The
Connection Machine
System

*Graphics Reference Manual

Version 6.0
November 1990

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, November 1989
Revised, November 1990

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

About This Manual v
Customer Support viii

Chapter 1 *Graphics: *Lisp Graphics Interface **1**

1.1 Overview of Display Functions 1
1.1.1 Levels of Display Function Interface 2
1.2 Overview of Rendering Functions 4
1.3 Using *Graphics 5

Chapter 2 Display Windows **7**

2.1 Creating a Display Window 7
2.1.1 Display Window Type, Location, and Size Arguments 8
2.1.2 Other Display Window Property Arguments 10
2.2 Getting Information about a Display Window 11
2.3 Selecting a Display Window 12
2.4 Reading and Writing to Display Windows 12
2.5 Deleting Display Windows 14
2.6 Hardware Panning and Zooming 14

Chapter 3 Color Maps **17**

3.1 Overview of Device-Level and Symbolic Color Map Interfaces 17
3.2 Symbolic Color Map Interface 18
3.2.1 Creating Colors 18
3.2.2 Creating Color Ranges 19
3.2.3 Creating Color Maps 20
3.2.4 Example 22
3.2.5 Selecting a Color Map 23
3.2.6 Getting Information about a Color Map 23
3.3 Device-Level Color Map Interface 25
3.3.1 Getting Information about Device Color Maps 26

3.3.2 Setting the Device-Level Color Map 26
3.3.3 Reading Contents of Device-Level Color Maps 27
3.4 Using 24-bit Color Maps 27
3.5 Color Maps on X Windows 27

Chapter 4 Displays **29**

4.1 Ways to Use Displays 29
4.2 Creating Displays 30
4.3 Getting Information about a Display 31
4.4 Displaying an Image 31
4.4.1 Specifying How Much of the Image to Display 32
4.4.2 Specifying Where to Display the Image 32
4.4.3 Specifying the Colors for Displaying the Image 33
Numeric Pvars 33
Boolean Pvars 34
4.4.4 Specifying Multiple Images to Overlay 35
4.4.5 Specifying Other Features 35
4.4.6 Example 36
4.5 Clearing Displays 37
4.6 Reading a Pvar from a Display 37
4.7 Hardware Panning and Zooming of Displays 38

Chapter 5 Rendering Primitives **39**

5.1 Two-Dimensional Rendering Functions 39
5.2 Three-Dimensional Rendering Functions 42

Chapter 6 Z-Buffer Functions **43**

Chapter 7 Math Utilities **47**

Index 53

About This Manual

Objectives of This Manual

This manual describes *Graphics, the *Lisp interface to the Connection Machine graphics and visualization software.

Revision Information

This manual replaces the **Graphics Reference Manual*, Beta Version 5.2. You may discard that manual.

Version 6.0 Changes

Version 6.0 of the *Graphics software includes three routines not included in Version 5.2:

- **polygon–fill** draws draws filled polygons defined by pvars of vertices:

  polygon–fill *image–buffer–pvar x–vector–pvar y–vector–pvar color–pvar*
  &key :number–of–vertices :combiner :overwrite
  :clip–p :edge–color–pvar

  This function is described in Section 5.1 of this manual.

- ***new–draw–lines–2d** is a replacement for the ***draw–lines–2d** function with improved memory management:

  ***new–draw–lines–2d** *image–buffer–pvar x–start–pvar y–start–pvar*
  x–end–pvar y–end–pvar color–pvar
  &key (combiner :overwrite) (:clip–p t)

  This function is described in Section 5.1 in this manual.

- **display–window–cmsr–display** provides a hook to the Paris-level Generic Display Interface by returning the Generic Display display (an instance of **cmsr:display**) that implements a *Graphics display window:

  display–window–cmsr–display &optional *display–window*

This function is described in Section 2.2 of this manual.

In addition, three changes have been made to existing *Graphics routines:

- **display–image** and **display–image!!** now clip images that are larger than the display window, rather than signalling an error. See Section 4.4.1.
- **display–image** and **display–image!!** now support software zooming. The argument *zoom*, which defaults to 1, is the factor by which to zoom each pixel. See Section 4.4.5. An example of zooming has also been added to Section 4.4.6.
- The use of the **create–display–window** argument **:color–map** has been clarified. See Section 2.1.2.

Organization of This Manual

This manual contains seven chapters:

Chapter 1 ***Graphics: *Lisp Graphics Interface**
An overview of the *Graphics display and rendering functions and information on using *Graphics.

Chapter 2 **Display Windows**
Detailed information on using *Graphics display windows, including creating, selecting, reading and writing, and deleting display windows.

Chapter 3 **Color Maps**
Detailed information on *Graphics symbolic and device–level color map interfaces.

Chapter 4 **Displays**
Detailed information on setting up and using *Graphics displays.

Chapter 5 **Rendering Primitives**
Reference information on *Graphics 2D and 3D rendering functions.

Chapter 6 **Z–Buffer Functions**
Reference information on the *Graphics functions that create and control a z–buffer on the Connection Machine. A z–buffer is used in 3D graphics to perform hidden surface removal.

Chapter 7 Math Utilities
Describes the graphics math utilities supplied by *Graphics. These utilities provide tools for creating and applying coordinate transformations.

Notation Conventions

The table below displays the notation conventions:

Convention	Meaning
`bold typewriter`	UNIX and CM System Software commands, command options, and file names.
`bold typewriter`	C/Paris and C language elements, such as keywords, operators, and function names, when they appear embedded in text.
`italic typewriter`	Parameter names and placeholders in function and command formats.
`typewriter`	Code examples and code fragments.
% **`bold typewriter`** `typewriter`	In interactive examples, user input is shown in **`bold typewriter`** and system output is shown in `regular typewriter` font.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
Usenet Electronic Mail:	ames!think!customer-support
Telephone:	(617) 234–1000 (617) 876–1111

For Symbolics Users Only

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl–M to create a report. In the mail window that appears, the `To:` field should be addressed as follows:

`To:` customer–support@think.com

Please supplement the automatic report with any further pertinent information.

Chapter 1

*Graphics: *Lisp Graphics Interface

The *Lisp graphics interface, *Graphics (pronounced "star-graphics"), provides a set of functions and data structures that make it easy to display and manipulate 2-dimensional and simple 3-dimensional pvar data.

*Graphics includes two main types of functionality: display and rendering. The display functions write 2-dimensional pvars to a display device. The rendering functions generate 2-dimensional pvars inside the Connection Machine system; the pvars can then be displayed.

1.1 Overview of Display Functions

*Graphics provides a uniform interface for displaying pvars on CM Framebuffers, X windows, and Symbolics color and black-and-white windows from *Lisp.

The display tools provided are as follows:

1. **Display windows** are structures that represent a device onto which data is displayed. Functions dealing with display windows allow a user to create multiple windows, obtain information about each window, write pvar data to the display window, read data from the display window back into a pvar, delete display windows, and pan and zoom the display window's device.

2. **Color maps** define how values in pvars being displayed are mapped onto the colors shown on a display window.

Two levels of color maps are supported: device-level and symbolic color maps. At the device level, color maps are simply arrays that specify the mapping from pixel values to RGB intensities. At the symbolic level, color maps are mappings from named colors (like "red") and color ranges (like "rainbow") to pixel values.

Only display window devices that support color maps in hardware have device-level color maps. On the other hand, symbolic color maps can be defined for any type of display window. As a result, user code that uses colors can be run in a generic fashion on monochrome as well as color devices.

At the device level, users can change the color map of a display window and change individual entries in the color map. At the symbolic level, users can define and reference colors, color ranges, and color maps, and can change the color map of a display window.

3. **Displays** are rectangular regions of a display window onto which images are displayed. Displays can be positioned either by the user or automatically by a program that keeps track of empty space on the display window.

   User-managed displays are useful for implementing demos, where images are to be displayed in a predetermined format. Automatically positioned displays are particularly handy for debugging, with images positioned side by side.

1.1.1 Levels of Display Function Interface

The display functions of *Graphics can viewed as providing two levels of interface.

The display-level interface is useful for interactively displaying both images and building blocks for programs. It automatically rescales, dithers, positions, and sizes images for display. It lets the user overlay images, and provides functionality for defining color maps and for referencing color slots symbolically. The high-level display interface is based on that of the CM Vision Utilities written by Thinking Machines Corporation.

The display window interface, on which the higher-level display interface is built, is designed for users who want more detailed control or who don't require the model that the high-level interface provides. It allows the user to send pvar data to the display device, with rescaling, dithering, positioning, sizing, and color maps managed by the user. The low-level interface is in turn built on *Render and the Generic Display Interface.

Figure 1 shows the relationship between displays and display windows and lists the typical functions for each level.

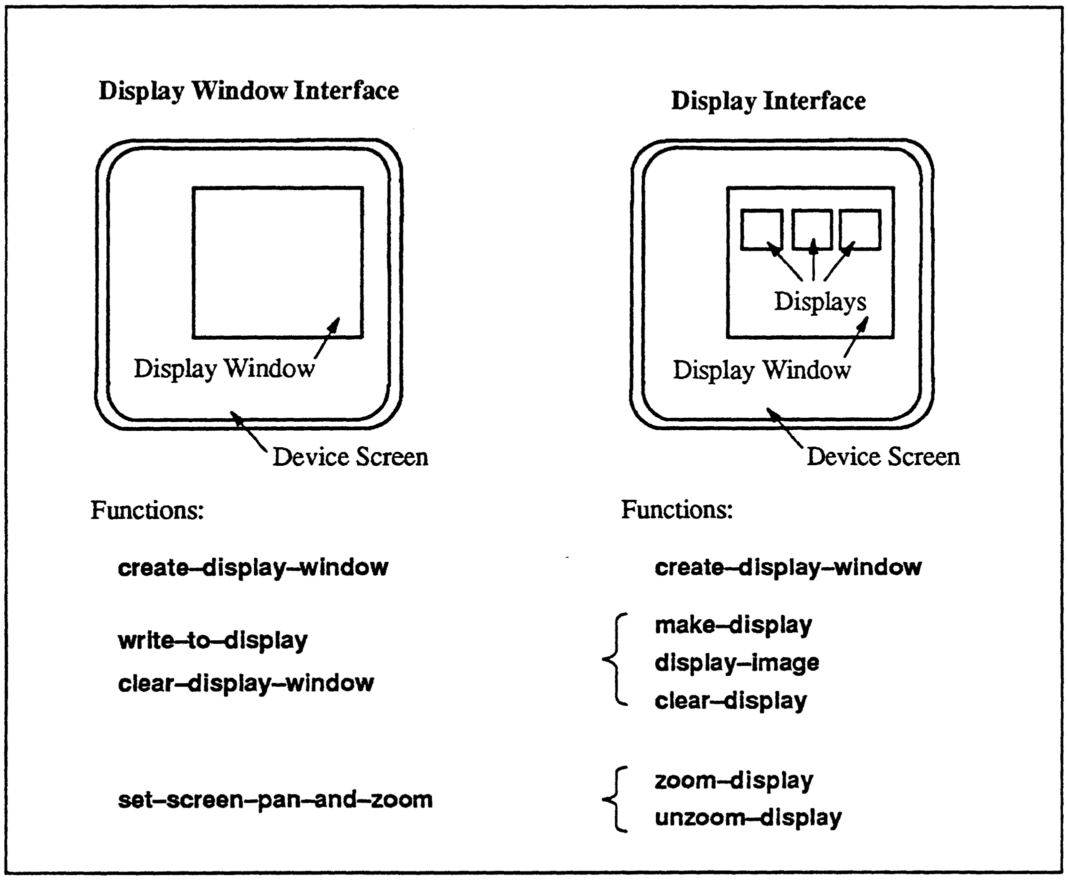

Figure 1 The relationship between displays and display windows, and the functions that users typically use for each level of interface

Figure 2 shows the two levels of color map interface and lists the typical functions.

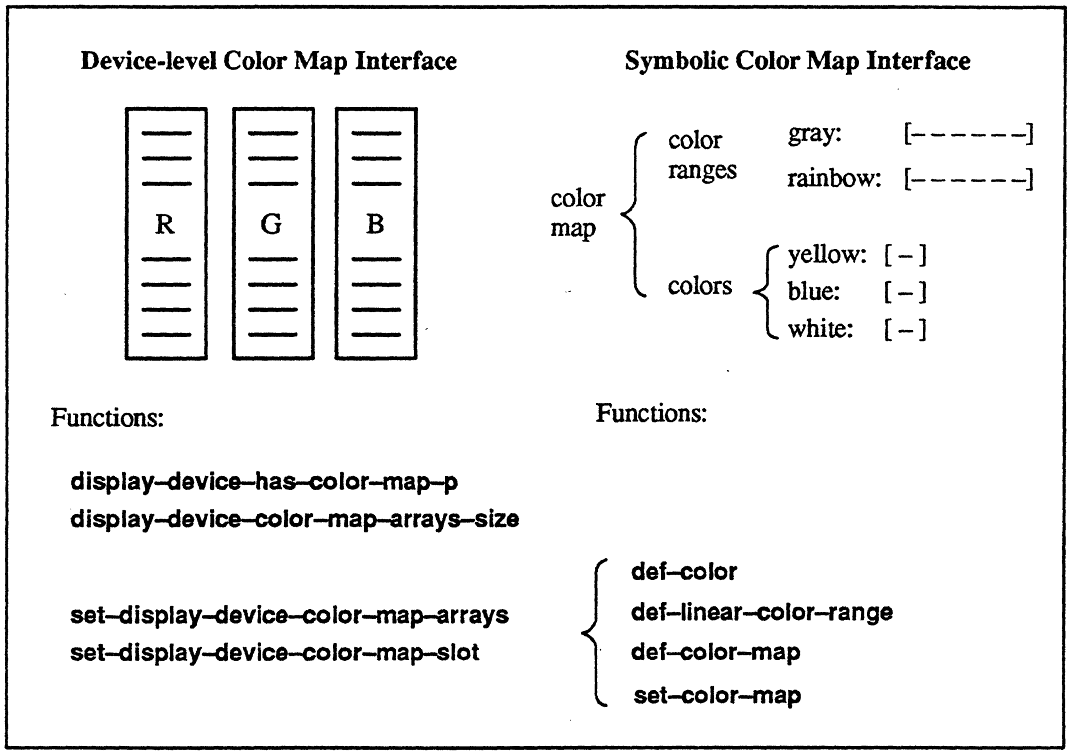
Device-level Color Map Interface
Symbolic Color Map Interface
R
G
B
color map
color ranges
gray: [------]
rainbow: [------]
colors
yellow: [-]
blue: [-]
white: [-]
Functions:
display-device-has-color-map-p
display-device-color-map-arrays-size
set-display-device-color-map-arrays
set-display-device-color-map-slot
Functions:
def-color
def-linear-color-range
def-color-map
set-color-map

Figure 2. The two levels of color map interface, and the functions typically used at each level

1.2 Overview of Rendering Functions

The rendering utilities are built on the Paris-level package *Render (pronounced "star-render"). These utilities are as follows:

1. **Z-buffers** are data structures used in rendering 3-dimensional objects. They are primarily useful for doing hidden line and surface removal. Users can create and delete Z-buffers and use the rendering functions described below to create images within a Z-buffer. The data can then be displayed using the display routines.

2. **Rendering primitives**, both 2- and 3-dimensional, are provided, including functions for drawing points and lines. The 3-dimensional versions create an image to

be displayed within a Z-buffer; the 2-dimensional versions create a pvar image that can be directly displayed.

3. **Math functions** are provided to deal with vector and matrix operations, especially those dealing with transformation matrices that allow the rotation, translation, and scaling of image coordinates.

1.3 Using *Graphics

The *Lisp graphics interface resides in the ***graphics** package. Users may wish to make the package in which they program (for example, ***lisp**) use the ***graphics** package, so that all *Graphics functions and variables are easily accessible. For example:

```
(in-package '*lisp :use '(*graphics))
```

If you do not do this, then all calls to functions and references to variables described herein must be prefixed by ***graphics:** or by ***g:** (***g** is a nickname for the ***graphics** package).

Example programs using the *Graphics package may be found in the file **examples. lisp**. In the 5.2 release of the CM System Software, this file is in the directory **/cm/starlisp/graphics/f5203/**.

Chapter 2

Display Windows

To use *Graphics, you must first create a display window using the function **create-display-window**. You can create more than one display window.

Display output is directed to the current display window. There are two ways to change the current display window. The function **create-display-window** sets by default the current display window to the window it creates. The function **set-display-window** sets the current display window to a display window that you specify.

Display window coordinates are specified as two-element lists of the form (*x y*), indicating offsets to the right and down, respectively, from the upper left corner of the screen or window. The *x* and *y* coordinates of the display correspond to dimensions 0 and 1, respectively, of a 2-dimensional pvar.

2.1 Creating a Display Window

The **create-display-window** function creates and returns a display window structure. Its syntax is as follows:

```
create-display-window  &key  :display-window-type                  [Function]
                             :physical-device-pointer
                             :desired-bits-per-pixel
                             :desired-width     :desired-height
                             :name              :display-padding
                             :make-current      :color-map
```

You can simply call **create-display-window** with no arguments to create a window of default size on a device selected from a menu of display devices, including (when appropriate) X Windows, CM Framebuffers, and Symbolics screens. The keyword arguments let you specify other options and avoid prompting. To avoid the menu, you can specify the type, location, size, and other properties of the returned display window by specifying values for the appropriate keyword arguments.

2.1.1 Display Window Type, Location, and Size Arguments

The following **create–display–window** arguments are used to specify the type, location, and size of the returned display window:

:display–window–type	Specifies type of returned window.
:physical–device–pointer	Specifies output device for window.
:desired–bits–per–pixel	Specifies desired bit depth for window.
:desired–width	Specifies desired window pixel width.
:desired–height	Specifies desired window pizel height.

The permitted values for these arguments are interrelated, and depend on the value provided for the **:display–window–type** argument.

The **:display–window–type** argument must be one of the following keywords:

:cmfb	Connection Machine Framebuffer display.
:xwindow	X Window on either local or remote host.
:symbolics	Symbolics window (color or black-and-white screen).
:symbolics–frame	A frame consisting of a Lisp Listener and display panel on the Symbolics black-and-white screen.

The **:cmfb** and **:xwindow** keywords may be used on any front end.The **:symbolics** and **:symbolics–frame** keywords may be used only on a Symbolics front end.

The **:physical–device–pointer** argument directs output to a specific device, avoiding prompting or automatic defaulting. This argument may be used only if a value has been specified for **:display–window–type.**

The legal values for **:physical–device–pointer** depend on the value of **:display–window–type** as follows:

:display–window–type value:	Legal **:physical–device–pointer** value(s):
:cmfb	Non-empty string, which must be a member of the list **cmfb:available–displays.** Defaults to first element of **cmfb:available–displays.**

:xwindow	String of the form "hostname:screen" (for example "unix:0"). Under Lucid this defaults to the value of the **display** environment variable. On Symbolics front ends, this defaults to Symbolics host display.
:symbolics	The **:physical–device–pointer** argument is currently ignored on Symbolics front ends. If the **:desired–bits–per–pixel** is 1, then **tv:main–screen** is used; otherwise the value of **(color: find–color–screen)** is used.
:symbolics–frame	The **:physical–device–pointer** argument is currently ignored on Symbolics front ends. The value of **tv:main–screen** is used.

The **:desired–bits–per–pixel** argument specifies the number of bits desired. It is used when a device, such as a CM Framebuffer, supports more than 1-bit depth for its display (for example, a display might support both 8- and 24-bit color).

The legal values for **:desired–bits–per–pixel** depend on the value of **:display–window–type** as follows:

:display–window–type value:	Legal **:desired–bits–per–pixel** value(s):
:cmfb	Can be either 8 or 24; defaults to 8.
:xwindow	Can be 1, 8, or 24. The default value is determined by the selected output device.
:symbolics	If color screen is available, can be 1, 8, or 24. If only console screen is available, must be 1. Defaults to 1 in either case.
:symbolics–frame	Must be 1, and defaults to 1.

The arguments **:desired–width** and **:desired–height** specify the dimensions of the window in pixels.

The legal values for **:desired–width** and **:desired–height** depend on **:display–window–type** as follows:

:display–window–type value:	Legal **:desired–bits–per–pixel** value(s):
:cmfb	Any supplied arguments are ignored; the entire screen is used.

:xwindow	Positive integers. Defaults to 256 by 256 display.
:symbolics	Positive integers. Defaults to size of screen.
:symbolics–frame	Any supplied arguments are ignored; the entire screen is used.

Owing to constraints imposed by the specified output device, the returned display window may not have exactly the specified values for **:desired–bits–per–pixel**, **:desired–width**, and **:desired–height**. For this reason, the function **create–display–window** returns four values:

- the display window
- the actual bit depth (number of bits per pixel) of the display window
- the actual width of the display window
- the actual height of the display window

2.1.2 Other Display Window Property Arguments

The argument **:name** optionally specifies the name of the newly created display window. If specified, it must be a string. If not specified, a name is generated that indicates the type of window.

The argument **:make–current**, which defaults to **t**, specifies whether the display window being created is to be the current display window.

The following arguments are relevant if displays are to be used with the display window:

- **:color–map**, if specified, must be a symbolic color map or the name of a symbolic color map. It should be specified only for 8-bit windows (although it can be specified for 1-bit and 24-bit windows as well). If not specified, it defaults as follows:

    - for 8-bit pseudo-color windows and 1-bit dithered windows, **:color–map** defaults to **:gray–and–rainbow**

    - for 24-bit RGB display windows, **:color–map** defaults to **:rgb** (see Section 3.4).

  Otherwise, the color map is not set.

  To prevent **create–display–window** from touching the color map, specify **:color–map** as **nil**.

- **:display–padding** specifies the spacing, in pixels, surrounding each automatically generated display within a display window. It defaults to 16. If the window is to be created just big enough to display a single image, **:display-padding** should be set to 0.

2.2 Getting Information about a Display Window

The functions listed below provide various kinds of information about display windows.

display-window-bits-per-pixel **&optional** *display-window* [Function]

returns the number of bits to represent a value for display window. *display-window* defaults to the current display window.

display-window-width **&optional** *display-window* [Function]

returns the width, in pixels, of a display window. *display-window* defaults to the current display window.

display-window-height **&optional** *display-window* [Function]

returns the height, in pixels, of a display window. *display-window* defaults to the current display window.

display-window-color-map *display-window* [Function]

returns the symbolic color map associated with a display window. (Symbolic color maps are described in Sections 3.1 and 3.2.) *display-window* defaults to the current display window.

display-window-cmsr-display **&optional** *display-window* [Function]

returns the generic display (an instance of **cmsr:display**) that implements *display-window*. *display-window* defaults to the current display window. This function provides a hook to the Paris-level Generic Display Interface.

valid-display-window-p *object* [Function]

returns **t** if *object* is a display window that has not been deleted.

2.3 Selecting a Display Window

The functions listed below have to do with selecting a display window.

set–display–window *display–window* [Function]

makes *display–window* be the current display window. It is an error if *display–window* is not a valid display window.

with–display–window *display–window* **&body** *body* [Macro]

evaluates *body* in the context of *display–window* being the current display window. After the form has been exited, the current display window reverts to what it was before the form was entered. *display–window* must be a valid display window.

current–display–window [Variable]

displays the current display window, or **nil** if no display window is current.

all–display–windows [Variable]

displays a list of all valid (non-deleted) display windows.

2.4 Reading and Writing to Display Windows

The functions listed below have to do with reading and writing to display windows, clearing display windows, and rescaling the pvar argument.

write–display–window *pvar* **&key :pvar–start :display–start :size** [*Defun]

The data within *pvar* is displayed on the device represented by the current display window. The data with *pvar* must be coercible into a field pvar of length **(display–window–bits–per–pixel)** bits. That is, floating-point data is not acceptable but a signed pvar that contains only non-negative values is acceptable as long as the values are small enough.

pvar itself must belong to a 2-dimensional VP set. Its geometry can be in either grid or framebuffer order. Framebuffer-ordered geometries are explained in the *Graphics Programming Release Notes* for Version 5.2.

:pvar–start specifies a grid address of the upper left corner from which the data to be written is taken. It is a two-element list (*x y*).

:display–start specifies the display coordinates of the upper left corner to which the data is written. It is a two-element list (*x y*).

:size specifies the extent in each dimension. **:size** defaults to the maximum extent possible, given the pvar size, the display size, and the start arguments (if provided). It is a two-element list (width, height).

To rescale the *pvar* argument of function **write–display–window**, use the following function:

```
rescale–pvar–for–display–window!! pvar &key :result–type          [Function]
                                            :result–min
                                            :result–max
                                            :pvar–min
                                            :pvar–max
                                            :result–underflow
                                            :result–overflow
```

This function rescales *pvar*, returning a new pvar of element-type **:result–type**, which defaults to a field pvar of size **(display–window–bits–per–pixel)**.

pvar must belong to the current VP set, and must be of a non-complex numeric or boolean type.

The function linearly maps each value of *pvar* so that **:pvar–min** maps to **:result–min** and **:pvar–max** maps to **:result–max**. **:pvar–min** and **:pvar–max** default to the actual minimum and maximum values of *pvar*; **:result–min**, and **:result–max** default to the minimum and maximum values which can be stored in the result pvar, except that if the **:result–type** is floating point, then the minimum value defaults to 0.0 and the maximum value defaults to 1.0.

Any values in *pvar* less than **:pvar–min** map to **:result–underflow** (which defaults to **:result–min**) and any values greater than **:pvar–max** map to **:result–overflow** (which defaults to **:result–max**). If *pvar* is boolean, **nil** maps to **:result–min** and **t** maps to **:result–max**.

The **:result–** options are particularly useful when the color map contains more than one range, as does the default color map. **:result–min** and **:result–max** can indicate the range, and **:result–overflow** and **:result–underflow** can be mapped to special color indices.

read–display–window &key :result–pvar :pvar–start display–start :size [Function]

copies contents of display window into **:result–pvar**, if provided. If not provided, an unsigned-byte pvar of the appropriate length is created and returned.

If **:result–pvar** is not provided, the current VP set must be 2-dimensional. If **:result–pvar** is provided, it must be an unsigned-byte pvar of length greater than or equal to **display–window–bits–per–pixel** bits. The pvar must belong to a 2-dimensional VP set, which need not be the current VP set and may be in either grid or framebuffer order. Framebuffer-ordered geometries are explained in the *Graphics Programming Release Notes* for Version 5.2.

The arguments **:pvar–start**, **:display–start**, and **:size** are as in **write–display–window.**

clear–display–window [Function]

clears the current display window by setting all values to 0 (the background color). Clears pointers to any displays automatically created by the function **display–image**.

2.5 Deleting Display Windows

There are two functions for deleting display windows:

delete–display–window *display–window* [Function]

kills the *display–window*, clearing it from screen. It is an error to use the deleted display window again. All displays within the display window are deleted.

delete–all–display–windows [Function]

kills all display windows.

2.6 Hardware Panning and Zooming

Generic functions exist for hardware that supports panning and zooming. The CM Framebuffer and some Symbolics color screens have this capability.

screen–pan–and–zoom–p [Function]

returns **t** if the screen containing the current display window supports hardware panning and zooming; otherwise returns **nil**.

set–screen–pan–and–zoom &optional *pan–x pan–y zoom–x zoom–y* [Function]

pans and zooms the screen containing the current display window, if it supports hardware panning and zooming; otherwise, it does nothing. *pan–x* and *pan–y* are the coordinates to appear in the upper left corner of the screen; these arguments both defrault to 0. *zoom–x* and *zoom–y* are the additional amount by which each pixel is replicated in each dimension. *zoom–x* defaults to 0, *zoom–y* defaults to the value at *zoom–x*. Thus, 0 means no zooming, 1 means double each pixel, 2 means triple each pixel, etc. Because *zoom–y* defaults to *zoom–x,* the zoom argument need not be repeated if it is the same in both dimensions. Because the arguments default to 0, calling the function with no arguments resets the screen to its initial state (no panning or zooming).

This function returns the new values of *pan–x, pan–y, zoom–x,* and *zoom–y*. If the screen does not support hardware panning and zooming, each of these values will be 0.

get–screen–pan–and–zoom [Function]

returns four values: *pan–x, pan–y, zoom–x,* and *zoom–y,* describing the current pan and zoom of the screen containing the current display window. If the screen does not support hardware panning and zooming, each of these values will be 0.

Chapter 3

Color Maps

*Graphics provides facilities for defining color maps for pseudo-color (8-bit) and 24-bit screens.

Two levels of interface are provided: a device-level interface, and a higher-level, symbolic interface.

3.1 Overview of Device-Level and Symbolic Color Map Interfaces

With the device-level interface, color maps are simply specified as a set of three arrays that map screen pixel values to red, green, and blue intensities. Each display window on a device that supports color maps has its own device color map. The following functions comprise the device-level interface:

display–device–has–color–map–p
display–device–color–map–array–size
set–display–device–color–map–arrays
display–device–color–map–arrays
set–display–device–color–map–slot
display–device–color–map–slot

The high-level, symbolic color map interface allows you to define a named color map as a combination of named component colors and color ranges. The symbolic interface serves several purposes:

- It lets you define a color map, independent of its actual size, without having to manage the allocation of individual slots.

- It lets you reference the color and color ranges by name rather than by slot number. (See function **display-image**, Section 4.4, which makes use of symbolic color maps.)
- Since a symbolic color map is associated with every display window, whether or not the underlying device actually supports color maps, you can write generic code that uses colors and color ranges. On black-and-white devices the system does its best to display such images instead of signaling an error.

The symbolic color map functions include:

def–color
create–color
def–linear–color–range
create–linear–color–range
def–color–map
create–color–map
set–color–map
current–color–map

Functions for getting information about color maps are also provided.

3.2 Symbolic Color Map Interface

The symbolic color interface lets you define and reference pseudo-color (8-bit) color maps.

The following subsections describe functions for defining colors, color ranges, and color maps. Examples using these functions are provided in Section 3.2.4. Functions for activating color maps and for getting information about color maps are also described.

3.2.1 Creating Colors

A color is a named RGB triple, which can be referred to symbolically in a color map.

See Section 3.2.4 for examples.

create–color *name space value* [Function]

creates and returns an object of type **color**.

name must be a keyword, like **:red**. *name* can be used to symbolically refer to the color when it is used in a color map.

space is the color space used for defining the color. Currently, the only legal value for *space* is **:rgb**.

The *value* provided is dependent on *space*. Currently, *value* must be a list or vector of three floating-point numbers between 0.0 and 1.0 inclusive, representing the red, green, and blue intensities.

def–color *name space value* [Macro]

creates a color, by calling the function **create–color**, and pushes the color onto the list ***defined–colors***, replacing any previously defined colors having the same name.

defined–colors [Variable]

is a list of all colors defined by **def–color**.

color *name* [Function]

returns the color named *name*, if one has been defined by **def–color**. If none is found, returns **nil**.

3.2.2 Creating Color Ranges

A color range is a set of colors. Currently, a color range is defined as a piecewise linear wash between a set of specified RGB triples or colors. A color range is defined independently of how many actual slots it will take up in a color map.

See Section 3.2.4 for examples.

create–linear–color–range *name space* **&key :knots :interval–weights** [Function]

creates and returns an object of type **color–range**, a range of color values linearly interpolated between specified knot points.

name must be a keyword, like **:shades–of–grey**. *name* can be used to symbolically refer to the color range whenit is used by other functions.

space specifies the color space for defining the range. Currently, the only legal *space* value is **:rgb**.

The **:knots** argument must be provided, and must be a list of at least two elements. Each element must be a list or vector of three floating-point numbers between 0.0 and 1.0 inclusive. The **:knots** specify the color values to be interpolated in order to fill the color range.

:interval–weights specify relatively how many color slots of the color map are to be devoted to each interval between two knot points. The **:interval–weights** argument must be a list of positive numbers, of length one less than the length of the **:knots** list. It defaults to a list of 1.0's.

def–linear–color–range *name* **&key :knots :interval–weights** [Macro]

creates a linear color range by calling function **create–linear–color–range**, and pushes it onto the list ***defined–color–ranges***, replacing any previously defined color ranges having the same name.

Two linear color ranges are pre-defined:

:gray	a linear wash from black to white
:rainbow	a linear wash through the colors of the spectrum, violet to red

defined–color–ranges [Variable]

is a list of all color ranges defined by **def–linear–color–range**.

color–range *name* [Function]

returns a color-range named *name* defined by **def–linear–color–range**. If none is found, returns **nil**.

describe–color–range *color–range* **&key :stream** [Function]

prints out in a pretty format information about the *color–range* object.

3.2.3 Creating Color Maps

The following functions allow you to define a color map in terms of color ranges and individual colors, both of which can be referred to symbolically.

See Section 3.2.4 for examples.

```
create-color-map  name  ranges  &key  :relative-range-lengths     [Function]
                                      :colors
                                      :background
                                      :foreground
                                      :t-color
                                      :nil-color
                                      :size
```

creates and returns an object of type **color-map**.

name must be a keyword like **:my-color-map**. *name* can be used to symbolically refer to the the color map when it is used by other functions.

ranges is a list, each element of which is either a **color-range** object or the name of a **color-range** object. The function **display-image** will treat the first color range as the default color range.

:relative-range-lengths is a list of positive numbers of length equal to the length of the *ranges* argument. It specifies the relative number of slots to be allocated to each range. It defaults to a list of all 1.0's.

:colors.is a list, possibly null, of **color** objects. It defaults to **nil**.

The arguments **:background, :foreground, :t-color,** and **:nil-color,** if provided, must be the *names* of color objects defined in the **:colors** argument. The *names* of color objects are keyword symbols. **:background,** which defaults to black, is mapped to slot 0, and **:foreground,** which defaults to white, is mapped to the highest slot in the color map (typically 255). The arguments **:nil-color** and **:t-color** define default colors for displaying boolean values with function **display-image. :nil-color** and **:t-color** default to the same color as **:background** and: **foreground**, respectively.

The argument **:size** is the number of color slots in the color map. Slot 0 is assigned to the **:background** color. The highest slot is assigned to the **:foreground** color. Successive slots, starting with slot 1, are filled with colors provided by the **:colors** argument. All remaining slots are divided among the color ranges specified, according to the weights specified by **:relative-range-lengths.** Colors are then assigned to the slots belonging to each color range according to the definition of the **color-range** object.

def–color–map *name* *ranges* **&key** **:relative–range–lengths** [Macro]
:colors
:background
:foreground
:size

creates a color map by calling **create–color–map**, and pushes it onto the list ***defined–color–maps***, replacing any previously defined color map having the same name.

defined–color–maps [Variable]

is a list of all color maps defined by **def–color–map**.

color–map *name* [Function]

returns a color map named *name* defined by **def–color–map**. If none is found, returns **nil**.

3.2.4 Example

This is how the color map **:gray** is defined:

```
(def-linear-color-range :gray
  :rgb
  :knots
  '((0.0 0.0 0.0)
    (1.0 1.0 1.0)))


(defparameter *standard-colors*
         (list
      (create-color :dim-blue :rgb '(0.00 0.00 0.30))
      (create-color :black :rgb '(0.00 0.00 0.00))
      (create-color :gray :rgb '(0.50 0.50 0.50))
      (create-color :white :rgb '(1.00 1.00 1.00))
      (create-color :red :rgb '(0.97 0.0 0.15))
      (create-color :orange :rgb '(0.98 0.79 0.05))
      (create-color :yellow :rgb '(0.96 0.95 0.04))
      (create-color :green :rgb '(0.25 0.72 0.13))
      (create-color :blue :rgb '(0.20 0.33 0.74))
      (create-color :violet :rgb '(0.55 0.00 0.59))
      ))
```

```
(def-color-map :gray (list :gray)
  :colors *standard-colors*
  :background :dim-blue
  :foreground :white
  :nil-color :black
  :t-color :red
  )
```

3.2.5 Selecting a Color Map

set–color–map *color–map* **&key :display–window** [Function]

changes the color map for a display window. The **:display–window** argument defaults to the current display window.

When the color map for a display window is changed, the colors on the display change immediately. Under X Windows, however, the mouse must be positioned over the window for the window's color map to be activated (see Section 3.5).

The following pseudo-color color maps are defined:

:gray	includes color range **:gray**
:rainbow	includes color range **:rainbow**
:gray–and–rainbow	includes color ranges **:gray** and **:rainbow**

All three color maps include the following overlay colors: **:red, :orange, :yellow, :green, :blue, :violet, :black, :white, :gray,** and **:dim–blue.**

current–color–map [Function]

returns the color map associated with the current display window, or **nil**, if no display window is current. See also function **display–window–color–map**, in Section 2.2.

3.2.6 Getting Information about a Color Map

describe–color–map &optional *color–map stream* [Function]

prints out in a pretty format information about *color–map,* which defaults to the current color map. Returns **nil**.

describe–color–map–color–arrays *color–map* **&key** **:start** [Function]
:end
:stream

prints out in tabular format the RGB values of each color slot in the *color–map*.

color–map–size *color–map* [Function]

returns the size of the *color–map* (the number of slots in its color table).

color–map–color–ranges &optional *color–map* [Function]

returns the names of the color ranges used to define *color–map*, which defaults to the current color map. Function **color–map–color–ranges** returns **nil** if there is no color map.

```
(color-map-color-ranges (color-map :gray-and-rainbow))
-> (:GRAY :RAINBOW)
```

color–map–color–names &optional *color–map* [Function]

returns the names of the colors used to define *color–map*, which defaults to the current color map. Since more than one name can be associated with a **color**, only the first name is returned. Function **color–map–color–names** returns **nil** if there is no color map.

```
(color-map-color-names (color-map :gray-and-rainbow))
-> (:VIOLET :BLUE :GREEN :YELLOW :ORANGE :RED :GRAY :BLACK
     :WHITE :DIM-BLUE)
```

color–map–color–aliases &optional *color–map* [Function]

returns a nested list of all the names of each color used to define *color–map*, which defaults to the current color map. Each element of the list is either a single name or a list of two or more names of each color in the color map. Function **color–map–color–aliases** returns **nil** if there is no color map.

```
(color-map-color-aliases (color-map :gray-and-rainbow))
-> ((:VIOLET) (:BLUE) (:GREEN) (:YELLOW) (:ORANGE) (:T-COLOR :RED)
     (:GRAY) (:NIL-COLOR :BLACK) (:FOREGROUND :WHITE)
     (:BACKGROUND :DIM-BLUE))
```

color–name–color *color–name* **&optional** *color–map* [Function]

returns the **color** named *color–name* of *color–map*, which defaults to the current color map. Function **color–name–color** returns **nil** if *color–name* is not present *color–map*, or if there is no color map.

color–slot *color–name* **&optional** *color–map* [Function]

returns the index into the color table, *color–map*, which contains the color referenced by *color–name*. The argument *color–map* defaults to the current color map. Function **color–slot** returns **nil** if *color–name* is not present in *color–map* or if there is no color map.

color–range–slots *color–range–name* **&optional** *color–map* [Function]

returns, as a list, the minimum index and maximum index of the interval of color slots that are spanned by the color range referenced by *color–range–name*. The argument *color–map* defaults to current color map. Function **color–range–slots** returns **nil** if *color–range–name* is not present in *color–map* or if there is no color map.

color–map–index–value *index* **&optional** *color–map* [Function]

returns a three-element list of floats between 0.0 and 1.0, representing the red, green, and blue intensities stored in the *color–map* at slot *index*. Function **color–map–index–value** returns **nil** if there is no color map.

3.3 Device-Level Color Map Interface

The device-level color map interface lets you specify each slot (entry) of the color map in terms of its red, green, and blue values. These functions are useful if the calling program manages its own color maps.

If color maps are managed at this level, the symbolic color maps should be disabled by evaluating the expression **(set–color–map nil)**. Also, the *color* and *color–range* arguments to function **display–image** cannot be specified. Instead, you can use function **write–display** or call **display–image** with argument *rescale–p* set to **nil**.

3.3.1 Getting Information about Device Color Maps

display–device–has–color–map–p [Function]

returns **t** if the current display device supports color maps, **nil** otherwise.

The following functions can be called only when the current display device supports color maps (i.e., function **display–device–has–color–map–p** returns **t**).

display–device–color–map–arrays–size [Function]

returns the length of the color map arrays of the current display device.

3.3.2 Setting the Device-Level Color Map

These functions can only be called when the current display device supports color maps (i.e., function **display–device–has–color–map–p** returns **t**).

set–display–device–color–map–arrays *red–array* [Function]
green–array
blue–array

sets the color map arrays of the current display device. The arguments *red–array*, *green–array*, and *blue–array* must each be arrays of type single float and be of length **display–window–color–map–size**. The arrays must contain values between 0.0 and 1.0, inclusive.

set–display–device–color–map–slot *index red green blue* [Function]

sets one slot of the color map arrays of the current display device. *red*, *green*, and *blue* must be floating-point numbers between 0.0 and 1.0, inclusive. *index* must be an integer greater than or equal to 0 and less than **display–window–color–map–size**.

3.3.3 Reading Contents of Device-Level Color Maps

display–device–color–map–arrays [Function]

returns the color map arrays of current display device as three values: *red–array*, *green–array*, and *blue–array*. Each array is of length **display–device–color–map–size** and of element type **single–float**.

display–device–color–map–slot *index* [Function]

Returns the *red*, *green*, and *blue* intensity values, as three values, of color slot *index*. *index* must be greater than or equal to 0 and less than **display–window–color–map–size**.

3.4 Using 24-bit Color Maps

On 24-bit devices, color maps are interpreted differently than they are on 8-bit devices. On a 24-bit device, the three bytes of a 24-bit image value index the red, green, and blue color map arrays independently.

For convenience, the symbolic color map **:rgb** is defined for displaying RGB images on 24-bit devices. Each red, green, and blue map is a linear ramp. No overlay, background, or foreground colors are defined.

Since the symbolic color map interface is designed for pseudo-color (8-bit) color maps, other 24-bit color maps are best specified using using the device-level interface.

3.5 Color Maps on X Windows

Under X Windows, the screen's color map is determined by the window under the mouse cursor. X display windows use their own color map, which is not generally compatible with the color maps used by other applications. Therefore, if displaying to an X window yields peculiar results, first check that the mouse is indeed over the display window. To minimize the difference between an X display window's color map and those of other applications, you may wish to try changing the color map's background color and order of overlay colors.

Chapter 4

Displays

Displays are rectangular regions of a display window onto which images are displayed. Displays can be positioned either by the user or automatically by a program that keeps track of empty space on the display window.

4.1 Ways to Use Displays

The function **display-image** (and its variant, **display-image!!**) provide a high-level interface for displaying images. There are three different ways to display images using the high-level interface:

1. To display images in predetermined positions, a program can create its own displays and use them over and over. This mode is especially useful for implementing demos that display output in a predetermined format. In this case, your program keeps a pointer to the display and passes it to the function **display-image**, as in the following example:

   ```
   (setq top-display
    (make-display :size (list 256 256) :position (list 32 20)))

   (display-image my-image!! :display top-display)
   ```

2. If you wish to display an image in a predetermined position only once, you can simply specify a position argument to **display-image** as follows:

   ```
      (display-image my-image!! :position (list 32 30))
   ```

   In this case, no display object gets created.

3. To let the system automatically position an image, you simply call function **display-image** without specifying a *display* or *position* argument:

```
        (display-image my-image!!)
```

In this case, the image appears on the next free space on the window—either beside the last display or on a new row. Function **display-image** actually creates a display for this purpose, which is accessible using function **last-display**, and which can be "undone" by function **clear-display**. The auto-positioning feature is especially useful for debugging, since it automatically places images side by side.

Function **display-image** is documented in Section 4.4.

4.2 Creating Displays

Displays are created in one of two ways: by the function **create-display** and by the function **display-image** (when neither of the arguments *display* or *position* is specified).

Function **create-display** is used when you wish to explicitly allocate and position your own displays. It is useful for demonstration programs that have a fixed-format display window for output, where each display is used over and over.

Function **display-image** is documented in Section 4.4.

create-display *size position* **&key :display-window :label** [Function]

returns a **display** object. The display object is associated with the **:display-window** specified, which defaults to the current display window. The display object defines a certain region of the display window, delineated by *size* and *position*. *position* specifies the upper left corner of the display (*x y*), and *size* specifies the extent (*x-extent y-extent*).

An error is signalled if the *size* and *position* arguments specify an area that is not totally within the boundaries of the display window. The arguments *size* and *position* must both be two-element lists of non-negative integers.

The **:label** argument, if provided, must be a string. Currently, this argument is ignored.

4.3 Getting Information about a Display

describe–display *display* **&optional** *stream* [Function]

prints out in a pretty format information about *display*.

display–position *display* [Function]

returns the position of *display* with respect to its display window. The position is returned as a two-element (*x y*) list of integers.

display–size *display* [Function]

returns the size of *display*, in pixels. The size is returned as a two-element list of integers of the form (*width height*).

display–display–window *display* [Function]

returns the display window object to which *display* belongs.

display–label *display* [Function]

returns the label associated with *display*.

valid–display–p *object* [Function]

returns **t** if *object* is a display associated with an allocated display window.

4.4 Displaying an Image

display–image [*Defun]
display–image!! [*Defun]

```
        image &key :rescale–p   :color–range  :t–color   :nil–color
                   :image–min   :image–max    :display   :display–window
                   :position    :image–start  :size      :overlay
                   :label       :zoom
```

displays an image on a portion of a display window. The image is rescaled by default and is dithered for display on monochrome windows.

image is a 2-dimensional pvar of numeric or boolean type. Its geometry can be in either grid or framebuffer order. Framebuffer-ordered geometries are explained in the *Graphics Programming Release Notes* for Version 5.2.

display–image returns **nil**. You can call **display–image** at top level to display an image without consuming stack space. **display–image!!** returns *image*. It is especially useful for displaying a pvar in the middle of debugging a program.

When the operations **display–image** and **display–image!!** are called with only the *image* argument specified, they automatically rescale, position, and, possibly, dither the entire image. The keyword arguments allow you to manually specify scaling, colors, position, and sizing of the image, and also allow you to overlay additional images.

4.4.1 Specifying How Much of the Image to Display

The entire image is displayed by default, but **:image–start** and **:size** can be used to display only a portion of the image.

Arguments **:image–start** plus **:size**, if specified, must not exceed the dimensions of *image*.

If **:display** is specified, the portion of *image* displayed is the maximum amount possible, given the constraints imposed by the size of **:display** and by **:image–start** and **:size**. Portions of the image that extend beyond the boundaries of the display are clipped.

4.4.2 Specifying Where to Display the Image

If **:display** is specified, **:position** must not be specified, because **:position** defaults from **:display**. Otherwise, **display–image** creates a display for displaying the image.

If **:position** is specified, it must be a list (x y) of two non-negative integers specifying the x and y coordinates, where the upper left corner of the image is displayed.

If neither **:position** nor **:display** is specified, **:position** defaults to the next "free area of screen space" large enough to display the desired portion of the image. This will be either beside the most recent automatically positioned image, or on a new row below it. If the next empty space is not large enough to display *image*, the display window is cleared and the image is displayed at the top left. If the specified portion of *image* is still too large to be displayed on the display window, an error is signalled. **:display–window** specifies the window for displaying the image, which defaults to the current display window, or to that of

:display, if **:display** is specified. If **:display-window** and **:display** are specified, the **:display** must belong to **:display-window.**

4.4.3 Specifying the Colors for Displaying the Image

Numeric Pvars

If *image* is a numeric pvar, then the keywords **:rescale-p, :color-range, :image-min,** and **:image-max** are meaningful.

If: **rescale-p** is **t**, the default, then *image* is assumed to be a monochrome or pseudo-color image, and it is rescaled as follows.

:color-range specifies a color range for displaying the image. It is meaningful only if *image* is numeric. If specified, **:color-range** must be of the values returned by the function **color-map-color-ranges**; otherwise, it defaults to the first color range of the current color map. If the default color map **:gray-and-rainbow** is used, then **:color-range** can be either **:gray** (the default color range), which displays the image using gray levels, or **:rainbow**, which displays the image using spectral colors.

The values in the image within the viewing area (specified by **:image-start** and **:size**) are mapped to indices in the **:color-range** such that **:image-min** (which defaults to the minimum image value in the area specified) is mapped to the minimum index and **:image-max** (which defaults to the maximum image value in the area specified) is mapped to the maximum index. Any values outside this range are clipped to either **:image-min** or **:image-max.**

If **:rescale-p** is **nil**, then the image is not rescaled, and keywords **:color-range, :color, :image-min,** and **:image-max** are ignored. In this case *image* must be coercible to a unsigned pvar of element length **display-window-bits-per-pixel** for color-mapped screens, and to element length 8 for 1-bit screens. Figure 3 illustrates the stages for displaying monochrome and pseudo-color images on color-mapped and black-and-white windows.

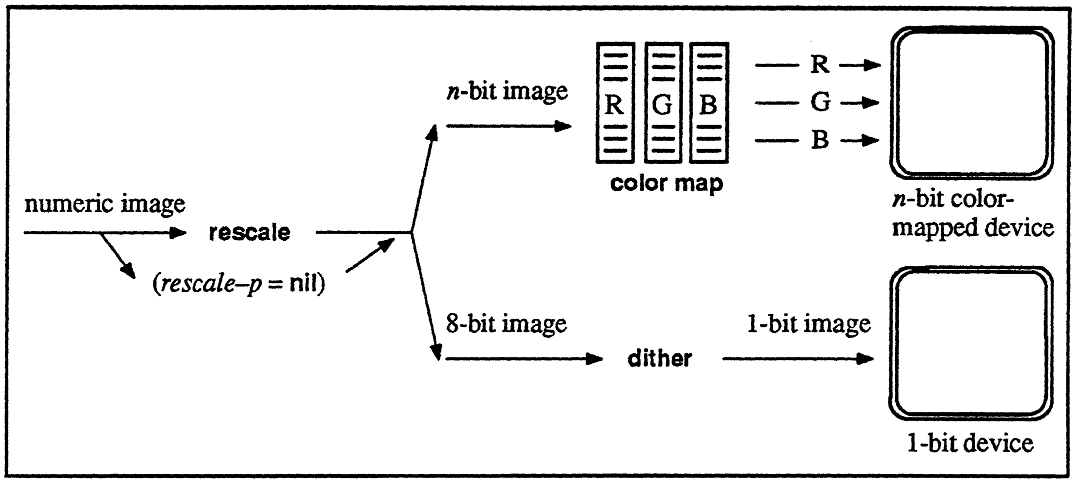

Figure 3. Display stages for monochrome and pseudo-color images on color-mapped and black-and-white windows

To display a 24-bit RGB image on a 24-bit screen, set the color map to **:rgb** and call **display–image** with **:rescale–p** set to **nil**. Figure 4 illustrates the stages for displaying a 24-bit image on a 24-bit window.

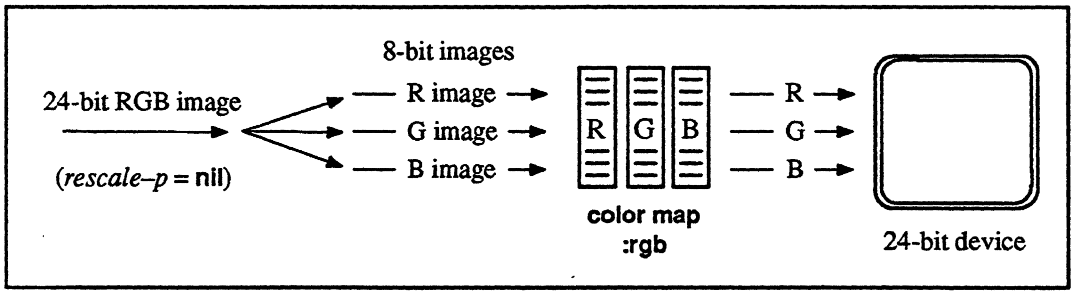

Figure 4. Display stages for 24-bit images on a 24-bit device

Boolean Pvars

If *image* is a boolean pvar, then the keywords **:t–color** and **:nil–color** are meaningful.

:t–color and **:nil–color** specify the colors for displaying **t** and **nil** values of *image*. **:t–color** and **:nil–color**, if specified, must be the names of colors in the current color map. They default to the color map's **:t–color** and **:nil–color**, respectively (which are **:white** and **:black** in the default color map.) **:t–color** and **:nil–color** are meaningful only if *image* is a boolean pvar and if the current color window supports color maps. On monochrome devices, **nil** always maps to 0 (black) and **t** maps to 1 (white).

4.4.4 Specifying Multiple Images to Overlay

The keyword argument *overlay* optionally specifies a list of images to overlay or superimpose onto the original image, *image*. *overlay* is either an overlay specification or a list of overlay specifications, with each specification of the form

```
(overlay-mask &key   :overlay-image   :rescale-p   :color-range
                     :t-color  :nil-color  :image-min  :image-max)
```

overlay–mask is an a boolean pvar that specifies where the image to overlay, **:overlay–image**, is to be drawn. **:overlay–image** is either a boolean or numeric pvar. It defaults to **t!!**. When **:overlay–mask** is t within the specified image area, **:image–min** and **:image–max** default to the minimum and maximum value of **:overlay–image**.

The remaining arguments mean the same thing and follow the same restrictions as the corresponding top-level arguments to **display–image**, but in this case apply to the overlay image. Every **:overlay–mask** and **:overlay–image** must be in the same VP set as *image*.

Wherever *overlay–mask* is **t**, the overlay image is displayed instead of the original image. Where two or more overlays satisfy this condition, the first overlay satisfying this condition is displayed. Where no overlays satisfy this condition, the original image is displayed.

4.4.5 Specifying Other Features

label is a string to be drawn under the image. It defaults to an empty string or to the label of *display*, if *display* is specified.

NOTE: Currently, the drawing of labels is not implemented.

The argument *zoom*, which defaults to 1, is the factor by which to zoom each pixel. If specified, *zoom* must be a positive integer or a list of two positive integers (*zoom–x zoom–y*). If *zoom* is a single integer, it specifies the zoom factor to be applied to both the X and Y dimensions of the image. If *zoom* is a list of two integers, the first is applied to the X dimension and the second to the Y dimension.

4.4.6 Example

```
(in-package '*lisp :use '(*graphics))


(*cold-boot :initial-dimensions (list 256 64))
; let this be the image size


(create-display-window)
   ; choose a color device, if available.


(set-color-map :gray-and-rainbow)
   ; this is the default color map anyway


(*defvar ramp!! (self-address-grid!! (!! 0)))
  ; define a numeric pvar


(display-image ramp!!)
  ; displays ramp using gray levels


(display-image ramp!! :size (list 128 64))
  ; display only the left half of the image


(display-image ramp!! :color-range :rainbow)
  ; displays ramp using rainbow colors


(*defvar random!! (zerop!! (random!! (!! 20))))
  ; define a random boolean pvar


(display-image random!! :t-color :red :nil-color :black)
  ; display random in red and black


(display-image random! :size (list 40 10) :zoom 6)
        ;; display zoomed in section of random


(display-image ramp!! :overlay (list random!! :t-color :green))
  ; display ramp in gray, overlaying t values of random in green


(display-image ramp!! :image-max (* 2 (*max ramp!!))
           :overlay (list random!! :t-color :green))
 ; display same thing, but use only lower half of gray levels for ramp


(*defvar overlay-mask!! (>!! (self-address-grid!! (!! 0))
                (!! (round (dimension-size 0) 2))))
  ; overlay mask is t for the right half of the image
```

```
(display-image overlay-mask!!)

(display-image ramp!! :overlay '((,overlay-mask!! :overlay-image ,random!!)))
  ; superimpose random over ramp where overlay-mask is t.
  ; note that random is drawn using default t-color and nil-color.
```

4.5 Clearing Displays

last–display [Function]

returns the last *display* object automatically created by function **display–image**. This function is useful for redisplaying an image over the last display used:

```
(display-image wrong-image!!)
(display-image right-image!! :display (last-display))
```

clear–display &optional *display* [Function]

clears the contents of *display*.

If display is not specified, it "undoes" the last display by clearing the display returned by **last–display**, and by freeing its space for the next image automatically positioned by **display–image**. Thus, the example above is equivalent to the following:

```
(display-image wrong-image!!)
(clear-display)
(display-image right-image!!)
```

4.6 Reading a Pvar from a Display

read–display &key :display :result–pvar :pvar–start
:display–start :size [Function]

reads data from a device represented by the display window to which **:display** belongs into **:result–pvar**, if specified, or a newly created pvar otherwise.

Returns **:result–pvar** or the newly created pvar.

Note that this function, in general, is not the inverse of function **display–image**, since **display–image** may rescale the image, but **read–display** does not.

The **:display** from which data is read defaults to the value of the function **last–display**.

:result–pvar, if specified, must be an unsigned pvar of length greater than or equal to **display–window–bits–per–pixel**. It must be a 2-dimensional pvar, not necessarily in the current VP set, in either grid or framebuffer order.

:pvar–start is a list (*x y*) that specifies the location where the data is placed within the resulting pvar, and defaults to (0 0). **:display–start** and **:size** are lists (*x y*) that specify the area of the **:display** from which data is read. **:display–start** defaults to the position (0 0). **:size** defaults to "as much as possible," given the constraints specified by the dimensions of **:display** and **:result–pvar**, and the arguments **:display–start** and **:result–start**.

4.7 Hardware Panning and Zooming of Displays

These functions allow you to closely inspect a display on a device that supports hardware panning and zooming. The CM Framebuffer and some Symbolics color screens have this capability.

zoom–display &optional *display* [Function]

pans and zooms the device so that *display* takes up as much of the screen as possible, if the hardware supports panning and zooming; otherwise does nothing. Returns **t** or **nil** to indicate whether the operation was performed. The *display* argument defaults to **last–display**.

unzoom–display [Function]

pans and zooms the current screen to its initial state. Returns **t** or **nil** to indicate whether the current device actually supports hardware panning and zooming.

Chapter 5

Rendering Primitives

None of these routines actually display data on a display device. They all render data into a separate pvar (an image-buffer-pvar for 2d, or a pvar within a Z buffer, for 3d). Data is then displayed to display devices using display functions such as **write–display–window** and **display–image.**

5.1 Two-Dimensional Rendering Functions

***draw–image** *image–buffer–pvar source–color–pvar* [*Defun]
image–start source–start size **&key :combiner**

copies a specified region *source–color–pvar* to a specified region of *image–buffer–pvar.* The regions are specified using *image–start, source–start,* and *size,* which must all be two-element lists of integers.

The **:combiner** argument dictates how the *source–color–pvar* data is combined with data already existing in *image–buffer–pvar.* The **:combiner**, which defaults to **:overwrite**, can be specified as one of the following: **:default, :overwrite, :logior, :logand, :logxor, :u–add, :s–add, :u–min, :s–min, :u–max, :s–max.**

***draw–lines–2d** *image–buffer–pvar x–start–pvar y–start–pvar* [*Defun]
x–end–pvar y–end–pvar source–color–pvar
&key :combiner :clip–p : draw–end–point–p

defines lines using start and end X and Y coordinates in *x–start–pvar, y–start–pvar, x–end–pvar,* and *y–end–pvar.* Each active processor defines a line. Each line has a color value, defined by *source–color–pvar.*

The color value for each line is written into *image-buffer-pvar* at the processors (pixels) that constitute the line.

The argument *image-buffer-pvar* must be a 2-dimensional pvar.

The combiner argument is as defined with ***draw-image** and defaults to **:overwrite**.

The argument **:clip-p** specifies what action to take if lines extend beyond the limits of the *image-buffer-pvar*. If **:clip-p** is **t** (the default), such lines are clipped; if **:clips-p** is **nil**, an error is signalled.

***new-draw-lines-2d** *image-buffer-pvar x-start-pvar y-start-pvar x-end-pvar y-end-pvar color-pvar* **&key (combiner :overwrite) (:clip-p t)** [*Defun]

***new-draw-lines-2d** is a replacement for the ***draw-lines-2d** function, which tended to run out of memory. ***new-draw-lines-2d** is less likely to run out of CM memory.

***new-draw-lines-2d** modifies *image-buffer-pvar* such that *image-buffer-pvar* contains the values of *color-pvar* in processors representing the points of the lines defined by *x-start-pvar*, *y-start-pvar*, *x-end-pvar*, and *y-end-pvar*.

If **:clip-p** is non-**nil**, then points defined by the lines outside the region defined by *image-buffer-pvar* are discarded. If **:clip-p** is **nil**, and any point in any line is outside the region defined by *image-buffer-pvar*, an error is signalled.

The **:combiner** argument is used to combine color values destined for the same processor/pixel in *image-buffer-pvar*. Its legal values are the same as those for ***draw-points-2d.**

x-start-pvar, *y-start-pvar*, *x-end-pvar*, and *y-end-pvar* must be vector pvars of type single-float. However, all floating-point values are rounded towards minus infinity (**floor**'ed) before any computation on them is done. Thus, a point stored in *x-start-pvar* and *y-start-pvar* as #(0.3 0.9) actually represents the point #(0 0) in *image-buffer-pvar* space. Therefore it is not possible to specify that a line starts or ends "between pixels."

draw-line-2d *image-buffer-pvar x-start y-start x-end y-end color* **&key :combiner :clip-p :draw-end-point-p** [Function]

draws a single line into *image-buffer-pvar*. This is the scalar equivalent of the function ***draw-lines-2d.**

***draw–points–2d** *image–buffer–pvar x–pvar y–pvar source–color–pvar* **&key :combiner :clip–p :draw–end–point–p** [*Defun]

draws points using the X and Y coordinates in *x–pvar* and *y–pvar.* Each active processor defines a point. Each point has the color value specified by *source–color–pvar.*

The color value for each point is written into *image–buffer–pvar* at the processor (pixel) defined by X and Y. *image–buffer–pvar* must be a 2-dimensional pvar.

The **:combiner** and **:clip–p** arguments are as defined and default as in the function ***draw–lines–2d.**

draw–point–2d *image–buffer–pvar x y color* **&key :combiner** [Function]

This is the scalar analog of ***draw–points–2d.** Draws a single point with value *color* into the *image–buffer–pvar* at grid address (*x y*).

polygon–fill *image–buffer–pvar x–vector–pvar y–vector–pvar color–pvar* **&key :number–of–vertices :combiner :overwrite :clip–p :edge–color–pvar** [Function]

modifies *image–buffer–pvar* such that *image–buffer–pvar* contains the values of *color–pvar* in processors representing the interiors of the polygons defined by *x–vector–pvar* and *y–vector–pvar.*

If **:edge–color–pvar** is **nil** (the default), then the processors representing the edges of the polygons also contain the values of *color–pvar*; otherwise they contain the colors specified by **:edge–color–pvar.**

The polygons are defined in terms of their vertices. The coordinates can define a polygon in either clockwise or counterclockwise order.

Polygons of different numbers of sides can be drawn in different processors. If **:number–of–vertices** is **nil** (the default), or is not provided, it is assumed that each polygon is as big as the length of *x–vector* and *y–vector* (which must be the same size). If **:number–of–vertices** is provided and is a pvar, then it dictates the number of sides in the polygon in that processor. It is an error if **:number–of–vertices** exceeds the length of *x–vector* in any processor. **:number–of–vertices** may be a scalar integer, in which case all polygons are assumed to have that many sides.

If **:clip–p** is non-**nil** (it defaults to **t**), then polygon points outside the region defined by *image–buffer–pvar* are discarded. If **:clip–p** is **nil**, and any point in any polygon is outside the region defined by *image–buffer–pvar*, an error is signalled.

The **:combiner** argument is used to combine color values destined for the same processor/ pixel in *image–buffer–pvar*. Its legal values are the same as those for ***draw–points–2d** and it defaults to **:overwrite**.

image–buffer–pvar must be a fixed-size pvar large enough to contain values from **:edge–color–pvar** and *color–pvar*. An error is signalled if *image–buffer–pvar* is a mutable pvar.

x–vector–pvar and *y–vector–pvar* must be vector pvars of element type single-float. However, all floating-point values are rounded towards minus infinity (**floor**'ed) before any computation on them is done. Thus, a point stored in *x–vector–pvar* and *y–vector–pvar* as #(0.3 0.9) actually represents the point #(0 0) in *image–buffer–pvar* space. Therefore it is not possible to specify that a vertex is located "between pixels."

5.2 Three-Dimensional Rendering Functions

***draw–points–3d** *z–buffer* *vector3–pvar* *color–pvar* [*Defun]
&optional *:clip–p*

defines points using X, Y, and Z coordinates stored in the zeroth, first, and second elements of the vector pvar *vector3–pvar*.

The X and Y coordinates specify a processor (pixel) in *z–buffer*. The value of *color–pvar* is written into the Z-buffer at that location. If more than one color value is to be written at a single location, the color value with the smallest associated Z coordinate is used.

z–buffer must be a Z-buffer structure created previously with **create–z–buffer**.

:clip–p determines whether coordinates defining points outside the Z-buffer space are ignored (the default, **t**) or signal an error.

draw–point–3d *z–buffer* *vector3* *color* [Function]
&optional *clip–p*

This is the scalar analog of ***draw–points–3d**. A single color value, *color*, is written into *z–buffer* at the coordinates specified by the three values in the vector *vector3*.

Chapter 6

Z-Buffer Functions

Z-buffers are a type of data structure used to perform hidden surface removal. For each pixel in an image, a Z-buffer records the most recent color assigned to that pixel, along with a measure of the Z distance of the surface that made that color assignment. As surfaces are added to the image, their distances are compared with the value stored in the Z-buffer for each point. If a surface is closer than the most recently stored Z value, its color overrides any previous color assignment, and the Z distance is updated; otherwise, the surface's color at that point is ignored.

This chapter describes the Z-buffer functions provided by *Graphics.

The **create–z–buffer** function is used to create a Z-buffer data structure.

create–z–buffer *color–length* [Function]
&key **:float–type** **:initial–color–value**

creates and returns a Z-buffer object. The Z-buffer object internally consists of two pvars: an unsigned color value pvar of length *color–length*, and a floating-point Z pvar of either single- or double-float type. The color value pvar records the most recent color value assignment for a point, and the Z pvar records the "distance" of the surface that made that color assignment.

When **create–z–buffer** is called, the current VP set must be a 2-dimensional VP set. The image size of the Z-buffer is determined by the size of the current VP set.

The color value pvar is initialized to the specified **:initial–color–value** in all processors. The **:initial–color–value** argument defaults to 0.

The Z pvar is initialized to the most positive floating-point value that can be stored, given the representation. (This effectively says that the last surface seen by each pixel was at infinity; the first surface to be stored in the Z-buffer will have its color value recorded in every affected processor.)

The float type argument determines the floating-point type of the Z pvar, and must be either **:single–float** or **:double–float.**

describe–z–buffer *z–buffer* **&optional** *stream* [Function]

prints out in pretty format information about *z–buffer* to stream *stream.* The *stream* argument defaults to ***standard–output***.

z–buffer–color–length *z–buffer* [Function]

returns the *color–length* of the *z–buffer*, as specified by the **create–z–buffer** call that created it.

z–buffer–float–type *z–buffer* [Function]

returns the Z pvar type of the *z–buffer*, either **:single–float** or **:double–float**, as specified by the **create–z–buffer** call that created it.

z–buffer–image!! *z–buffer* [Function]

returns an unsigned byte pvar of length **(z–buffer–color–length z–buffer)**, which is a copy of the color value pvar in the *z–buffer*.

z–buffer–z!! *z–buffer* [Function]

returns a floating-point pvar (either single- or double-float, depending on the float type of the *z–buffer*), which is a copy of the Z pvar in the *z–buffer*.

z–buffer–p *object* [Function]

returns **t** if *object* is a Z-buffer, **nil** otherwise.

z–buffer–valid–p *z–buffer* [Function]

returns **nil** if *z–buffer* has been deallocated, **t** otherwise.

clear–z–buffer *z–buffer* [Function]

sets the color pvar of *z–buffer* to **(!! 0)**, and sets each value of the Zpvar to the most positive value possible, given the float type of *z–buffer*.

write–z–buffer–to–display–window *z–buffer* [*Defun]
&key :field–start :display–start :size

writes the color pvar of *z–buffer* to the current display window. **:field–start** and **:size**, specify the region of the Z-buffer displayed. **:field–start** defaults to the (*x y*) location (0 0), and **:size** defaults to as much as possible, given **:field–start**. **:display–start**, which defaults to the (x y) position (0 0), specifies the location on the display window where the upper left corner of the image is displayed.

delete–z–buffer *z–buffer* [Function]

deletes *z–buffer*. This deallocates the pvars associated with the Z-buffer. It is an error to use a Z-buffer object after it has been deallocated.

Chapter 7

Math Utilities

The math utilities are mainly concerned with coordinate transformations. Functions for creating and multiplying together transformation matrices are provided, and for multiplying vectors by transformation matrices.

A transformation matrix is a 4 x 4 array with floating-point values, either on the front end or per processor. Lisp **typedef** definitions are provided to ease the declaration of such objects.

degrees–to–radians!! *degrees–pvar* [Macro]

converts degrees to radians in parallel.

degrees–to–radians *degree* [Macro]

converts a scalar value in degrees to radians. This is the scalar analog of **degrees–to–radians!!**.

radians–to–degrees!! *radians–pvar* [Macro]

converts radians to degrees in parallel.

radians–to–degrees *radian* [Macro]

is the scalar analog of **radians–to–degrees!!**. It converts a scalar value in radians to degrees.

identity–matrix!! *n* **&key :element–type** [Function]

returns an *n* x *n* identity matrix pvar in each processor. The pvar type is determined by the **:element–type** argument, which defaults to a single-float pvar.

***identity–matrix** *matrix–pvar* [Function]

matrix–pvar must be an *n* x *n* array pvar. In each active processors its values are set to those of the *n* x *n* identity matrix.

identity–matrix &optional *n* [Function]

returns a front-end floating-point identity matrix, of dimensions *n* x *n*.

sf–3d–tmatrix [Deftype]

This is a Lisp **deftype**, equivalent to

```
(array single-float (4 4)).
```

It can be used to specify a single-float 3D transformation matrix.

sf–3d–tmatrix–pvar [Deftype]

This is a *Lisp **deftype**, equivalent to

```
(array-pvar single-float (4 4)).
```

It can be used to specify a single-float 3D transformation matrix. pvar.

df–3d–tmatrix [Deftype]

This is a Lisp **deftype**, equivalent to

```
(array double-float (4 4)).
```

It can be used to specify a single-float 3D transformation matrix.

df–3d–tmatrix–pvar [Deftype]

This is a *Lisp **deftype**, equivalent to

```
(array-pvar double-float (4 4)).
```

It can be used to specify a single-float 3D transformation matrix pvar.

transform–vector *vector transform–matrix* [Function]

multiplies *vector* by *transform–matrix* and returns a new vector. If *vector* is two elements long, *transform–matrix* must be 3 x 3, otherwise *vector* must be three elements long and *transform–matrix* must be 4 x 4.

***transform–vector** *dest–vector–pvar source–vector–pvar transform–matrix* [*Defun]

performs a vector-matrix multiplication operation in each processor. *source–vector–pvar* must be a two- or three-element vector. The *dest–vector–pvar* must be of the same length as *source–vector–pvar.*

If *source–vector–pvar* is two elements, then *transform–matrix* must be a 3 x 3 array or array pvar. If *source–vector–pvar* is three elements, then *transform–matrix* must be a 4 x 4 array or array pvar.

Because *transform–matrix* may be either an array or an array pvar, a pvar of coordinates can be transformed by a single transformation matrix stored on the front end without the necessity of creating a transformation matrix pvar inside the CM.

The standard 2D or 3D graphics transformation is performed on *source–vector–pvar* and the result is stored in *dest–vector–pvar.*

create–transformation–matrix–3d &rest *transform–specs* [Function]

creates and returns a 4 x 4 floating-point matrix. If no *transform–specs* are provided, it returns an identity matrix.

transform–specs is a sequence of keyword value pairs. The keywords are one of the following:

Keyword	**Meaning**
:rx	rotation-in-x
:ry	rotation-in-y
:rz	rotation-in-z
:sx	scale-in-x
:sy	scale-in-y
:sz	scale-in-z
:tx	translation-in-x
:ty	translation-in-y
:tz	translation-in-z

The matrix returned is a transformation matrix that embodies all the rotations, scaling, and translations specified, in the order in which they were specified.

Example:

```
(create-transformation-matrix-3d :rx 0.5 :sy 2.0 :tz 2.0 :rz 0.2)
```

creates a transformation matrix that, when applied to a vector using ***transform-vector**, will rotate the vector 0.5 radians, then scale it by 2.0 along the Y dimension, then translate it 2.0 units along Z, and finally rotate it 0.2 radians about the Z axis.

***create-transformation-matrix-3d** *tmatrix-pvar* **&rest** *transformation-specs* [*Defun]

destructively modifies *tmatrix-pvar* (which must be a 4 x 4 floating-point matrix pvar) to contain, in each processor, a transformation matrix specified according to *transformation-specs*.

The *transformation-specs* argument is as documented in **create-transformation-matrix-3d**, except that the values may either be float scalar or float pvars.

This allows you to create a potentially different transformation matrix in each processor.

***tmatrix-multiply** *dest source1 source2* **&rest** *sources* [*Defun]

All the arguments must be transformation matrix pvars. The matrices are multiplied in order on the right. That is,

```
dest <- (source1 matmul source2)
dest <- (dest matmul (first sources))
dest <- (dest matmul (second sources)) ...
```

***x-rotation-matrix-3d** *tmatrix-pvar rx* [*Defun]

destructively modifies *tmatrix-pvar* to be a transformation matrix that represents a rotation of *rx* radians around the X axis. *rx* must be a pvar. *tmatrix-pvar* must be a transformation-matrix pvar.

***y–rotation–matrix–3d** *tmatrix–pvar ry* [*Defun]

destructively modifies *tmatrix–pvar* to be a transformation matrix that represents a rotation of *ry* radians around the Y axis. *ry* must be a pvar. *tmatrix–pvar* must be a transformation-matrix pvar.

***z–rotation–matrix–3d** *tmatrix–pvar rz* [*Defun]

destructively modifies *tmatrix–pvar* to be a transformation matrix that represents a rotation of *rz* radians around the Z axis. *rz* must be a pvar. *tmatrix–pvar* must be a transformation-matrix pvar.

***scale–matrix–3d** *tmatrix–pvar sx sy sz* [*Defun]

destructively modifies *tmatrix–pvar* to be a transformation matrix that represents a scaling of *sx*, *sy* and *sz* units in X, Y, and Z, respectively. *sx*, *sy*, and *sz* must be pvars. *tmatrix–pvar* must be a transformation-matrix pvar.

***translation–matrix–3d** *tmatrix–pvar tx ty tz* [*Defun]

destructively modifies *tmatrix–pvar* to be a transformation matrix that represents a translation of *tx*, *ty*, and *tz* units along X, Y, and Z, respectively. *tx*, *ty*, and *tz* must be pvars. *tmatrix–pvar* must be a transformation-matrix pvar.

Index

This index lists all the *Graphics functions and variables in the **Graphics Reference Manual, Version 6.0*. References are to page numbers.

Symbols

all–display–windows, 12
***create–transformation–matrix–3d**, 50
current–display–window, 12
defined–color–maps, 22
defined–color–ranges, 20
defined–colors, 19
***draw–image**, 39
***draw–lines–2d**, 39
***draw–points–2d**, 41
***draw–points–3d**, 42
***identity–matrix**, 48
***new–draw–lines–2d**, 40
***scale–matrix–3d**, 51
***tmatrix–multiply**, 50
***transform–vector**, 49
***translation–matrix–3d**, 51
***x–rotation–matrix–3d**, 50
***y–rotation–matrix–3d**, 51
***z–rotation–matrix–3d**, 51

C

clear–display, 37
clear–display–window, 14
clear–z–buffer, 44
color, 19
color–map, 22
color–map–color–aliases, 24
color–map–color–names, 24
color–map–color–ranges, 24
color–map–index–value, 25
color–map–size, 24
color–name–color, 25
color–range, 20
color–range–slots, 25
color–slot, 25
create–color, 18
create–color–map, 21
create–display, 30
create–display–window, 7
create–linear–color–range, 19
create–tranformation–matrix–3d, 49
create–z–buffer, 43
current–color–map, 23

D

def–color, 19
def–color–map, 22
def–linear–color–range, 20
degrees–to–radians, 47
degrees–to–radians!!, 47
delete–all–display–windows, 14
delete–display–window, 14
delete–z–buffer, 45
describe–color–map, 23
describe–color–map–color–arrays, 24
describe–color–range, 20
describe–display, 31
describe–z–buffer, 44
df–3d–tmatrix, 48
df–3d–tmatrix–pvar, 48
display–device–color–map–arrays, 27

display–device–color–map–arrays–size, 26
display–device–color–map–slot, 27
display–device–has–color–map–p, 26
display–display–window, 31
display–image, 31
display–image!!, 31
display–label, 31
display–position, 31
display–size, 31
display–window–bits–per–pixel, 11
display–window–cmsr–display, 11
display–window–color–map, 11
display–window–height, 11
display–window–width, 11
draw–line–2d, 40
draw–point–2d, 41
draw–point–3d, 42

G

get–screen–pan–and–zoom, 15

I

identity–matrix, 48
identity–matrix–!!, 47

L

last–display, 37

P

polygon–fill, 41

R

radians–to–degrees, 47
radians–to–degrees!!, 47
read–display, 37
read–display–window, 14
rescale–pvar–for–display–window!!, 13

S

screen–pan–and–zoom–p, 15
set–color–map, 23
set–display–device–color–map–arrays, 26
set–display–device–color–map–slot, 26
set–display–window, 12
set–screen–pan–and–zoom, 15
sf–3d–tmatrix, 48
sf–3d–tmatrix–pvar, 48

T

transform–vector, 49

U

unzoom–display, 38

V

valid–display–p, 31
valid–display–window–p, 11

W

with–display–window, 12
write–display–window, 12
write–z–buffer–to–display–window, 45

Z

z–buffer–color–length, 44
z–buffer–float–type, 44
z–buffer–image!!, 44
z–buffer–p, 44
z–buffer–valid–p, 44
z–buffer–z!!, 44
zoom–display, 38