# Table of Contents

1. Introduction
   2.1 Structure of the System
   2.2 Functionality
   2.3 The Object Structure
   2.4 Environment
3. Programming Interface to the Meta-Device
   3.1 Common Parameter Values
      3.1.1 Coordinate Mappings
      3.1.2 Drawing Modes
      3.1.3 Line Styles
      3.1.4 Orientation
   3.2 Devices
      3.2.1 Creation and Destruction
      3.2.2 Status Information
   3.3 Viewports
      3.3.1 Creation and Destruction
      3.3.2 Status Information
      3.3.3 Modifications
      3.3.4 Graphical Operations
      3.3.5 Mapping Conversions
         3.3.5.1 Converting Coordinates
         3.3.5.2 Converting Dimensions
   3.4 Text and Fonts
      3.4.1 Creation and Destruction
      3.4.2 Status Information
      3.4.3 Drawing Text
   3.5 Event Handling
      3.5.1 Events
      3.5.2 Defining Event Handling Methods
4. An Example
5. Summary
6. Acknowledgments
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>The Device and Viewport class hierarchy in the Meta-Device</td>
<td>3</td>
</tr>
<tr>
<td>3-1</td>
<td>An example of different coordinate systems</td>
<td>5</td>
</tr>
<tr>
<td>4-1</td>
<td>The two viewports in our example</td>
<td>17</td>
</tr>
<tr>
<td>4-2</td>
<td>Viewport ANOTHER-VP after several mouse clicks</td>
<td>18</td>
</tr>
</tbody>
</table>
1. Introduction

The Dante project [Giuse 86] [Kuokka and Giuse 88] is an investigation into various man-machine interface issues. An important component of the system is the lowest-level graphical interface tool, which is referred to as the Meta-Device and is described in this document.

The Meta-Device, which was originally implemented by Roy F. Busdiecker, is implemented in Common Lisp [Steele 84] and provides efficient access to the low-level graphical operations and input handling of the host hardware and window system. The Meta-Device uses the object-based paradigm extensively as a means of guaranteeing independence of the application program from both the underlying hardware and the particular window system.

Being the first layer of the graphical system, the Meta-Device does not provide retained graphical objects: no storage is allocated for graphical entities, and no display list is kept. Other layers in the system provide this functionality, thus letting the Meta-Device specifically concentrate on the handling of graphical output operations and input events.

The Meta-Device thus fills two important roles: The first is that of a building block for the rest of the Dante tools, which use it to provide low-level graphical input and output. The second role is that of an efficient graphical layer for application programs which cannot afford the overhead of going through a retained-object layer. Such applications may contain their own handling of retained objects, or they may use a mixture of retained objects and direct calls to the Meta-Device for operations which must be executed very efficiently.

The first section of this document presents the general principles behind the design of the Meta-Device and its relationship with similar systems. The central portion describes in detail the functional interface to the system and gives a complete specification of all the functions that comprise the interface. Finally, the last section presents an example of actual usage of the system.


This section briefly describes the structure of the Meta-Device and its functionality. After indicating the class of application programs for which the system is best suited, we describe the object system on which the Meta-Device is based and its paradigm for handling graphical output and input events.

2.1 Structure of the System

The Meta-Device is a Common Lisp library package. It does not depend on any other portion of the Dante system; it does, however, require the presence of CLOS, the Common Lisp Object System [Bobrow et al. 85].

The Meta-Device consists of two separate parts: a device-independent portion and a device-dependent portion. The device-independent portion provides the interface to the application program, the declaration of the main classes used by the system, and a variety of internal functions. This portion is totally independent of the particular host hardware and window system.

The device-dependent portion specializes the system for a particular host window system and hardware. This portion, also known as a device driver, takes care of encapsulating differences among different window systems, for example, thus providing device independence. Porting the Meta-Device to a different system requires the writing of a new device driver, whereas the generic portion would remain unchanged.
The device-dependent portion can also be specialized to provide an interface to other than a window system; for instance, it could generate output for a particular printer or file format. Please note that the only fully supported device driver is currently the one for the X window system [Scheifler and Gettys 86], version 10.4; we are considering an extension to X version 11.

The interface from the device driver to the underlying window system is of course totally dependent on the particular conventions of the window system. An application program, however, need never be concerned about the details of this interface, which is totally hidden inside the Meta-Device.

### 2.2 Functionality

The Meta-Device provides a device- and window-system independent graphical interface for application programs. It is not by itself a window system, although it does provide a standardized interface to most window system operations.

The functionality of the Meta-Device falls into four categories:

- Window management.
- Graphical output operations.
- Input handling.
- Redisplay/reshape handling.

**Window management** includes operations like creating and destroying viewports, modifying the shape and position of existing viewports, and obtaining information about the current status of the system. It also includes operations on a device, such as creating a new device or finding out the status of all the viewports that are currently active. The Meta-Device translates all requests in this category into whatever commands are required by the underlying window system.

**Graphical output operations** include the drawing of graphical entities in a viewport of a given device, clearing the contents of a viewport, outputting text in different fonts, etc. This category is probably the most straightforward of the four. Since the Meta-Device does not support retained graphical objects, all output operations are performed only once, at the time a request is issued. The Meta-Device itself does not keep any track of what graphical output operations were performed in a viewport.

**Input handling** enables an application program to define handlers for all input from the user. Such input, typically from a keyboard or a mouse, consists of a sequence of input events, where each input event is thought of as an elementary action. Examples of input events are a single keystroke from the keyboard, or a movement of the mouse from its previous position.

**Redisplay/reshape handling** enables an application program to be notified whenever one of the viewports it controls is modified by the user through window system commands. The Meta-Device relays this information to the application program as an asynchronous event; the application program is then responsible for handling the event in the appropriate way.

### 2.3 The Object Structure

The most important concepts in the Meta-Device are the device and the viewport. These two concepts are implemented by hierarchies of object classes and subclasses, following the object-oriented paradigm. Class hierarchies allow considerable sharing of functionality among different instances of a class and among different levels of a hierarchy. When defining the output handlers for a new device, for instance, it is often possible to inherit much of the functionality from a higher level in the device hierarchy. This simplifies the incorporation of new devices into the system while reducing the total size of the system via sharing of information.
Figure 2-1 illustrates the main class hierarchy, which includes devices and viewports. The top-level of the Meta-Device class hierarchy is the *device* class. A device object, i.e., an instance of the *device* class, represents a connection from the Meta-Device to a particular device (such as a display screen) under a particular window system. An example is indicated in Figure 2-1 as DEVICE-1.

The creation of a *device* object initiates any hand-shaking required between the Meta-Device and the specific hardware and/or operating system being worked with. Currently the only fully supported *device* class is the sub-class *x-device*, which works with the X Window System [X 86]. Each device sub-class can be used to create one or more instances, each of which corresponds to an actual device of that class.

Parallel to the *device* class is the *viewport* class. *Viewport* objects define sub-regions of a *device* object, much like windows define sub-regions on a screen. Typically an application first creates an instance of a *device* class and then creates *viewports* on that device object. Viewports act as the basic unit of display area which can receive graphical output commands and can process input events. The Meta-Device, for instance, provides a sub-class of *viewport* called *x-viewport* which is compatible with the *x-device* class. Applications define sub-classes of the *viewport* class by adding new fields and methods; viewports sub-classes inherit the properties and methods of their parent class. Instances of these sub-classes can then be created which stand for actual viewports on actual devices. An example of this is the viewport VIEWPORT-1 shown in Figure 2-1.

The Meta-Device defines methods on each viewport class to handle input and exposure events. These methods are called automatically and asynchronously when the events are received. Most of the default event-handling methods provided by the Meta-Device receive an event but take no action. These defaults methods are inherited by a sub-class of *x-viewport* unless alternative event-handling methods are supplied. Generally, an application defines exposure methods on viewport sub-classes to update the contents of a viewport upon reception of an exposure event, and input methods to determine action based on mouse and keyboard input.

The Meta-Device also provides graphical output methods which enable an application program to draw in a viewport. By defining input event handling methods which invoke graphical output methods, an application is of course free to alter the display in a viewport based on the input events it receives.

An important feature of the Meta-Device is that it allows for each viewport object to have its own coordinate system, referred to as a *mapping*. For example, a viewport that handles text could have a mapping within which units are relevant to a font, and another viewport could have a mapping within which units are equivalent to a centimeter.
2.4 Environment

The Meta-Device is implemented in CMU Common Lisp [McDonald, Fahlman, and Wholey 87], which runs on the IBM RT workstation under the Mach operating system [Mach 86]. The system uses the Common Lisp Object System, CLOS [Bobrow et al. 85], to provide support for class specialization and the object-based interface to the application program.

The only device driver which is currently fully supported is for the X window system, version 10.4; development of the driver for X version 11 is currently underway. The Meta-Device interfaces to the window system through a system dependent foreign-function call mechanism.

3. Programming Interface to the Meta-Device

This section describes the programming interface of the Meta-Device in full detail. Functions are documented using the same conventions as in [Steele 84]. Note that generic function is the CLOS name for an object method, i.e., a function which may be invoked on objects that are instances of potentially many different classes.

3.1 Common Parameter Values

Since some details of the programming interface are repeated in several places with exactly the same meaning, we present them here once and for all. These are typically sets of admissible values for certain parameters that are common to many functions. The rest of this document will then freely refer to them by name whenever needed.

3.1.1 Coordinate Mappings

A MAPPING determines the coordinate mapping used for a device or a viewport. Note that in all the default mappings the Y coordinate increases down the device or viewport. A MAPPING should be one of the following values:

- nil specifies the default mapping, which is rectangular and maps the top-left corner to (0.0, 0.0) and the bottom-right corner to (1.0, 1.0). Positions within a device are interpreted as normalized device coordinates (NDC).
- :pixel maps the top-left corner to (0,0) and the bottom-right corner to (width - 1, height - 1), where width and height are the width and height of the region in pixels.
- a font object (see below) specifies a mapping similar to :pixel except that each unit is the size of an average character in the specified font.
- a vector of four numbers, where the first two numbers define the coordinates of the left-top corner and the next two define the width and height of the region.

To clarify the meaning of a MAPPING a bit further, consider the example in Figure 3-1. The figure shows a device, a viewport (VIEWPORT-3) and a nested viewport, i.e., a viewport whose parent is another viewport.

Since the device was defined with the default mapping (i.e., nil), its coordinates are normalized, with the point (0.0, 0.0) at the top-left corner and the point (1.0, 1.0) and the bottom-right corner. The position of viewport VIEWPORT-3, which is a child of DEVICE-1, is thus specified in normalized device coordinates. Its upper-left corner, for example, is at (0.07, 0.1) in the parent’s coordinate system.

The mapping for VIEWPORT-3, however, was specified as :pixel, which means that the viewport’s own coordinate system is expressed in pixels rather than normalized device coordinates. Consequently,
the position of VIEWPORT-12 (which is a child of VIEWPORT-3) is expressed in the coordinate system of the parent, which is in pixels: the bottom-right corner, for instance, is at (450, 230). This coordinate system is, of course, relative to the parent viewport.

3.1.2 Drawing Modes

A MODE determines the function to be used when drawing graphical objects on devices that are capable of rendering objects in different ways. Some of these modes may only apply to certain graphical operations. The value of a MODE should be one of the following:
- :draw: the object is drawn in an opaque color different from the color of the background.
- :erase: the object is drawn in the same color as the background.
- :or: destination := source OR destination
- :xor (the default): destination := source XOR destination.
- :and destination := source AND destination
- :replace: destination := source
- :nor: destination := NOT source OR destination
- :xnor: destination := NOT source XOR destination
- :nand: destination := NOT source OR NOT destination
- :not: destination := NOT source

3.1.3 Line Styles

A THICKNESS determines the thickness of graphical objects with a line component, such as straight lines and circles. The actual interpretation of this parameter depends on the drawing capabilities of the underlying device. The value should be one of the following:
- :very-thin (the default)
- :thin
- :medium
- :thick
A PATTERN specifies the general appearance of graphical objects with a line component, i.e., whether the line should be continuous or have different dot-dash renderings. The actual interpretation of this parameter depends on the drawing capabilities of the underlying device. The value should be one of the following:

- :solid (the default)
- :dotted
- :dashed

3.1.4 Orientation

The ORIENTATION parameter applies to circles and filled circles. It is used to specify how to compute the radius of a circle in the case when the mapping of a viewport is not the same in the horizontal and vertical directions, i.e., the mapping is not "square". In this situation a radius specification would have different effects depending on whether the radius is to be measured horizontally or vertically. The value should be one of the following:

- :horizontal (the default): the radius of the circle is considered specified in the horizontal direction, and thus is half the "width" of the circle.
- :vertical indicates that the radius is specified in the vertical direction.

3.2 Devices

The Meta-Device provides sub-classes of the device class to interact with specific hardware/software. Currently the only fully supported sub-class of the device class is the x-device, an interactive device which uses the X Window System.

3.2.1 Creation and Destruction

\[\text{make-instance class-name &key :mapping}\]

Returns a device object that communicates with the display specified by the given class-name. For the X window system, this corresponds to the display defined by the Unix environment variable "DISPLAY". Most applications will never need more than one instance of x-device.

\[\text{destroy device}\]

Cleans up and makes device unusable.

3.2.2 Status Information

The following methods can be used to determine the properties of a device. When more than one value is mentioned, the functions return multiple values. Values which represent position or size information are always expressed in the current coordinate system of the object.
Returns the coordinate position along the x-axis of the upper left-hand corner of the device.

top device  [generic function]
Returns the coordinate position along the y-axis of the upper left-hand corner of the device.

width device  [generic function]
Returns the dimension of the device along the x-axis.

height device  [generic function]
Returns the dimension of the device along the y-axis.

mapping device  [generic function]
Returns the current mapping of the device, which is a MAPPING as described above. This method is SETF-able, allowing an application to change the mapping of the device at any time.

children device  [generic function]
Returns a list of the children of the device, i.e., a list of all the meta-device viewports that are currently mapped to the device.

3.3 Viewports

All devices supported in the Meta-Device protocol have some representation of a rectangular display area with a two-dimensional coordinate system. A viewport defines a region within that display area and a coordinate system for that region. Viewports may be arranged hierarchically, that is, not only may the region be within the display area of a device, but also within the display area of another viewport.

3.3.1 Creation and Destruction


Creates a viewport. Most, but not all, of the properties can be altered after the viewport has been created. Properties that are immutable are noted in the documentation below. Any of the :left, :top, :width, and :height arguments may be specified as the keyword :ask-user (the default), which indicates that their numerical value will be based on user input at runtime. The :ask-user value is only available when parent is an instance of the device class. When any of those arguments are :ask-user, the user will be prompted to draw out a viewport and the corresponding keyword argument will be given values based on the position and size of the freshly drawn viewport. The viewport will possibly change size and/or location after it has been drawn out, to match whichever of the dimensions and coordinates were originally given numerical values.

The &key parameters have the following meaning:

:parent Must be specified. It should be either an instance of device or another viewport, and determines the area within which the viewport will be created. This cannot be altered
once the viewport has been created.

:left Coordinate value along the x-axis of the upper left-hand corner of the viewport. This value is relative to the mapping of the parent.

:top Coordinate value along the y-axis of the upper left-hand corner of the viewport. This value is relative to the mapping of the parent.

:width Dimension of the viewport along the x-axis. This value is relative to the mapping of the parent.

:height Dimension of the viewport along the y-axis. This value is relative to the mapping of the parent.

:border May either be true or nil, specifying whether the viewport should have a border. This property cannot be altered once the viewport has been created.

:mapping specifies the mapping to be used in the viewport, and it should be a MAPPING as described in Section 3.1.1.

:reverse-video may be either true or nil. The default will be whatever is specified in the user’s window system customization files, or nil if no default value was specified.

:viewable may be either true or nil. True, the default, specifies that the viewport is visible on the screen immediately upon creation. Note that a viewport’s children will never be visible if their parent is not visible, regardless of their own viewable state.

:track-mouse-moved Determines whether or not the viewport is listening for mouse-moved events. The default value is nil, in which case no input events are generated when the mouse is moved.

:title specifies what string gets displayed in the upper left-hand corner of the display when the user is being prompted to draw a viewport. The default string is “Meta-Device viewport”.

### 3.3.2 Status Information

The following methods can be used to query a viewport about its status.

**left viewport**

Returns the viewport’s coordinate value along the x-axis of the upper left-hand corner in the mapping of the parent. This method is SETF-able.

**top viewport**

Returns the viewport’s coordinate value along the y-axis of the upper left-hand corner in the mapping of the parent. This method is SETF-able.

**width viewport**

Returns the viewport’s dimension along the x-axis. The return value is in the mapping of the parent. This method is SETF-able.

**height viewport**

Returns the viewport’s dimension along the y-axis. The return value is in the mapping of the parent. This method is SETF-able.
mapping viewport

Returns the mapping property of the viewport. This method is SETF-able.

font viewport

Returns the font object of the viewport. This method is SETF-able.

viewable viewport

Returns the viewable status of the viewport. This method is SETF-able.

reverse-video viewport

Returns the reverse-video status of the viewport. This method is SETF-able.

track-mouse-moved viewport

Tells whether or not the viewport is listening for mouse-moved events. This method is SETF-able.

status viewport

Returns as multiple values the status of the viewport in this order: left, top, width, height, mapping. This method is not SETF-able.

parent viewport

Returns the parent of the viewport. This method is not SETF-able.

children viewport

Returns a list of the children of the viewport. This method is not SETF-able.

3.3.3 Modifications

Most of the status methods in the previous section can be used as place arguments to SETF; for example, one can change the position of a viewport with (setf (left my-viewport) 0.5). Used in this manner, status methods become attribute modification functions for viewports. In addition to the individual modification functions, the methods below are also provided for convenience. These methods are potentially more efficient than separate calls to the individual modification functions.

modify viewport &key :left :top :width :height :mapping

Modifies the viewport to correspond with the specified arguments. Arguments not specified will not be altered.

move viewport xy

Modifies the viewport to correspond with the specified arguments. Arguments not specified will not be altered.
Repositions a viewport in the parent coordinate system so that its new top-left corner is at \((x, y)\).

\[\text{destroy viewport} \quad \text{[generic function]}\]

Cleans up and makes the viewport unusable.

\[\text{cleanup-viewport} \text{ viewport} \quad \text{[generic function]}\]

Destroys all Meta-Device viewports.

### 3.3.4 Graphical Operations

The following output methods draw in viewports. Many of these methods may be invoked with style descriptor keyword arguments, such as \texttt{MODE}, \texttt{THICKNESS} and \texttt{PATTERN}, which control subtler aspects of the graphical operation. As described in Section 1, all drawing operations are ephemeral and application programs are responsible for maintaining the drawings in a viewport.

\[\text{clear viewport} \quad \text{[generic function]}\]

Erases any drawings in the viewport.

\[\text{flash viewport} \quad \text{[generic function]}\]

Does something to attract attention to viewport, leaving it in its original state.

\[\text{point viewport} \text{ xy \&key :mode :thickness} \quad \text{[generic function]}\]

Makes a small mark ("draws a point") in viewport at \((x, y)\). The values of \texttt{mode} and \texttt{thickness} should be as indicated in 3.1.1 for \texttt{MODE} and \texttt{THICKNESS}, respectively.

\[\text{line viewport} \text{ x0 y0 xl yl \&key :mode :thickness :pattern} \quad \text{[generic function]}\]

Draws a straight line in viewport from \((x0, y0)\) to \((xl, yl)\). The keyword attributes determine the appearance of the line and default to :invert, :very-thin, and :solid respectively.

\[\text{rectangle viewport} \text{ left top width height \&key :mode :thickness :pattern} \quad \text{[generic function]}\]

Draws a hollow rectangle in viewport with its left-top corner at \((left, top)\).

\[\text{filled-rectangle viewport} \text{ left top width height \&key :mode} \quad \text{[generic function]}\]

Draws a filled rectangle in viewport with its left-top corner at \((left, top)\).

\[\text{circle viewport} \text{ xy radius \&key :mode :thickness :pattern :orientation} \quad \text{[generic function]}\]
Draws a hollow circle in viewport centered at \((x, y)\). The `:orientation` argument, whose admissible values are `:horizontal` (the default) or `:vertical`, is as explained in Section 3.1.1.

```lisp
filled-circle viewport x y radius &key :mode :orientation
```

Draws a filled circle in viewport centered at \((x, y)\).

### 3.3.5 Mapping Conversions

#### 3.3.5.1 Converting Coordinates

The following methods convert a coordinate pair from one viewport's coordinates to another's.

```lisp
map-coordinates-to-parent viewport x y
```

Takes \(x\) and \(y\), which should be in the mapping of the `viewport`, and returns equivalent values in the mapping of the parent.

```lisp
map-coordinates-to-device viewport x y
```

Takes \(x\) and \(y\), which should be in the mapping of the `viewport`, and returns equivalent values in the mapping of the device which is associated with the `viewport`.

```lisp
map-coordinates-to-viewport source-viewport destination-viewport x y
```

Takes \(x\) and \(y\), which should be in the mapping of the `source-viewport`, and returns equivalent values in the mapping of the `destination-viewport`.

```lisp
map-coordinates-from-parent viewport x y
```

Takes \(x\) and \(y\), which should be in the mapping of the parent, and returns equivalent values in the mapping of the `viewport`.

```lisp
map-coordinates-from-device viewport x y
```

Takes \(x\) and \(y\), which should be in the mapping of the device, and returns equivalent values in the mapping of the `viewport`.

#### 3.3.5.2 Converting Dimensions

```lisp
map-dimensions-to-parent viewport width height
```

Takes \(width\) and \(height\), which should be in the mapping of the `viewport`, and returns equivalent values in the mapping of the parent.

```lisp
map-dimensions-to-device viewport width height
```

Takes \(width\) and \(height\), which should be in the mapping of the `viewport`, and returns equivalent values in the mapping of the device.
map-dimensions-to-viewport source-viewport destination-viewport
width height

Takes width and height, which should be in the mapping of the source-viewport, and returns equivalent values in the mapping of the destination-viewport.

map-dimensions-from-parent viewport width height

Takes width and height, which should be in the mapping of the parent of the viewport, and returns equivalent values in the mapping of the viewport.

map-dimensions-from-device viewport width height

Takes width and height, which should be in the mapping of the device, and returns equivalent values in the mapping of the viewport.

3.4 Text and Fonts

A font is used to specify the appearance and size of a piece of text drawn on a device. A font is a collection of "pictures", where each picture represents a particular character. Fonts in the Meta-Device are at the top of a hierarchy separate from the device hierarchy. Currently, the only fully supported font sub-class is x-font, which uses fonts available through the X Window Manager.

3.4.1 Creation and Destruction

make-instance class-name &key :style :face :size

Returns a font object. The font descriptors are analogous to style descriptors for graphical operations. Font descriptors are considered to be hints; the returned font may or may not match all of the specified font descriptors, depending on what is available in the device. The following are the admissible values for the keyword parameters:

:style :
fixed, the default, specifies that the font uses fixed spacing.

:face one of :roman, :italic, or :bold.

:size one of :very-small :small :medium :large :very-large.

destroy font

Cleans-up and makes the font object font unusable.

3.4.2 Status Information

The following methods return the keyword values of the specified font descriptor. An application may use these to test the properties of a font object. These methods are not SETF-able.

style font

Returns the style property of the font.
**face font** [generic function]

Returns the face property of the font.

**size font** [generic function]

Returns the size property of the font.

**descriptors font** [generic function]

Returns, as multiple values, the style, face and size of the font.

### 3.4.3 Drawing Text

**text viewport x y string &key :font :mode** [generic function]

Draws string horizontally in the viewport, with the left-top corner of the bounding box of string at (x,y). The default for the :font keyword is the font currently associated with the viewport.

**string-width viewport string &optional font** [generic function]

Returns the width of the string in the mapping of the viewport. Again, the default font is the one currently associated with the viewport.

**font-height viewport font** [generic function]

Returns the height of an average character from the font in the current mapping of the viewport.

### 3.5 Event Handling

The Meta-Device provides input and exposure methods defined on the viewport sub-classes. These methods are called automatically and asynchronously when an event is received; the method is then free to handle the input event in any way it pleases. Most of the default methods provided by the Meta-Device receive the event but take no action; exceptions are enumerated below. Applications are expected to specialize the viewport class and define their own event-handling methods, unless they intend to inherit the default event-handling provided by the Meta-Device.

#### 3.5.1 Events

An event is an internal structure whose fields get reassigned each time a new event is received. Hence any application wishing to keep a state of the event structure should make a copy of any information it wishes to keep around, since the event will be overwritten when the next event arrives.

Functions are provided to query an event about its fields. Note that for efficiency reasons these are actual Common Lisp functions, as opposed to CLOS generic functions. Also note that not all fields make sense for all types of events. A return value of nil means that the particular information is not available for that kind of event. For instance, requesting the height of a region-exposed event makes sense, while requesting the height of a button-pressed event does not. The event querying functions are:
x-event-x \textit{event} \hspace{1cm} [\textit{function}] \\
Returns the \textit{x} coordinate of the mouse position at the time the event happened.

x-event-y \textit{event} \hspace{1cm} [\textit{function}] \\
Returns the \textit{y} coordinate of the mouse position at the time the event happened.

x-event-button \textit{event} \hspace{1cm} [\textit{function}] \\
Returns which button was used. Possible values are :left, :middle, and :right. \texttt{nil} is returned if the event did not involve a mouse button.

x-event-character \textit{event} \hspace{1cm} [\textit{function}] \\
Returns which character was typed. This will be, in general, a Common Lisp character object.

x-event-left \textit{event} \hspace{1cm} [\textit{function}] \\
Returns the coordinate along the \textit{x}-axis of the upper left-hand corner of the region. This and the following functions only make sense for events which involve a region, like :viewport-exposed and :region-exposed.

x-event-top \textit{event} \hspace{1cm} [\textit{function}] \\
Returns the coordinate along the \textit{y}-axis of the upper left-hand corner of the region.

x-event-width \textit{event} \hspace{1cm} [\textit{function}] \\
Returns the dimension along the \textit{x}-axis of the region.

x-event-height \textit{event} \hspace{1cm} [\textit{function}] \\
Returns the dimension along the \textit{y}-axis of the region.

\subsection*{3.5.2 Defining Event Handling Methods}

When defining an event handling method for a sub-class of \textit{viewport}, the method must have a lambda-list that is congruent with pre-defined event handling methods. The pre-defined methods all take two arguments: (1) an object of class \textit{viewport}, or subclass thereof, and (2) an \textit{event}.

Methods for particular input events are defined with \texttt{defmethod}, so an application program which has defined a sub-class \texttt{my-viewport-class} could specify a handler for mouse buttons in all viewports of the sub-class by the following code fragment:

\begin{verbatim}
(defmethod button-pressed ((vp my-viewport-class) event) ;; insert code here to handle the button-down event.
)
\end{verbatim}
See the last section of the document for examples of event-handling code. The event-handling methods for a viewport are:

**button-pressed viewport event**

Default: no action. The event contains information about the \( x \) and \( y \) of the mouse position, the modifiers used, and which button was pressed.

**button-released viewport event**

Default: no action. The event contains information about the \( x \) and \( y \) of the mouse position, the modifiers used, and which button was released.

**key-pressed viewport event**

Default: no action. The event contains information about the \( x \) and \( y \) of the mouse position and which character was typed.

**mouse-moved viewport event**

Default: no action. The event contains information about the \( x \) and \( y \) of the mouse position and the modifiers used.

**region-exposed viewport event**

Default action: clear the regions of the viewport that were exposed. The event contains information about the coordinates and dimensions of the exposed region. If several regions of the viewport are exposed at once, the Meta-Device tries to generate as few region-exposed events as possible.

**viewport-entered viewport event**

Default action: changes the viewport border from gray to the opposite of the background.

**viewport-exited viewport event**

Default action: changes the viewport border back to gray.

**viewport-exposed viewport event**

Default: no action. The event contains information about the coordinates and dimensions of the exposed region.

### 4. An Example

This section contains examples of a few very simple operations using the Meta-Device, and is organized as the transcript of an interactive session. Please note that this example was created by typing expressions to the top level read-eval-print loop in Lisp, and therefore an actual Lisp program would look rather
different.

User input is preceded by the Lisp prompt (a *, in this case); the return value of each expression is also shown, even though in most cases it is not particularly relevant. Comments have been freely added to the transcript of the session, and are marked by three semicolons at the beginning of a line.

;;; NOTE: loading instructions are specific to CMU Common Lisp.
;;; * (load "meta-device:dante-loader")

;;; make all meta-device functions directly available.
* (use-package "META-DEVICE")
T

;;; Create an instance of the x-device class.
* (setf d (make-instance 'x-device))
#<Console X-Device>

;;; Define a sub-class of x-viewport.
* (defclass new-vp (x-viewport) ()
NIL

;;; Create a viewport to work with, use device "d" as the parent.
;;; Make the viewport an instance of the new class.
* (setf v (make-instance 'new-vp :parent d
  :left 0.03 :top 0.2 :height 0.3 :width 0.3
  :reverse-video nil))
#<X-Viewport #17105092>

;;; Create a viewport within viewport "v", use reverse video.
* (setf sub-v (make-instance 'new-vp :parent v
  :left 0.4 :top 0.4
  :width 0.55 :height 0.25
  :reverse-video T))
#<X-Viewport #17170627>

;;; Define a handler for input keystrokes for our viewports. This
;;; inverts a rectangle when "i" is typed and clears the viewport when
;;; "c" is typed.
;;; *
;;; * (defmethod key-pressed ((vp new-vp) event)
; (case (x-event-character event)
  (#\c ; Clear the whole viewport.
   (clear vp))
  (#\i ; Invert a rectangle in the viewport.
   (filled-rectangle vp 0.1 0.1 0.5 0.87 :mode :invert))))
NIL
*

At this point, let us imagine that the user moves the cursor over viewport v, i.e., the exterior viewport, and types the single key "i". The two viewports will now look as shown in Figure 4-1.

The exterior viewport corresponds to the variable v in the example. Its background is white, and the
input event handler which was invoked when the user typed the key "i" caused a black rectangle to be drawn. This rectangle is partially covered by the viewport sub-vp, which is wide and short. Notice that since sub-vp was created with :reverse-video equal to T, its background is black. Neither viewport has a title.

The second portion of the example creates yet another viewport, named another-vp. Since the call to make-instance does not specify any of the viewport dimensions, the user is queried for the position and size of the viewport. In the X window system, for instance, this will ask the user to draw the outline of the viewport with one of the mouse buttons. We then define a handler for mouse clicks in the viewport we have created, and finally move another-vp to a different position within the display.

;;; create a new viewport, let the user specify its size using the
;;; middle button.
* (setf another-vp (make-instance 'new-vp .-parent d))
#<X-Viewport #17236162>

;;; Define a handler for mouse clicks:
;;; - pressing left-button should make a line appear;
;;; - pressing middle or right should display a filled-circle.
;;; *
* (defmethod button-pressed ((vp new-vp) event)
  (if (eq (x-event-button event) :left)
    ;; LEFT BUTTON clicked: draw a random line.
    (let ((x (random 1.0))
           (y (random 1.0)))
      (line vp 0 0 x y :mode :draw))
    ;; MIDDLE/RIGHT: draw a circle centered around the cursor.
    (filled-circle vp (x-event-x event) (x-event-y event)
                   0.1 :mode :xor)))
NIL

;;; Move another-vp so it is halfway across the screen.
* (setf (left another-vp) 0.5)
0.5
After this example, suppose that the user clicks the left button six times and the middle button once in another-vp. The viewport would then look as in Figure 4-2. Notice that the lines are drawn from the top-left of the viewport (position 0,0) to a random point in the viewport. The filled circle, on the other hand, is always centered around the position of the cursor when the user clicks a button other than the left one.

Figure 4-2: Viewport ANOTHER-VP after several mouse clicks

5. Summary

The Meta-Device implements the lowest layer of graphical input and output. While it does not provide for retained graphical objects, which are implemented by other layers above it, it gives application programs a completely device- and window-system independent mechanism to describe graphical output operations and to handle input from the user.

Because it does not provide retained graphical objects, the Meta-Device driver is ideally suited as a building block for more elaborate layers and as a graphical interface for application programs which cannot afford the performance and storage overhead of retained graphical objects. In addition, the Meta-Device uses the object-oriented paradigm extensively in its programming interface, and thus makes it very convenient for application programs to customize any particular device or viewport to their specific needs.

The system is organized around a few central class hierarchies, such as the device hierarchy and the viewport hierarchy. The device hierarchy is the basic mechanism through which implementors may port the system to new devices. Class inheritance may be used to simplify the initial implementation of a new device driver and reduce the size of the system via sharing of methods. The viewport hierarchy, on the other hand, is the basic mechanism through which application program developers may specialize the behavior of a particular device, in particular for handling input events in a powerful, flexible way.

The Meta-Device is entirely implemented in Common Lisp and uses CLOS, the Common Lisp Object System, in its programming interface. This results in an interface which is very flexible and easy to extend and modify as needed by the application program developer, while at the same time ensuring complete portability of application programs to different hardware and window systems. The explicit omission of retained graphical objects makes the system ideally suited for applications which do their
own object management and cannot afford the high overhead of going through a graphics layer which duplicates that functionality.

6. Acknowledgments

We gratefully acknowledge the many contributions of Roy F. Busdiecker to the Meta-Device. He was responsible for the initial implementation of the Meta-Device and its maintenance through several changes of the underlying Lisp and CLOS systems. He was also the author of a preliminary version of this document.
References


