OBVIUS:

Object-Based Vision and Image Understanding System

Version 2.2

©1988, 1989, 1990, 1991, 1992
Vision Science Group, Media Laboratory, MIT.

September 21, 1993

David Heeger Eero Simoncelli Eduardo-Jose Chichilnisky

Contents

1 Introduction **7**

2 Getting Started **8**

3 Organization of OBVIUS **10**

3.1 Viewables 10

Displaying Viewables 11

Modifying Viewables 11

Destroying Viewables 12

Orphans 12

History and Constructor 13

3.2 Pictures and Panes 13

Picture Slots and Default Values 14

Manipulating Pictures and Panes 15

Zoom 16

3.3 File System Interface 17

3.4 Memory Management 19

4 Viewables **21**

4.1 Images 21

Make-image 22

Iref 22

Print-values 23

Pictures of Images 23

4.2 One-d-images and Slices 23

4.3 Discrete Functions and Histograms 24

4.4 Bit-images 25

4.5 Sequences 25

4.6 Image Pairs, Complex Images, and Polar Images 25

4.7 Filters 26
3D filtering 27
Hexagonal filtering 27
4.8 Pyramids 27
4.9 Color Images 27
4.10 Viewable Matrices 28

5 Pictures **29**
5.1 Display 29
5.2 Gray Pictures 29
5.3 Dither Pictures of Images 30
5.4 Bitmaps 30
5.5 Color Pictures 31
5.6 Pasteups 31
5.7 Surface Plots 32
5.8 Contour Plots 33
5.9 Vector Fields 34
5.10 Graph Pictures 34
5.11 Polar Plots 35
5.12 Scatter Plots 36
5.13 Flipbooks 37
5.14 Overlays 38
5.15 Hardcopy 38

6 OBVIUS Functions **40**
Result Argument 40
Point Operation Macros 40
6.1 Image Utilities 41
6.2 Generating Images 41
6.3 Loading and Saving Images and Image-Sequences 44

6.4 Operations on Images 44
Numerical Point Operations 44
Geometrical Operations 46
Comparison Point Operations 47
Statistics 49
Fourier Transform 49
Filtering Operations 50
6.5 Functions for One-d-images 51
6.6 Operations on Bit-Images 51
6.7 Operations on Image-Pairs, Complex-Images, and Polar-Images 52
6.8 Operations on Discrete Functions 54
6.9 Operations on Sequences 55
6.10 Operations on Color-Images 57
6.11 Operations on Viewable-Matrices and Image-Matrices 57
6.12 Operations on Filters 59
6.13 Operations on Pyramids 60
6.14 Operations on Arrays 62
6.15 Operations on Lists 63
6.16 Operations on Matrices 63
Matrix Arithmetic 64
Functions Related to SVD and Other Decompositions 66
Matrix Utilities 67

7 Data Analysis **71**

7.1 Matrix Tools 71
7.2 Statistical and Simulation Tools 72
7.3 Curve Fitting 73

8 User Interface **76**

8.1 The Mouse 76

8.2 The Control Panel 79
8.3 Current Picture, Picture Defaults and Viewable Defaults menus 80
8.4 Current Screen Parameters 80
8.5 Global Parameters 81
8.6 Running OBVIUS on a remote host 82
8.7 Screens and Windows 82
8.8 Emacs and OBVIUS 83
8.9 Help 84

9 Hacking **85**

9.1 Writing Additional Image Processing Functions 85
The with-result macro 85
Temporary Viewables 86
Efficient Looping 87
Calling Foreign Functions 87
9.2 The Memory Manager 87
Orphans and OGC 89
Troubleshooting Memory Problems 90
9.3 Adding New Viewable Subclasses 91
9.4 Multi-Processing 95
Mouse Events 95
Repl 95
Adding Menus and Dialog Boxes 96
9.5 Display of Pictures 96
9.6 Adding a New Picture Subclass 99
9.7 Porting To Other Machines, Window Systems, or Lisp Implementations . . 101

A Installation **103**

A.1 Requirements 103
A.2 Installation Instructions 104

B Release Notes **107**

B.1 Version 2.1 Release Notes 107

B.1.1 General Changes 107

B.1.2 Specific Notes for OBVIUS Users 107

B.1.3 Notes for OBVIUS Hackers (i.e. non-exported changes) 111

B.2 Version 2.2beta Release Notes 112

B.3 Version 2.2 Release Notes 113

C Known Bugs **115**

Index **117**

Copyright 1991 by the Massachusetts Institute of Technology. All rights reserved.

Developed by David J. Heeger and Eero P. Simoncelli at the Media Laboratory, MIT, Cambridge, Massachusetts. This distribution is approved by Nicholas Negroponte, Director of the Media Laboratory, MIT.

Permission to use, copy, or modify these programs and their documentation for educational and research purposes only and without fee is hereby granted, provided that this copyright notice appears on all copies and supporting documentation. For any other uses of this software, in original or modified form, including but not limited to distribution in whole or in part, specific prior permission must be obtained from MIT. These programs shall not be used, rewritten, or adapted as the basis of a commercial software or hardware product without first obtaining appropriate licenses from MIT. MIT makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

1 Introduction

OBVIUS (Object-Based Vision and Image Understanding System) is an image-processing system based on Common Lisp and CLOS (Common Lisp Object System). The system provides a flexible interactive user interface for working with images, image-sequences, and other pictorially displayable objects. By using Lisp as its primary language, the system is able to take advantage of the interpretive lisp environment (the "listener"), object-oriented programming, and the extensibility provided by incremental compilation.

The basic functionality of OBVIUS is to present certain lisp objects to the user pictorially. These objects are refered to as "viewables". Some examples of viewables are monochrome images, color images, binary images, complex images, image sequences, image pyramids, filters and discrete functions. A "picture" is the displayable representation of a viewable. Note that a given viewable may be compatible with several different picture types. For example, a floating point image may be displayed as an eight bit grayscale picture, as a one bit dithered picture, or as a 3D surface plot. OBVIUS also provides postscript hardcopy output of pictures.

In the typical mode of interaction, the user types an expression to the lisp listener and it returns a viewable, which is automatically displayed as a picture in a window. Each window contains a circular stack of pictures. Standard stack manipulation operations are provided via mouse clicks (e.g., cycle, pop, and push). Commonly used operations such as histogram and zoom are also provided via mouse clicks.

OBVIUS provides a library of image processing routines (e.g., point operations, image statistics, convolutions, fourier transforms). All of the operations are defined on all of the viewable types. The low-level floating point operations are implemented in C for speed. OBVIUS also provides a library of functions for synthesizing images. In addition, it is straightforward to add new operations and new viewable and picture types.

OBVIUS currently runs on Sun workstations in Lucid v4.0 Common Lisp. We hope to have it running on other machines in the future (e.g., DecStations, SGI workstations, Apple Macintoshes, etc).

This manual describes the overall organizational strategy of the system, and provides a description of the user interface. Throughout, we assume the reader is familiar with Emacs, Common Lisp, and some form of object-oriented programming environment. The manual is written primarily for Sun users – users of other machines should see the Appendices for machine dependencies. Section 2 and the OBVIUS tutorial (in the file `<obv>/tutorials/obvius/obvius-tutorial.lisp`) are provided for new users to aquaint themselves with the system without getting bogged down in hairy implementational details. The middle sections 3 through 8 are a more detailed description of the system for more experienced users. Section 9 is provided for programmers ("hackers").

2 Getting Started

We assume in this section that you have installed OBVIUS on your system (if you have not yet created an OBVIUS image, instructions for doing so are given in appendix A). We also assume that you are running the X window system (version 11, release 4 or higher), or Sun's OpenWindows (version 2.0 or higher). You may use a window manager of your choice. We have used OBVIUS with "twm", "uwm", "mwm", and "olwm".

We recommend that you run OBVIUS from within the Gnu emacs editor, version 18.49 or later (Gnu emacs is avaiable for anonymous ftp from prep.ai.mit.edu). The tutorial (in the file `<obv>/tutorials/obvius/obvius-tutorial.lisp`) assumes that you are doing this. OBVIUS can, however, be run from a Unix shell.

If you are running from Emacs, you should add the following lines to your "`.emacs`" startup file:

```
(setq load-path (cons <obv>/emacs-source load-path))
(setq *cl-program* <lisp-image>)
(autoload 'run-cl "cl-shell" nil t)
(setq *obvius-program* <obvius-image>)
(autoload 'run-obvius "cl-obvius" nil t)
```

The symbol `<obv>` is used throughout this document to indicate the pathname of the primary OBVIUS directory (on our system, this is "`/usr/local/obvius`"). The symbol `<lisp-image>` refers to the pathname for a Lucid 4.0 image that includes CLOS and LispView. On our system, this is in "`/usr/local/bin/lispview`".

To start lisp, you type `M-x run-cl` from within Emacs. To start OBVIUS, you type `M-x run-obvius`. After several seconds, a new Emacs buffer will appear named `*lisp*` and various system loading messages will be printed into the buffer. Eventually, you will get OBVIUS's prompt: `OBVIUS>`. This buffer provides an interface to Lisp that is running as a sub-process of Emacs. As described in the next section, hitting the `cr` key in this buffer will send text to the Lisp process for evaluation, and Lisp output will be printed into this buffer.

At this point, new users go through the OBVIUS tutorial in the file `<obv>/tutorials/obvius/obvius-tutor`
The purpose is to familiarize new users with the basic functionality of OBVIUS, and with the user interface.

The remaining part of this section describes customization of your OBVIUS environment, and is unnecessary for novices. At run-time, OBVIUS looks for two startup files in your home directory. These can be used for personal customization of your environment. The first is named either "`.obvius-windows`" or "`obvius-window-init.lisp`". This file loads the appropriate window system code. The default version is in "`<obv>/obvius-window-init.lisp`".

The second startup file is called either **.obvius** or **obvius-init.lisp**. You can use

this file to create a fixed number of OBVIUS windows in predefined locations, to initialize default values for global parameters and to have code loaded automatically. The default file is in "`<obv>/obvius-init.lisp`". This file contains many (commented out) function calls that you may wish to add to your personal initialization file.

3 Organization of OBVIUS

As described in the introduction, OBVIUS is based on three types of object: viewables, pictures, and panes. This section covers these objects in more detail and also explains the memory management system. We assume that you have gone through the exercizes of the OBVIUS tutorial.

3.1 Viewables

`viewable` s are the objects that one works with in OBVIUS. In this section, we describe the general attributes of viewables. For a discussion of the particulars of each viewable subclass, see section 4. Every viewable (simple or compound) contains the following slots:

- `name` – the name of the viewable can be either a string, a symbol, or nil. If the name is bound to a symbol, there is a symbol with that print-name bound to the viewable, i.e., the symbol may be used to refer to the viewable. The name of a viewable is usually set using the standard lisp `setq` function or using the `:->` (result) keyword argument. NOTE: Using (setf slot-value) to set the name slot will cripple the ability of OBVIUS to keep track of the existing viewables. For this reason, the name slot accessor is not exported, and should not be used.

- `superiors-of` – Some viewables contain other viewables. These are known as "compound" viewables. For example, a complex image contains two images (real and imaginary parts). Viewables containing other viewables are called *compound* viewables. Each compound viewable knows how to access its "inferior" viewables, and each of the inferior viewables has a pointer to the "superior" (compound) viewable. This slot contains a list of all viewables containing the viewable as a sub-viewable. For instance, if the viewable is the real-part of a complex-image, the list will contain that complex-image.

- `display-type` – this slot contains the name of the sub-class of picture that will be used when the viewable is automatically displayed. A nil value indicates that the viewable should *not* be displayed automatically. The slot may be set using the standard setf syntax: `(setf (display-type <viewable>) '<picture-type>)`

- `pictures-of` – this slot contains a list of all of the pictures of the viewable. There may be several different types of picture of the same viewable, and there may be more than one of a given picture type. For example, there may be a grayscale picture *and* a surface plot of an image, or there may be two different grayscale pictures.

- `history` – this slot contains a history object for the viewable. See the section below for more information.

- `current` – this slot contains a number that is incremented each time a viewable is modified. It is used to keep track of whether the pictures of that viewable are up-to-date or not. There is a similar slot in the picture class.

- `info-list` – this slot contains a lisp "plist" (property list) of attributes. For example, if the mean of an image is computed, it is stored in this p-list so it does not need to be recomputed. The same is true for histograms, minimum, maximum, and other image statistics. The info-list is set to nil when the viewable is destructively modified (this is done by the `set-not-current` method).

  The following functions may be used to access the info-list of a viewable:

- **(info-list** `viewable`**)**
  Returns the info-list of an image.

- **(info-get** `viewable slot`**)**
  Returns the value of the slot named *slot* in im's info-list. May be used with setf to set the value instead of using info-set.

- **(info-set** `viewable slot value`**)**
  Sets the value of the slot named *slot* in im's info-list to have the value *value*. You can also use `(setf (info-get vbl slot) val)`.

Displaying Viewables

When a viewable is returned to the top-level lisp listener, it will be displayed automatically if the global parameter `*auto-display-viewables*` is non-nil. If a picture of the viewable already exists, it will be brought to the top of the stack of the current pane if the variable `*preserve-picture-pane*` is nil. If this variable is non-nil, then the picture will stay on the same pane, but still be brought to the top of the stack. Otherwise, a picture is created of the type defined by the display-type slot of the viewable and pushed onto the picture stack of the current pane. If you wish to explicitly create a picture of a viewable with a particular set of display parameters, use the `display` function (described in Section 3.2). Section 3.2 also describes methods for altering the parameters of pictures.

Modifying Viewables

When a viewable is destuctively modified (e.g. by passing it as the `:->` argument to a function) the method `set-not-current` is called. This method tells OBVIUS that the pictures that depend on that viewable need to be recomputed. In general, the method does nothing to the viewable, but calls itself on each of the superiors of the viewable, and on each of the pictures of the viewable. The pictures then either update themselves immediately, or indicate to the user that they are "not current" by placing a double asterisk in the title

bar (this is consistent with the double asterisk Emacs uses to indicate a buffer has been modified). The display may be updated by refreshing the pane via a mouse click (see Section 8.1). Note that modifying a viewable does *not* cause related viewables to be re-computed. For example, if a histogram is computed from an image and the image is subsequently modified, the histogram will not be recomputed.

Destroying Viewables

A viewable may be destroyed by evaluating the expression `(destroy <viewable>)` or via a mouse click (see Section 8.1). The function `purge!` destroys all viewables. When a viewable is destroyed, the following things occur:

1. Superiors of the viewable are notified. Note that some superiors will not allow their inferiors to be destroyed (e.g. a complex image will trigger a continuable error when the user tries to destroy its real part).
2. Inferiors of the viewable are notified. Typically, the inferiors just take the superior off of their `superiors-of` lists. If the global parameter `*auto-destroy-orphans*` is non-nil, the inferior is destroyed (see section on orphaned viewables below).
3. All pictures of the viewable are destroyed (i.e., the `destroy` method is called on each picture). The static space used by the pictures is freed (see section 3.4).
4. The static space of the viewable is freed (see Section 3.4).
5. If the viewable name slot contains a symbol, it is unbound in the current package of the global Lisp environment.
6. The name of the viewable is set to the keyword `:destroyed`, as an indicator to the user in case they try to re-use the viewable.

Orphans

A viewable is called an "orphan" if it is no longer accessible. As far as OBVIUS is concerned, a viewable is considered inaccessible if it has no superiors, there are no pictures of it, and there is no global symbol bound to it. When you run the OBVIUS garbage collector, `ogc`, it will recycle all orphaned viewables. Furthermore, if the global variable `*auto-destroy-orphans*` has a non-nil value, OBVIUS will attempt to destroy orphans when they are created. Thus, users of OBVIUS should understand when they are creating orphans!

There are three situations when OBVIUS can prevent the "orphan'ing" of an existing viewable::

1. The (symbol) name of the viewable is (re-)bound (using `setq` or `set`) to another value.

2. The only picture of the viewable is popped from a pane (destroyed).

3. The viewable is an inferior of a compound viewable, and that compound viewble is destroyed.

In each of these cases, if the global parameter `*auto-destroy-orphans*` is non-nil, the orphan will be automatically destroyed.

Note that this can be dangerous. For example, if a user writes code that returns an inferior viewable from within a function, but the function destroys the superior, the inferior may be destroyed *before* it is returned to top level. Note also that automatic destruction of orphans *will not* prevent the user from creating new viewables that are orphans. For example, one can create a list of images with no name and avoid dislaying them. These images will be orphans and will not be preserved by the OBVIUS garbage-collector (`ogc`), although the user will have access to them. If you want to return several images from a function, you should return them as an image-sequence, or using the Common Lisp multiple-values facility. See section 9 for a more thorough discussion of this problem.

History and Constructor

The `constructor` function produces a lisp S-expression that would evaluate to a copy of the viewable. When you save a viewable to a datfile (see section 3.3, this expression is written to the descriptor file. It is also useful as a reminder of where the viewable came from. The function `(history-sexp <viewable> &key (level 1))` allows the user to print a condensed listing of the history, substituting symbols for the viewables with symbol name slots. The `:level` parameter determines how many nested levels of history (function calls) will be displayed before the system attempts to replace them with symbols. These functions are defined in the file **viewable.lisp**.

3.2 Pictures and Panes

A `pane` is a window that contains a stack of pictures. The picture on top of the stack is the one that is displayed on the screen. Each picture is allowed to reside on the stack of only one pane. To create a new pane, the user should call the function `new-pane`. The keyword arguments :left, :bottom, :right, and :top specify the position in pixels of the lower left and upper right corners of the pane.

The system maintains a notion of the *selected* or *current* pane. In general, clicking the mouse in a pane will select that pane (see section 8.1). Whenever a viewable is automatically displayed by the system, it is displayed in the current pane.

A `picture` is a particular visual representation of a viewable.

Picture Slots and Default Values

The following slots are found in every picture type:

- **pane-of** – the pane in which this picture lives. Pictures cannot exist in isolation; they must be on a picture stack.
- **viewable** – the viewable of which this is a picture.
- **system-dependent-frob** – a window-system dependent intermediate representation. The window system knows how to rapidly "render" the "frob" into a pane. For gray pictures on X-screens, the frob is an offscreen pixmap. For graphs with retain-bitmap non-nil, it is an offscreen bitmap. But for graphs with retain-bitmap nil, it is an object containing lists of graphical objects (such as lines) to draw.
- **index** – a unique registration number identifying the picture.
- **current** – a number that is is used to keep track of whether the picture is up-to-date with its viewable (given that the viewable may have been modified since the picture was originally made). The current slot is reset each time the picture is updated. There is a similar slot for the viewable class.
- **zoom** – magnification of the picture, relative to its "natural" size (this is described in detail below).
- **overlays** – this slot contains a list of the viewables that are overlayed on this picture (see section 5.14).
- **y-offset** and **x-offset** – these slots specify the position of the picture within the pane (relative to centered position).

In addition to those listed above, each picture class has parameters that control its computation from the underlying viewable. For example gray pictures have `scale` and `pedestal` parameters that specify the transformation of the floating-point image array into a grayscale colormap. The values of these parameters may be changed using the `Current Picture` dialog from the `Parameters` menu on the contral panel, or using the `M-left` mouse click. The parameter values may also be reset using the `setp` macro, which acts on the picture on top of the current pane. The `setp` macro is described in detail below.

When a new picture is created, default values are chosen for the picture parameters. The default slot values for each picture class may be set from the `Picture Defaults` submenu of the `Parameters` menu. Typically, the default value may be a number or it may be the keyword `:auto`. If it is a number, that number will be used as an initial slot value for all pictures of this type. If the default value is not a number, then OBVIUS will choose parameter values automatically.

For details about the parameters for each picture class, see the descriptions in Section 4.

Manipulating Pictures and Panes

In general, the user should not need to directly manipulate pictures or panes. OBVIUS is designed with the intention that the user perform computations on *viewables*. There are, however, several methods provided that allow the user to alter pictures:

`*current-pane*`
: This global symbol is bound to the current pane. This symbol can be used when calling functions that take a pane as an argument.

(display `&optional display-type &key pane make-new &allow-other-keys`**)**
: This function is used to create a picture of a viewable. The optional `display-type` argument should be a symbol corresponding to the name of a picture subclass (eg. `'gray`, or `'vector-field`). The default value is t, meaning to use the `display-type` slot of the viewable. The `:pane` keyword argument specifies the pane in which to display the picture and defaults to the value of `*current-pane*`. If this is nil, a new pane is created. The `display` function first tries to find an existing picture of the desired type. If no picture of the right type exists, or if a non-nil value is passed with the keyword `:make-new` , a new picture is created with all default parameter values. Otherwise, one of the existing pictures is displayed, and updated if it is out-of-date and `*auto-update-pictures*` is non-nil. All additional keyword args (`picture-initargs`) are used as initialization arguments in creating the picture (i.e., they are passed along to `make-instance`). They typically correspond to initargs for the slots of the picture being created. For example, you can create a gray picture of an image, with the scale factor set to 1.0, and the zoom factor set to fill the pane, as follows:

```
(display <image> 'gray :scale 1.0 :zoom :auto)
```

(refresh `&optional pane`**)**
: This function re-displays the picture on top of the specified pane. The optional `pane` argument defaults to the value of `*current-pane*`. This function is typically called via a mouse click (see Section 8.1).

(next-pane)
: This function sets the current pane to be the next pane in the list of panes.

(cycle-pane `&optional pane number`**)**
: This function cycles the circular picture stack of a pane and refreshes the display. The default pane is the value of `*current-pane*`. The optional `number` argument defaults to a value of 1, and specifies how many pictures to cycle past. Negative values cycle in reverse. This function is typically called via a mouse click (see Section 8.1).

(current-viewable)
: Returns the viewable currently visible in the current pane.

(get-viewable `&optional pane`**)**

Returns an S-expression that will evaluate to the viewable currently visible on top of the pane. Optional argument `pane` defaults to the value of `*current-pane*`. This function is typically called via a mouse click (see Section 8.1).

(pop-picture `&optional pane`**)**

This function removes the picture on the top of a picture stack and destroys it. The underlying viewable, however, is left intact. The default pane is the current pane. It is usually called via a mouse click (see Section 8.1).

(setp `&rest args`**)**

It is often desirable to modify the parameters that are associated with the various picture types. The macro setp has been provided for this purpose. The syntax for setp is just like setq except that all of the symbols to be set should correspond to slot names in the picture on top of the current pane. After setting the new slot values, the picture will be recomputed using the new values and the display will be refreshed. Note that some picture slots are not settable; calling setp on these will have no effect. Picture parameters can also be reset using the `Current Picture` option from the `Parameters` menu.

(getp `&optional slot-name`**)**

This macro is complementary to setp. It takes a single argument that must be the name of a slot in the current picture and returns the value of that slot. When called with no arguments, it prints a list of the names and values of *all* of the slots of the current picture.

Zoom

Every picture class has a `zoom` parameter that controls the size of the picture in the pane. The zoom amount is relative to the "natural" size of the picture (this depends on the type of picture). The `zoom` parameter may be set to `:auto`, a number, or a list of two numbers. If it is `:auto`, then OBVIUS sizes the picture to fill the pane. If it is a list, then OBVIUS sizes the picture to have dimensions equal to that list. If `zoom` is a number, then the size of the picture will be this multiple of its natural size. For example, if a 64x64 image is displayed as a gray picture with zoom = 2, then the picture will by 128x128 pixels. If zoom = 1/2, then the picture will be 32x32 pixels. For some pictures (e.g., gray), zoom will be rounded to the nearest integer, or reciprocal of an integer. For others (e.g. graph), it can take a floating point value.

The value of the zoom parameter may be changed from the `Current Picture Parameters` menu, using the `setp` macro, or by using the mouse (see section 8.1).

3.3 File System Interface

OBVIUS uses a subset of the *datfile* image file format used by several groups at the Media Lab and in other labs at MIT. A datfile is a directory containing one or more files. One of these files, named "data", contains the data organized in a rectangular matrix with any number of dimensions. A separate file in the datfile directory, named "descriptor", contains a pseudo-LISP description of the data file. Keyword/value pairs of ASCII strings, enclosed in matching parentheses, describe the organization, data type, history and other aspects of the datfile. The advantage of this scheme is that information about the image is kept separate from the image data itself, and is human-readable and easily modified. No special-format headers are required.

A typical descriptor file looks like the following (the corresponding data file is 16K bytes):

```
(_data_type unsigned_1)
(_data_sets 1)
(_channels 1)
(_dimensions 256 256)
```

The most important keywords are `_data_type` and `_dimensions`. The `_dimensions` keyword specifies the size of the image (y-dimension is first). The `_data_type` keyword in the descriptor file specifies the type of the data file elements, and may take any of the following values (on the left is the `_data_type` specification in the descriptor file, and on the right is the Common Lisp type specifier):

- unsigned_1 = '(unsigned-byte 8)
- unsigned_4 = '(unsigned-byte 32)
- float = 'single-float
- ascii = 'character

The `_data_sets` and `_channels` keywords are currently not used by OBVIUS, but they must both be specified in the descriptor file and equal to 1. In the future, `_channels` may be used for color images (3 channels in an rgb image stored consecutively in the same data file) and `_data_sets` will be used for image sequences (multiple images from an image sequence stored in the same data file).

Currently, OBVIUS stores `image-sequences` as a datfile directory with one descriptor file and several data files named "data0", "data1", etc. The keyword `_data_files` in the descriptor file specifies how many data files are present.

The function `load-image` (see Section 6) loads images from datfiles and the function `save-image` saves images to datfiles. The `load-image` function will also load images from a file that is a stream of bytes in row-major order with one byte per pixel.

OBVIUS is designed to load images in various machine formats. Some machines (e.g., Suns and Symbolics) store 32-bit numbers in reversed byte order from each other. OBVIUS uses the `:byte-order` keyword in the descriptor of a datfile to determine the byte-order of the file. If necessary, `load-image` will reverse the order of the bytes as it reads in a 32bit image file. OBVIUS is not currently set up to load different floating point formats. Rather, you should use ascii or 32-bit file format if you have different machines at your site.

The following describes the functions for load and saving images and image-sequences:

(load-image `path &key ysize xsize reverse-bytes ->)`

If path is a directory, load-image assumes that path is a datfile, and `:ysize` and `:xsize` are ignored. Otherwise it assumes that path is an image file stored in row-major order with one byte per pixel. In such a case the keywords `:ysize` and `:xsize` may be used to specify the size of the image. If they are not specified then load-image assumes that the image is square. For a datfile, OBVIUS uses the information in the descriptor file to determine the element-type, the byte-order, and the dimensions of the data.

First, `load-image` converts the incoming data into floats. Then it uses a scale and pedestal, if specified in the descriptor file, to rescale the image data.

The `:reverse-bytes` argument is only relevant for 32-bit datfiles. If `:reverse-bytes` is nil then OBVIUS uses the byte-order keyword in the descriptor file to determine whether or not to reverse the bytes. If the `:reverse-bytes` is non-nil, then load-image ignores always reverses the bytes (ignoring the byte-order keyword in the descriptor file).

If `:->`is not specified and the global `*auto-bind-loaded-images*` is non-nil, then the image will be named (see Section 4) with the file-name or directory-name of path (not the whole path-name). See the tutorial for a simple example.

The `load-image` function calls the `load-image-sequence` function if the `_data_files` keyword is present in the descriptor file.

(save-image `image path &key element-type auto-scale reverse-bytes df-keys)`

Saves image as a datfile in path. The `:element-type` keyword specifies what type of file to save it as. `:Element-type` may be '(unsigned-byte 8), '(unsigned-byte 32), 'single-float, or 'character. The default is '(unsigned-byte 8). WARNING: Floating point word formats are different for different manufacturers, so it may not be possible to load an image that was saved from a different machine (use 'character or '(unsigned=byte 32) instead of 'float).

If `:element-type` is '(unsigned-byte 32) then the image is rescaled to fill the 32bit range. The appropriate scale and pedestal are written to the descriptor file so that the 32-bit numbers will be rescaled correctly when the image is reloaded. If the reverse-bytes keyword is T (default is nil) and the `:element-type` is '(unsigned-byte 32), then the byte-order is swapped during write.

If `:element-type` is '(unsigned-byte 8) and `:auto-scale` is NIL then no rescaling is

done. If `:auto-scale` is T (the default) then the image is rescaled to fill the range from 0 to 255, and the appropriate scale and pedestal are written to the descriptor file so that the 8-bit numbers will be rescaled correctly when the image is reloaded.

(load-image-sequence `path &key start-index end-index ->`**)**
Loads an image sequence from a datfile directory with multiple data files name "data0", "data1", etc. The keywords :start-index and :end-index may be used to load a subsequence. The load-image function calls the load-image-sequence function if the _data_files keyword is present in the descriptor file.

(save-image `sequence path &key element-type auto-scale reverse-bytes df-keys start-index end-index`**)**
Saves a sequence as a datfile directory with multiple data files. The keywords are the same as for saving images. The :start-index and :end-index keywords may be used to save a subsequence.

3.4 Memory Management

Lisp memory space is divided into several different areas. The most important distinction is between the dynamic and static areas. All user-defined storage space normally resides in the "dynamic" areas. Lucid Lisp provides several functions for examining and manipulating the dynamic Lisp memory space from top level. A complete explanation is available in the Lucid Common Lisp User's Guide. A few very useful functions are:

(change-memory-management `&key growth-limit expand ...`**)**
If a number is supplied for the `:growth-limit` keyword, it specifies the maximum Lisp process size in 64k segments. If a number is supplied for the `:expand` keyword, the Lisp dynamic space is expanded by that many 64k segments.

(room `&optional info`**)**
This function prints out information about the current state of the lisp memory map. If the optional argument `info` is t, more information is given.

In order to avoid garbage collecting large arrays such as image arrays, OBVIUS allocates storage for viewables and pictures from the static heap. *Since these arrays are never garbage collected, they must be returned explicitly to the system.* OBVIUS maintains separate "heaps" of static storage space for different array types (e.g., 'single-float, 'bit, etc). The memory allocator is designed to create heaps for any element-type that is a valid argument to the Common Lisp `make-array` function. Arrays are allocated from and returned to the static heaps with the functions `allocate-array` and `free-array` as described in Section 9.

Sometimes in the midst of a computation, OBVIUS will need to allocate an array from one of the static heaps that has no more free space. The global parameter `*auto-expand-heap*` controls the behavior in this situation. If `*auto-expand-heap*` is

non-nil, then OBVIUS will automatically call the `expand-heap` function (described below) to increase the size of the appropriate heap. The amount of expansion is controlled by the variable `*heap-growth-rate*`, which should contain an integer specifying the number of *elements* to add to the heap. If `*auto-expand-heap*` is nil, then OBVIUS will signal a continuable error: you can either abort the computation or enter `:c` to expand the heap by `*heap-growth-rate*` elements.

Several functions are provided for top-level management of the static heaps:

(expand-heap `type &optional size`**)**

This function expands the size of OBVIUS's static array area, for the given type of element. The type argument can be any array type. Examples are: 'single-float, '(unsigned-byte 8), and 'bit. The optional parameter `size` specifies the number of elements by which to expand the respective space. It defaults to the value of the global parameter `*heap-growth-rate*`.

(heap-status)

Prints a description of the current static memory usage.

(ogc `&key verbose`**)**

A primitive garbage collector that offers a means of reclaiming lost storage space (i.e., allocated arrays that belong to orphaned viewables – see the previous section on orphans for an explanation). Note that unlike the Common Lisp garbage collector, `ogc` must be called manually by the user. `Ogc` reclaims static arrays that do not belong to: (1) pictures, (2) viewables of pictures, (3) viewables that are bound to symbols in the current package or the `USER` package, or (4) inferior or superior viewables of the aformentioned viewables. All arrays that are not preserved are returned to the heap for reallocation. `Ogc` also prints a list of symbols in the current package that are currently bound to viewables, so that the user may explicitly destroy these. The former name for this function was `scrounge!`, which has been retained for back-compatibility.

Warnings: this function should *only* be called from top level, since it will not preserve viewables bound to local variables. Furthermore, it only preserves symbols that are in the current package (the name of the current package is the value of the variable `*package*`) or the `USER` package. Furthermore, if you have created viewables that are not accessible through pictures, symbols, or inferiors or superiors, *these viewables will be recycled by the garbage collector.*

4 Viewables

This section describes the various viewable subclasses. To add new viewable classes to OBVIUS, see section 9 of this manual. The currently defined viewables are are (indentation indicates inheritance):

image
    one-d-image
    slice
    bit-image
viewable-matrix
    image-matrix
    viewable-sequence
        image- sequence
            color-image
            image- pair
                one-d-image-pair
                one-d-complex-image
                complex-image
                polar-image
        complex-discrete-function
filter
    separable-filter
    hex-filter
discrete-function
    histogram
    log-discrete-function
    periodic-discrete-function
gaussian-pyramid
    laplacian-pyramid
qmf-pyramid [not yet ported from obvius-1.2]
overlay-viewable
    overlay-gray-viewable

4.1 Images

This section describes images and the functions provided for their manipulation. OBVIUS has several other related types of viewables (e.g., bit-image, image-pair, complex-image, polar-image, image-sequence, and one-d-image) that are also described below.

The following files in the `<obv>` directory contain source code that is relevant to images:

- **image.lisp** – contains the definition of the image class, accessors for the slots of images, and standard methods and utilities for images.
- **lucid-image-loops.lisp** – contains macros such as `loop-over-image-pixels` for efficient image accessing and looping.
- **imops.lisp** – contains the definitions for all of the arithmetic methods on images as well as some other assorted methods. See Section 6 for a complete list.
- **fft.lisp** – contains a fast Fourier transform, a Hilbert transform, and a discrete cosine transform.
- **synth.lisp** – contains functions as described below for generating synthetic images.

Make-image

The function `make-image` may be used to make a new image:

```
(make-image dims &key display-type name)
```

This function returns a new image of the specified dimensions, initialized to zeros. The dims parameter must be a list of two fixnums, the first specifying pthe y-dimension and the second specifying the x-dimension.

Note: The convention in OBVIUS is always to specify y first and x second. This is consistent with the row-major storage of arrays in both Common Lisp and C.

If only one dimension is specified or if one of the dimensions is 1, then `make-image` will return a `one-d-image`. This is a subclass of image that is displayed as a graph. Otherwise, its behavior should be the same as any other image.

The `:display-type` argument must be the name of a picture sub-class that is compatible with the image. The default display-type is either `'gray` or `'dither`. The default may be set from the `Viewable Default` sub-menu of the `Parameters` menu on the control panel.

Iref

"Iref is to images as aref is to arrays". It returns the value of the image at the specified location, and may be used with setf to modify the image values. For example,

```
(setf (iref im 20 30) 5.1)
```

sets pixel `(20,30)` to the value `5.1`. NOTE: It is not efficient to use `iref` in loops, since it is a method and therefore has substantial dispatch-time overhead (relative to the operation

being performed). Instead, you should either: 1) restructure your algorithm to use existing OBVIUS functions, which call C code, 2) write C code to perform your operation, 3) use the looping macros `loop-over-image-pixels`, `loop-over-image-positions`, or with-images (see section 9), 4) write lisp code that extracts the data arrays and uses aref to access them efficiently.

Print-values

The `print-values` function is used to generate a formatted printout of the values in some part of an image. For example,

```
(print-values image :y 0 :x 0 :y-size 10 :x-size 10)
```

prints the top-left 10x10 pixels of the image. The keyword parameters `:x` and `:y` specify the location of the upper-left hand corner of the subimage to be printed, and `:x-size` and `:y-size` specify the size of that sub-image. If these keywords are nil then the subimage is determined by the global parameters `*x-print-range*` and `*y-print-range*`. Each of these global parameters is a list of two floats between 0 and 1. For example if `*x-print-range*` is '(0.0 0.5) and `*y-print-range*` is '(0.5 1.0) then the lower left quarter of the image will be printed. The subimage is subsampled so that no more than `*max-print-vals*` samples are printed on each line. The globals `*max-print-vals*`, `*x-print-range*`, and `*y-print-range*` parameters may be reset from the `Globals` dialog box that is accessed from the `Parameters` menu on the control panel.

The `print-values` function is also defined for arrays, and for `image-pairs`.

Pictures of Images

Images may be displayed as `gray` (see section 5.2), `dither` (see section 5.3), as `surface-plot` (see section 5.7), or as `contour-plot` (see section 5.8).

4.2 One-d-images and Slices

One-d-images are really the same as images except that one of the dimensions is 1. So, all of the image operations that are defined for images work on one-d-images as well. The only difference is that a one-d-image is (by defaults) displayed as a `one-d-graph`. A one-d-image can be generated using make-image, e.g.,

```
(make-image '(100 1) :-> '1d-im)
```

or any of the synthetic image generating functions. The `slice` viewable class is a subclass of the `one-d-image` class. The one-d-image class and the slice class are defined in the file **image.lisp**. `one-d-image`s are typically displayed as `graph` pictures (see section 5.10).

4.3 Discrete Functions and Histograms

The viewables `discrete-function`, `periodic-discrete-function`, and `log-discrete-function` are defined in the file **discrete-function.lisp**. The class `histograms` are a subclass of `discrete-functions`. `Discrete-functions` are also used as lookup tables to do fast non-linear point operations on images (see the function `point-operation` in Section 6).

`Discrete-functions` are essentially a lookup table of function values — they are made up of a `data` array, an `origin`, and an `increment`. The first entry in the `data` array is taken to be the value of the function evaluated at `origin`, the next entry is taken to be the value of the function evaluated at `(+ origin increment)`, etc. The `evaluate` function is used to evaluate a discrete-function at a particular value. If that value falls between samples, then the result is interpolated.

`Periodic-discrete-functions` are the same as `discrete-functions` except they are interpreted as one full period of a periodic function. If the periodic function is defined from 0 to 1, evaluating at 1.5 is the same as evaluating it at 0.5. `Log-discrete-functions` are sampled on a log axis.

The following functions are used to make and evaluate discrete functions.

(make-discrete-function `func min max &key size interpolator endpoint name ->`**)**

Makes a discrete function of the function `func`, from `min` to `max`, with `:size` samples. The default for `:interpolator` is `'linear-interpolation` which does linear interpolation. If `:endpoint` is non-nil then the discrete-function includes a sample at `max`. Otherwise, the last sample is slightly below `max`. An example is given in the OBVIUS tutorial.

(make-periodic-discrete-function `func min max &key size interpolator endpoint name ->`**)**

The default `:interpolator 'periodic-linear-interpolation` which does linear interpolation, modulo `min` and `max`.

(make-log-discrete-function `func min max &key size base interpolator endpoint name ->`**)**

The default for `:base` is 10. Example:

```
(make-log-discrete-function '(lambda (x) (log x 10))
                            0.01 10.0 :base 10 :size 4)
```

The `discrete-function` includes samples at .01, .1, 1, and 10.

(evaluate `discrete-function value`**)**

Evaluates the discrete-function function at value. Interpolates if value is between two entries in the table.

`Discrete-functions` are displayed by default as `graph` pictures (see section 5.10), but they may also be displayed as `polar-plot` pictures (see section 5.11).

4.4 Bit-images

The `bit-image` class is a subclass of the `image` class. `Bit-images`, utilities for using `bit-images`, and operations on `bit-images` are defined in the file **bit-image.lisp** (see section 5.4). `Bitmaps` are the only display-type for `bit-images`. The threshold functions `greater-than`, `less-than`, `equal-to`, `greater-than-or-equal-to`, and `less-than-or-equal-to` discussed in Section 6 can be used to make `bit-images` from images.

Bit-images are just like images except that they have only one bit of data at each pixel (the bit-image data is stored using common-lisp bit arrays). So, some of the functions and accessors are inherited from images (e.g., `iref`, `dimensions`, `x-dim`, `y-dim`, `total-size`). Other methods are functionally equivalent to those defined on images (e.g., `mul`, `copy`, `similar`, `crop`, and `paste`). Other methods are modified somewhat (e.g., add is modified to perform a logical-or operation on bit-images).

4.5 Sequences

The classes `image-sequence` and `viewable-sequence` are defined in the file **sequence.lisp**.

`Image-sequences` may be used to work with temporal sequences of images or 3d image data (e.g., volumetric data used in medical imaging). Since `image-sequences` are made up of `images`, all of the image operations that are defined for `images` work on `image-sequences` as well. Image Sequences may be displayed as `pasteup` pictures (see section 5.6), or as flipbook pictures (see section 5.13).

A `viewable-sequence` consists of a number of sub-viewables, all of the same type. See the example in the OBVIUS tutorial. General (non-image) sequences can be displayed as `flipbooks` or as `overlays`.

4.6 Image Pairs, Complex Images, and Polar Images

The file **image-pair.lisp** contains the definitions of the `image-pair` , `complex-image` , and `polar-image` classes, that are compound images formed from a pair of standard images. `Image-pairs` are used to store the left and right frames of a stereo pair or the y- and x-components of a flow field. `Complex-images` are used for Fourier transforms. `Polar-images` store the magnitude and complex-phase of a `complex-image`. Since `image-pairs` are made up of `images`, all of the image operations that are defined for `images` work on `image-pairs` as well.

`Horizontal-pasteup` pictures are the default display type for `image-pairs`. These are a subclass of `pasteup` pictures (see Section 5.6). `Image-pairs` may also be displayed

as `vector-field`s (see section 5.9).

4.7 Filters

The viewable subclasses `filter` and `separable-filter` are defined in the file **filter.lisp**. Currently, all filters are considered to be finite impulse response (FIR) filters – infinite impulse response (IIR) filters will be implemented in a future release. Display methods are provided to plot them as `graph`s (for one-dimensional filters) or as `gray`s (for two-dimensional filters). The user may also create three-dimensional filters for application to image sequences: these are currently not displayed.

Filters are made up of a kernel array, an edge-handler, and a grid. For purposes of efficiency, filters are implemented to allow subsampling of the output. The grid slot specifies the subsampling to be performed in the convolution (see make-filter examples below). The edge-handler specifies how to handle the edges of the image while convolving. Current choices for edge-handler are nil (circular-convolution), and the keywords `:reflect1` (reflect the image through the edge pixel), `:reflect2` (reflect the image about the edge), `:zero`, and `:repeat`. Separable filters are compound viewables containing two filters.

The function `make-filter` is used to make filters. The first argument to these function should be a vector, a list, or an array specifying the filter kernel. Additional keyword arguments may be provided: `:step-vector`, `:start-vector`, `:edge-handler`. Examples:

```
(make-filter '(1.0 2.0 1.0))
(make-filter '((1.0 2.0 1.0) (2.0 4.0 2.0) (1.0 2.0 1.0)))
(make-filter 2darray :step-vector '(2 2))
```

The first of these makes a 1d filter, and the second makes a 2d filter. The last example makes a filter that will downsample by a factor of two when the `apply-filter` function is called (or upsample by a factor of two when the `expand-filter` function is called).

The function `make-separable-filter` may be used to create filters that are applied separably (i.e., two one dimensional filters, applied in the x- and y- directions). This function takes two filters, or two arrays, or two nested lists as arguments, along with all of the keywords described above. The first filter will be applied in the vertical (y) direction. You can also make higher dimensional separable filters (although the dimensionality cannot exceed that of the thing you are applying them to). For example, you can make a separable filter out of a 1D and a 2D filter, and apply it to an image sequence. The 2D filter may be separable or non-separable.

The `:edge-handler` argument determines what happens at the borders of an image when the `apply-filter` or `expand-filter` functions are called. Any local linear edge-handling scheme may be implemented. Some examples are implemented in the file **<obv>/c-source/edges.c**. The associated with each type of edge-handler is a C function that computes a new filter (set of weights) to be used when computing one of the edge samples

of the convolution. The arguments to this edge-handler function are described in the file **<obv>/c-source/edges.c**.

3D filtering

Three-dimensional filtering (on image-sequences) is also supported. To make a 3D filter, just pass a 3D array (or list) to the `make-filter` function. To make a 3D separable filter, you can call `make-separable-filter` with a 2D filter and a 1D filter.

Hexagonal filtering

Two-dimensional `hex-filter`s are defined on a hexagonal lattice: half of each row of the samples should be zero. These are a special class because the subsampling must be done differently. They have not been fully ported from OBVIUS-1.2 yet.

4.8 Pyramids

The `gaussian-pyramid` module may be loaded into OBVIUS by evaluating:

```
(obv-require 'gaussian-pyramid)
```

Pyramids are built with the function `make-gaussian-pyramid` and reconstructed with the `collapse` function. Various subbands may be accessed with the `access-band` function. Other pyramid operations are listed in in section 6.

Pyramids are currently undocumented. For now, look at example in the OBVIUS tutorial, and at the code in **gaussian-pyramid.lisp**.

4.9 Color Images

The `color-image` class is defined in the file **color-image.lisp**. `Color-images` inherit from `image-sequences`, but the images in the sequence are interpreted as different color bands (e.g., R, G, and B electron beam gun values).

This is unfinished code. We are writing a full color analysis package that keeps track of color-spaces, input devices (e.g., photoreceptors, cameras, scanners), output devices (e.g., monitors, printers), spectral sensitivities of these input and output devices, and spectral properties of surface reflectances.

For the time being, **color-image.lisp** defines simple color images that can be displayed as `color-pictures` or as `pasteups` (`pasteups` are the default display type).

4.10 Viewable Matrices

The `viewable-matrix` and `image-matrix` are defined in the file **viewable-matrix.lisp**. `Viewable-matrices` contain an arrays of sub-viewables. The `data` slot of a `viewable-matrix` is the array of sub-viewables. A number of other viewable types inherit from `viewable-matrices` (e.g., `image-sequence`s are `viewable-matrices` with a 1xN array of images).

`Viewable-matrices` are particularly useful for parallelizing pixel-by-pixel matrix operations. For example, this kind of operation is used in OBVIUS to convert `color-images` from one color space to another. For another example, let's say that you had six images. At each pixel you want to take the corresponding values out of each of the six images, put four of those values in a matrix and the other two in a vector, and then compute the inverse of the matrix times the vector. Believe it or not, this type of operation comes up a lot in visual modeling and image analysis, and it is very easy to do with `viewable-matrices`.

5 Pictures

This section describes the various picture subclasses. To add new picture classes to OBVIUS, see section 9 of this manual. The currently defined picture classes are listed below. Not every picture is applicable to every viewable. For example, `image`s may be displayed as `gray` or `surface-plot` pictures, but not as `graph`s. The indentation indicates inheritance of classes, and the viewables to which each picture type applies are listed in parentheses.

gray (image)
    pasteup (image-matrix, image-sequence, color-image, gaussian-pyramid, laplacian-pyramid, image-pair, complex-image, polar-image)
    dither (image)
color-picture (color-image)
bitmap (bit-image)
drawing
    graph (image, discrete-function, histogram)
    vector-field (image-pair)
    surface-plot (image)
    contour-plot (image)
    polar-plot (discrete-function, image-pair, viewable-sequence)
    scatter-plot (image-pair)
flipbook (viewable-sequence, image-sequence)
overlay (viewable-sequence)

5.1 Display

Pictures of a given type may be created using the `display` function, which is described in section 3.2.

5.2 Gray Pictures

The `gray` picture subclass is the usual display type for images. Each gray picture has several parameters that control its computation from the underlying image. The parameter values of a `gray` picture may be changed using the `Current Picture` dialog from the `Parameters` menu on the control panel. The parameter values may also be reset using the `setp` macro (described in section 3).

The transformation from a (floating point) image pixel value to a gray lookup table index is given by:

$$\text{round}\left(\frac{x-p}{s}\right),$$

where p is the picture's `pedestal` parameter, s is the `scale` parameter.

The size of the gray lookup table is determined by the `gray-shades` slot of the screen. The gray lookup table is a vector of indices into the screen colormap. The indices are ordered, from darkest to lightest. The sequence of gray shades that they refer to may not be linear: the `gray-gamma` parameter of the screen controls the set of gray values that are allocated.

OBVIUS will make use of the gray shades available. The screen parameter `gray-dither` is used to decide whether or not to dither the gray pictures. If its value is a number, and if there are fewer than `gray-dither` grays in the color map, then OBVIUS will automatically dither the picture. For example, if there are 8 gray shades allocated in the color map, and if `gray-dither` is greater than 8, then OBVIUS will dither the viewable values using a simple error-propagating dither algorithm into those 8 gray shades. If the value of `gray-dither` is nil, then OBVIUS will never dither. If it is non-nil (but not a number), then OBVIUS will always dither. All of the screen parameters mentioned (`gray-shades`, `gray-dither`, `gray-gamma`), may be adjusted from the `Current Screen` item on the `Parameters` menu.

If the value of `scale` is set to `:auto`, then OBVIUS automaticaly uses the range (max minus min) of the image. If the value of `pedestal` is set to `:auto`, then OBVIUS automaticaly uses the minimum of the image. Since these are stored on the info list, they only need be computed once.

In addition, each gray picture has `zoom`, `y-offset`, and `x-offset` parameters (inherited from the general picture class) that are described in Section 3.

5.3 Dither Pictures of Images

A `dither` picture is like a `gray` picture, except that OBVIUS forces it to only use two shades of gray (i.e., black and white). The dither class inherits from the gray class. This is a separate class for two reasons: 1) The user can request an image to be displayed in 1-bit dithered format, even on a deeper screen, and 2) the internal representation for dithers is more efficient (it uses a 1-bit deep bltable).

5.4 Bitmaps

The `bitmap` picture subclass, defined in the file **gray.lisp**, are used for displaying `bit-images`. The parameters of a `bitmap` are:

- `foreground` specifies the foreground color (e.g., :red or :green or any other color supported by LispView).
- `background` specifies the background color (e.g., :red or :green or any other color supported by LispView).

5.5 Color Pictures

The `color-picture` class is defined in the file **color-picture.lisp**, and may be used for displaying `color-images`. The `color-picture` interprets the sub-images of a `color-image` as corresponding to R, G, and B electron beam gun values. The parameters of a color picture are:

- `scale` analogous to grays and pasteups.
- `pedestal` analogous to grays and pasteups.

Example:

```
(progn
  (obv-require :color)
  (load-image "/images/clown")
  (change-class clown 'color-image))
(display clown 'color-picture)
```

In this example, `clown` is a datfile directory with three data files corresponding to R, G, and B. `Load-image` loads it as an `image-sequence` by default, so we use `change-class` to convert it to a `color-image`. Then it can be displayed as a `color-picture`.

Unfortunately, LispView (and consequently OBVIUS) does not (yet) support 24bit displays, so `color-pictures` are dithered. The parameter `rgb-bits` specifies how many bits to use for dithering each band. This parameter may be set in the `Current Screen` dialog from the `Parameters` button in the control panel.

5.6 Pasteups

The `pasteup` picture subclass are used for displaying `image-matrices`, `image-sequences`, `color-images`, `gaussian-pyramids`, `laplacian-pyramids`, `image-pairs`, `complex-images`, and `polar-images`. `Pasteups` are defined in the file **pasteup.lisp**.

A `pasteup` is a way of laying out (pasting up) a number of pictures so that they can all be seen simultaneously. The `pasteup` class inherits from the `gray` class, and thus these objects have all of the parameters of `grays`. In addition, there is an `independent-parameters` slot that forces all of the individual sub-pictures to have indentical rescaling parameters. If `independent-parameters` is non-nil, then the images that make up the `image-sequence` are rescaled independently. Otherwise, all of the images are rescaled with the same `scale` and `pedestal`.

The parameter values of a `pasteup` picture may be changed using the `Current Picture` dialog from the `Parameters` menu, or using the `setp` macro. The default values (the values used to initialize a new picture) for these parameters may be changed using the `Picture Defaults` menu. The parameters are:

- `independent-parameters`. If non-nil, then the images are rescaled independently.
- `scale` is ignored if `independent-parameters` is non-nil. `Scale` is the same as for gray pictures of images, if `independent-parameters` is nil. That is, if `independent-parameters` is nil and `scale` is not a number, then OBVIUS automatically uses:

  `(range image)`
- `pedestal` is ignored if `independent-parameters` is non-nil. `Scale` is the same as for gray pictures of images, if `independent-parameters` is nil. That is, if `independent-parameters` is nil and `scale` is not a number, then OBVIUS automatically uses:

  `(minimum image)`
- `border` specifies the spacing between pictures in the pasteup.
- `pasteup-format` may be either `:horizontal`, `:vertical`, or `:square`, specifying the layout of the sub-pictures.
- `zoom` see Section 3.2.

5.7 Surface Plots

The `surface-plot` pictures are perspective wire-frame views of image intensity data (i.e, they are plots of image intensity as a function of image position). They are defined in the file **drawing.lisp**, and the code for creating them is in **surface-plot.lisp**.

Surface plots have a number of parameters that control the transformation from the image pixel (world) coordinates to the display. The parameter values of a `surface-plot` picture may be changed using the `Current Picture` dialog from the `Parameters` menu, or using the `setp` macro. The default values (the values used to initialize a new picture) for these parameters may be changed using the `Picture Defaults` menu. The parameters are:

- `Theta` and `phi` specify the orientation of the viewer relative to the center pixel of the image data. The viewer always points toward the center of the intensity surface. `theta` is the angle (counter-clockwise) from the y-axis. `phi` is the elevation (aximuthal angle) above the x-y plane.
- `z-size` determines the height of the surface relative to its base. The height (in world coordinates) will be the product of `z-size` and the average of the image dimensions.
- `r` specifies the distance of the camera from the center of the image in world coordinates. It also controls the perspective distortion, since OBVIUS attempts to maintain an image-plane size consistent with the `zoom` parameter. Thus, `focal-length = r * zoom`. If the value is not a number, then OBVIUS automatically uses:

$$2\sqrt{\left(\frac{x}{2}\right)^2+\left(\frac{y}{2}\right)^2+\left(\frac{z}{2}\right)^2},$$

where x and y are the image dimensions, and z is the product of the `z-size` parameter and the average of the image dimensions. This is twice the minimal safe distance: the camera will be far enough away not to hit the surface.

- `Zoom` determines the approximate size of the surface plot (see Section 3. It is not exact due to perspective distortion.
- `X-step` and `y-step` specify the sampling rate for the wireframe plot in x- and y-directions respectively. If the value is not a positive integer, then automatically uses:

  ```
  (round (/ (x-dim image) 20))
  (round (/ (y-dim image) 20))
  ```

- `box-p` (either `T` or `nil`) specify whether or not to draw a 3D box around the plot.

  `box-color` the color to use for the box (e.g., :red or :green or any other color supported by LispView).
- `retain-bitmap` (either `T` or `nil`) specifies whether to draw the surface plot to an off-screen bitmap, or redraw it each time it is displayed.

5.8 Contour Plots

The `contour-plot` pictures are defined in the file **contour.lisp**. The code works by constructing 3D triangles from sampling the image in a grid. Then for each z-value, it constructs a curve containing segments each of which represents the intersection of the constant z-value plane with a particular triangle. So in the end, the contour contains a list of curves. This is a lot of computation, so you should be careful to keep the `num-levels` parameter relatively small and the `skip` parameter relatively large.

The parameters for contour-plots are:

- `z-min` and `z-max`: are Z-values for the smallest and largest contour levels.
- `num-levels`: Total number of contour levels.
- `skip`: The subsampling factor. If not a number, the sampling will be chosen automatically.
- `retain-bitmap`: (either `T` or `nil`) specifies whether to keep a copy of the bitmap, or redraw it each time it is displayed.
- `zoom`: determines the size of the graph. A zoom factor of 1 indicates that the samples of the underlying array should be plotted one pixel apart in the x-direction. The height of the graph is determined relative to the width using the `aspect-ratio` slot.
- `line-width` specifies the width of the graph lines (bars) drawn, in pixels.
- `color` specifies the color of the vectors (e.g., :red, :green, or any other color supported by LispView).

5.9 Vector Fields

`vector-field` pictures, defined in the file **drawing.lisp**, may be used to display `image-pairs`. The first image in the pair is interpreted as the y-component and the second is interpreted as the x-component.

The parameter values of a `vector-field` picture may be changed using the `Current Picture` dialog from the `Parameters` menu, or using the `setp` macro. The default values (the values used to initialize a new picture) for these parameters may be changed using the `Picture Defaults` menu. The parameters are:

- `skip` determines the subsampling factor. If not a number, the sampling will be chosen automatically to give a reasonable vector density for the given vector-field size. The default density can be set from the `Picture Parameters` menu.
- `zoom` determines the overall size of the vector-field picture, relative to the size of the underlying viewable. If nil, then the vector field will be the size of the underlying viewable (i.e., they would line up correctly if the vector field were superposed on the image). If t, the zoom will be determined to make the overall vector field be the size of the pane. If a dimension list, this will be the size of the vector field.
- `scale` determines the length of the vectors. The vector magnitude will be divided (as for gray) by the scale and the resulting number will specify the length of the vector, in units corresponding to the distance between the vector bases.
- `arrow-heads` If a number, draws vectors with arrow heads of that length. If not a nubmer, draws oriented line segments without arrow heads (e.g., for displaying orientation fields rather than flow fields). With arrow heads, the tail of the vector will correspond to the appropriate pixel location in the image-pair. Without arrow heads, the *middle* of the line will correspond to the appropriate pixel location in the image-pair.
- `color` specifies the color of the vectors (e.g., :red, :green, or any other color supported by LispView).

5.10 Graph Pictures

The `graph` pictures are defined in the files **drawing.lisp** and **graph.lisp**. The parameter values of a `graph` picture may be changed using the `Current Picture` dialog from the `Parameters` menu, or using the `setp` macro. The default values (the values used to initialize a new picture) for these parameters may be changed using the `Picture Defaults` menu. The parameters are:

- `graph-type` may be either `:line`, `:point`, or `:bar`. For most viewables, the automatic choice is `:line`. For `histograms`, the automatic choice is `:bar`.

- `x-range` is a list of length 2 specifying the range of values covered by the x-axis. If this is `:auto`, then OBVIUS automatically uses `(list 0 (total-size image))`.
- `y-range` is a list of length 2 specifying the range of the y-axis.If this is `:auto`, then OBVIUS automatically uses `(list (minimum image) (maximum image))`.
- `y-axis` is the x-coordinate position of the y-axis. If this is nil, no y-axis is drawn.
- `x-axis` is the y-coordinate position of the x-axis. If this is nil, no x-axis is drawn.
- `y-tick-step` and `y-tick-step` are the distances (in graph coordinates) between tick marks on the y- and x-axes, respectively. If set to `:auto`, OBVIUS will choose this to display between 4 and 10 ticks on the axis.
- `y-label` `x-label` [currently not used].
- `aspect-ratio` determines the physical (i.e., in screen pixel coordinates) ratio of height to width for the graphing region.
- `retain-bitmap` (either `T` or `nil`) specifies whether to keep a copy of the bitmap, or redraw it each time it is displayed.
- `zoom` determines the size of the graph. A zoom factor of 1 indicates that the samples of the underlying array should be plotted one pixel apart in the x-direction. The height of the graph is determined relative to the width using the `aspect-ratio` slot.
- `line-width` specifies the width of the graph lines (bars) drawn, in pixels.
- `axis-color` specifies color of the axes (e.g., :red, :green, or any number of other colors defined by LispView).
- `color` color of the graph and plot-symbols (e.g., :red, :green,or any number of other colors defined by LispView).
- `plot-symbol` Either `:circle`, `:square`, or `nil` (for no plot symbol).
- `symbol-size` specifies size of plot symbol, must be integer between 1 and 10.
- `fill-symbol-p` specifies whether or not plot symbol is filled, must be `T` or `nil`.

5.11 Polar Plots

The `polar-plot` pictures, defined in the file **drawing.lisp**, may be used to display `discrete-functions`. They may also be used to display `image-pairs` or `viewable-sequences` made up of pairs of discrete-functions.

For a single `discrete-function`, the angular coordinate of the plot is specified (in radians) by the `origin` and `increment` of the `discrete-function`. The radial coordinate

is specified by the `discrete-functions`'s data array. An example is given in the OBVIUS tutorial.

For a pair (either `image-pair` or `viewable-sequence` pair), the first viewable of the pair is interpreted as the magnitude (radial component) and the second is interpreted as the angle (in radians). Example:

```
(display (make-viewable-sequence
  (list (make-discrete-function '(lambda (x) (* 100.0 (cos x)))
                                        0.0 2-pi :size 63)
(make-discrete-function '(lambda (x) (* 2.0 x))
                                        0.0 2-pi :size 63)))
 'polar-plot)
```

The parameters of a `polar-plot` are:

- `maximum` The range of values plotted radially are between 0 and maximum.
- `line-width` specifies the width of the graph lines (bars) drawn, in pixels.
- `axis-color` *** not implemented yet
- `color` color of the graph and plot-symbols (e.g., :red, :green,or any number of other colors defined by LispView).
- `plot-symbol` Either `:circle`, `:square`, or `nil` (for no plot symbol).
- `symbol-size` specifies size of plot symbol, must be integer between 1 and 10.
- `fill-symbol-p` specifies whether or not plot symbol is filled, must be `T` or `nil`.

5.12 Scatter Plots

The `scatter-plot` pictures, defined in the file **drawing.lisp**, may be used to display data that is stored in an `image-pair`. The parameters of a `scatter-plot` are the same as those of a graph. An example of displaying a scatter-plot is given in the OBVIUS tutorial. The `scatter-plot` method may be also used as a shorthand for generating `scatter-plots`:

(**scatter-plot**`thing1 thing2 &rest initargs`)

Thing1 and thing2 may be images or vectors. Makes an image-pair and displays it as a scatter-plot. Initialize values for the scatter-plot picture parameters are handled by handing them off to the display function.

5.13 Flipbooks

The `flipbook`s are the standard display type for `viewable-sequences` and `image-sequences`. `Flipbooks` are defined in the file **flipbook.lisp**. `Flipbooks` can be thought of as a list of sub-pictures. For example, `flipbooks` of `image-sequences` contain a list of `gray` pictures.

Mouse bindings are provided to cycle through the subpictures in the flipbook (see section 8.1). `Flipbook` pictures have a number of parameters that control the delay between frames when the pictures are played back. There is also a way to change the scaling parameters of the `gray` pictures that make up the `flipbook`.

The parameter values of a `flipbook` picture may be changed using the `Current Picture` dialog from the `Parameters` menu, or using the `setp` macro. The default values (the values used to initialize a new picture) for these parameters may be changed using the `Picture Defaults` menu. The parameters are:

- `frame-delay`. Delay between frames (in approximately 1/60th secs).
- `seq-delay`. Delay between each repeat (in approximately 1/60th secs).
- `back-and-forth`. When t, the flip-book is shown forward and back during each repeat.
- `independent-parameters`. If non-nil, then the images are rescaled independently (see above description of `pasteup` pictures).
- `zoom`. Described in section 3.2.
- `sub-display-type` specifies the display types for the pictures that make up the `flipbook`. If T, then OBVIUS chooses the display type automatically. For `flipbooks` of `image-sequences`, the automatic choice is for the sub-pictures to be `grays`.

The `setp` macro may be used to set the parameters of the sub-pictures that make up a `flipbook`. For example,

```
(setp scale 100.0)
```

If `independent-parameters` is non-nil, then this will affect only the sub-picture that is currently being displayed. If `independent-parameters` is nil, then this will change the scale for all of the sub-pictues.

The parameters for the individual sub-pictures may be set independently if `independent-parameters` is non-nil. The parameters may be set using the `Picture Parameters` dialog in conjunction with the `previous-frame` (C-M-left) and `next-frame` (C-M-middle) mouse bindings. Bring up the current picture dialog (e.g., from the `Parameters` button on the control panel), and change the picture parameters of the current sub-picture. Use the mouse bindings to

change the current sub-picture. The dialog box is automatically updated to give the parameter settings for the new sub-picture. Then go ahead and change the parameters of the new sub-picture.

5.14 Overlays

Some `viewable-sequences` can be displayed as `overlay` pictures. The `overlay` class is defined in the file **overlay.lisp**. Currently, overlays are supported only for overlaying: (1) (1) `graphs` of `discrete-functions`, (2) `graphs` of `one-d-images`, (3) `bitmaps` of `bit-images`, and (4) for overlaying `bitmaps` of `bit-images` on top of a `gray` of an `image`. Examples are given in the OBVIUS tutorial showing how to display `overlays`. For more examples, look in the **overlay.lisp** file.

The parameters for the individual sub-pictures in the overlay may be set independently using the `Picture Parameters` dialog in conjunction with the `previous-frame` (C-M-left) and `next-frame` (C-M-middle) mouse bindings. Bring up the current picture dialog (e.g., from the `Parameters` button on the control panel), and change the picture parameters of the current sub-picture. Use the mouse bindings to change the current sub-picture. The dialog box is automatically updated to give the parameter settings for the new sub-picture. Then go ahead and change the parameters of the new sub-picture.

5.15 Hardcopy

The function `hardcopy` produces postscript output of OBVIUS pictures:

```
(hardcopy viewable
          &optional display-type
          &key path printer
          x-location y-location x-scale y-scale
          invert-flag suppress-text make-new
  &allow-other-keys)
```

Note that the syntax is quite similar to that of the `display` function described in section 5.1. Produces a postscript version of the viewable. The parameters for `hardcopy` are similar to those of `display`: any additional keywords are used to specify initargs of the picture. The optional parameter `display-type` specifies the picture class for the postscript picture. The function searches for a picture of that display-type in one of the panes (starting with the top of the current pane). If one is found, then it is copied to make the postscript picture. If none is found, then it makes a new picture using default parameters. Additional keywords may be passed to the hardcopy function (similar to the display function) to override the parameters of the on-screen picture. By default, hardcopy passes the keyword-value pair (:zoom :auto) so that the resulting picture will fill the page. The argument `:make-new` will force OBVIUS to make a new picture using default-parameters.

The keyword argument `:path` specifies a file pathname. If this parameter is nil (the default), OBVIUS writes the postscript to a temporary file, ships it to a printer, and then deletes the temporary file. The arguments `:x-location` and :y-location determine the location of the picture on the page (default is top-left corner). The arguments `:x-scale` and `:y-scale` specify by how much to rescale the size of the the picture (default is 1). Hardcopy prints the name of the viewable on the page just below where the picture appears (unless the argument `:suppress-text` is non-nil, in which case nothing is printed). The keyword `:printer` specifies which printer to use (default is the global `*default-printer*`).

The mouse binding, C-M-Sh-right, calls hardcopy to print a picture to the default printer.

The following global variables must be set for `hardcopy` to work properly:

- `*temp-ps-directory*` – Directory to hold temporary postscript files before they are sent to the printer. Typical value is something like: `"/tmp/"`.
- `*default-printer*` – Name of postscript printer, to be inserted into the `*print-command-string*` (e.g., `"laserwriter1"`).
- `*print-command-string*` – Command string for sending a postscript file to a printer. This is used as a format string (i.e., an argument to the Common Lisp `format` function) with two arguments: the filename and the printer name. That is, this string will be used in the following fashion:

  ```
  (format nil *print-command-string* <file> <printer>)
  ```

  where `<printer>` will default to the value of `*default-printer*`. On our system, the value of `*print-command-string*` is `"cat ~A | lpr -P ~A ; rm ~@* ~A"`. The `~@* ~A` construct allows us to process the filename twice (see Steele, Common Lisp: The Language). The result of this call to `format` will then be run as a shell program. Thus, the our example string will cat the first argument (filename) to the `lpr` command with the appropriate printer argument (second argument), and then call `rm` on the first argument (deleting the temporary postscript file).

6 OBVIUS Functions

OBVIUS provides standard functions for generating viewables, and many standard image-processing operations for manipulating them. Many of these image-processing operations are defined for many of the viewable classes. Most of these may be found on the menus in the control panel. In addition, you may use the `C-c C-a` command in Emacs to get the argument list for a given function, and `C-c .` to view the source code, as explained in Section 8.9. In this section, we list many of the operations, giving the name of the function and the argument list.

Result Argument

By convention, all of the functions that return viewables are written to take an optional keyword argument `:->`. The action taken by the function depends on the type of argument passed:

- If no argument (or a nil argument) is passed, a new viewable is created with a nil name slot, and the result is written into the newly created viewable. This is the default behavior: it is wasteful (since it always creates a new viewable), but avoids destructive modification which can be dangerous.
- If the argument is an existing viewable, the result is written into that viewable, thus destructively replacing the contents of the original viewable.
- If the argument is a symbol, a new viewable is created with the name slot bound to the symbol, the global value of the symbol (i.e., its value in the current package) is bound to the newly created viewable, and the result is written into the newly created viewable.
- If the argument is a string, a new viewable is created with the name slot bound to the string, and the result is written into the newly created viewable.

Point Operation Macros

There are a set of point-operation macros defined to mimic many of the standard Common Lisp functions. The names of these macros are the same as the names of the Common Lisp functions, augmented by a period ("point"). The current list of these point-operation macros is as follows: `+. -. *. /. sqr. sqrt. expt. exp. log. abs. mod. round. truncate. floor. sin. cos. min. max. average. realpart. imagpart. phase. incf. decf. fill. append. subseq.`

Where possible, these functions are written to take arguments like their Common Lisp counterparts. For example, the macro `+.` takes multiple viewable arguments and calls

`add` on them iteratively. As we go through the operations in the following sections, we will mention equivalent point-operation macros for some of the functions.

6.1 Image Utilities

The following are utilities and accessors for images:

(iref `image y x`**)**
See Section 4.1.

(image-p `thing`**)**
Returns t if thing is an image, nil if not.

(dimensions `image`**)**
Returns a list of the dimensions (y-dimension first).

(rank `image`**)**
Returns 1 for a `'one-d-image` and 2 otherwise.

(x-dim `image`**)**

(y-dim `image`**)**

(total-size `image`**)**
The total number of pixels in the image.

(print-values `image &key y x y-size x-size`**)**
See Section 4.1.

6.2 Generating Images

The following is a list of functions for making new images, and for making synthetic images:

(make-image `dims &key display-type ->`**)**
Returns a new image of the specified dimensions, initialized to zeros, with the name set to the value of the keyword parameter `:->`. The dims parameter must be a list of two fixnums, the first specifying the y-dimension and the second specifying the x-dimension.

(similar `image &key ->`**)**
Makes a new blank image using the argument `image` as a model (i.e., the new image will have the same dimensions and display-type as the model). Note that `similar` is defined for most viewable types.

(copy `image &key ->`**)**
Returns a copy of `image`.

(image-from-array `array &key ->`**)**
Makes an image from an array.

(make-synthetic-image `dims func &key x-range y-range ->`**)**
This function provides a general facility for mapping a function of two variables onto an image. The `dims` argument specifies the size of the image (in pixels). The argument `func` should be something that can be passed to funcall (e.g. a symbol with a function binding, a lambda expression, or an actual function). It will be applied to two floating point numbers *and must return a floating point value*. The optional keyword arguments specify the range of argument values over which to map the function (i.e., the pixel values of the synthetic image will correspond to the evaluation of the function at discretely sampled points over this range, with the endpoints set to the start and end of the range). The defaults are '(-1.0 1.0) for the `:x-range`, and the `:y-range` defaults to the value of `:x-range`. Example:

```
(make-synthetic-image '(128 128) '(lambda (y x) x) :-> 'xramp)
```

Several more examples are provided at the end of the **synth.lisp** file.

(make-impulse `dims &key amplitude ->`**)**

(make-ramp `dims &key orientation pedestal slope origin ->`**)**
Makes a ramp image of the given dimensions. `:pedestal` is the value in the center of the image (i.e., at the (d-1)/2 pixel position). `:slope` is measured in change per pixel distance. `:orientation` is in radians relative to the +X axis." `origin` is a list of two numbers specifying the (y x) coordinates of the pixel to which the pedestal applies.

(make-gratingdims `func period &key orientation phase ->`**)**
Makes an image of a one-dimensional periodic function. `func` must be a valid argument for the Common Lisp function `funcall`. `:period` indicates the desired period of the function in pixels. `func` will be evaluated at `*default-df-size*` points between 0 and `:period`. Linear interpolation will be used between these points. `:phase` should be in the interval [0,`:period`] and defines the argument to the function at the center of the image.

(make-square-grating `dims &key period orientation phase ->`**)**

(make-saw-tooth-grating `dims &key period orientation phase ->`**)**

(make-sin-grating `dims &key period orientation phase amplitude dc frequency x-freq y-freq ->`**)**
Specify either: (1) `:period` and `:orientation`; or (2) `:frequency` and `:orientation`, or (3) `:x-freq` and `:y-freq`.

(make-atan `dims &key center phase ->`**)**
Makes an angle (arc-tangent) image (i.e., each pixel contains its angular coordinate) between $-\pi$ and π.

(make-r-squared `dims &key center ->`**)**
Make an image of the function $f(r) = r^2$, where r is the distance from the point specified by `:center`. Default center is computed as $(d-1)/2$, where d will be respective image dimensions in pixels.

(make-1-over-r `dims &key center exponent amplitude zero-val ->`**)**
Computes $f(r) = ar^{-\beta}$ where r is the distance from the center of the image in pixels, a is specified by the keyword `:amplitude`, β is specified by the keyword `:exponent`, and the keyword `:zero-val` specifies what value to use at $r = 0$.

(make-zone-plate `dims &key amplitude phase ->`**)**
Makes a zone plate ($f(r) = \cos(r^2)$) with given `:amplitude` and `:phase`. Phase is in radians and refers to the origin.

(make-pinwheel `dims &key ->`**)**

(make-disc `dims &key radius origin transition-width ->`**)**

(make-left `dims &key ->`**)**

(make-top `dims &key ->`**)**

(make-uniform-noise `dims &key ->`**)**
Makes image with values randomly distributed between 0 and 1.

(make-gaussian-noise `dims &key mean variance iterations ->`**)**
Approximation to normal distribution that is faster than using `make-normal-random-image`. Better approximation (and slower) as `:iterations` is increased.

(make-random-dots `dims &key density ->`**)**
`:density` is a value between 0 and 1. Returns a 'bit-image.

(make-fractal `dims &key fractal-dimension ->`**)**

6.3 Loading and Saving Images and Image-Sequences

The following functions are used for loading and saving images to and from datfiles. Datfiles are described in more detail in section 3.3.

(**load-image** `path &key ysize xsize reverse-bytes ->`)

(**save-image** `image path &key element-type auto-scale reverse-bytes df-keys`)

(**load-image-sequence** `path &key start-index end-index ->`)

(**save-image** `sequence path &key element-type auto-scale reverse-bytes df-keys start-index end-index`)

(**save-picture** `path`)
Saves the image displayed in the current pane as an 8-bit datfile. Rescales the image according to the current scale and pedestal as specified in the title bar of the pane. This function only works on `'gray` pictures of `'image` viewables.

6.4 Operations on Images

The following operations take images as arguments and return images as their results. These functions are also defined for viewables that are made up of compound images (e.g., image-sequences, image-pairs, pyramids, etc.)

(**zero!** `image`)
Sets all the image values to 0.0, destructively modifying the image.

(**fill!** `image value`)
Sets all the image values to `value`, destructively modifying the image.

Numerical Point Operations

(**add** `thing1 thing2 &key ->`)
equivalent macro `+`.
This method and the next few methods operate on the images point by point. Thing1 and thing2 are either viewables or numbers.

(**sub** `thing1 thing2 &key ->`)
equivalent macro `-`.

(mul `thing1 thing2 &key ->`**)**
equivalent macro `*`.

(div `thing1 thing2 &key zero-val suppress-warning ->`**)**
equivalent macro `/`.
Attempting to divide by zero prints a warning message describing the number of pixels that were zero, and sets the result to either the positive or negative value of `zero-val`, depending on the sign of the numerator. The default value for zero-val is given by the global parameter `*div-by-zero-result*`. The warning message can be avoided by setting the `suppress-warning` keyword to t (default is nil).

(linear-xform `image scale offset &key ->`**)**
This returns (image * scale) + offset.

(negate `image &key ->`**)**
equivalent macro `-`. (unary).

(square `image &key ->`**)**
equivalent macro `sqr`. (Note that OBVIUS provides a function on numbers called `sqr`).

(square-root `image &key ->`**)**
equivalent macro `sqrt`.
Signed square root on the image, e.g., `(square-root -4.0)` gives -2.0.

(abs-value `image &key ->`**)**
equivalent macro `abs`.

(power `image thing &key ->`**)**
equivalent macro `expt`.
`Thing` may be an image or a number. If a number, then this function raises each pixel in `image` that number. If an image, then it raises each pixel in `image` to the correponding pixel value in `thing`.

(natural-logarithm `image &key zero-val ->`**)**
Attempting to take the log of zero prints an warning message and uses the value of `zero-val`. The default value for zero-val is given by the global parameter *div-by-zero-result*.

(gamma-correctim `gamma &key below above binsize ->`**)**
Clips between `below` and `above`, and calls point operation to gamma correct the rest of the values. Defaults for `below` and `above` are the minimum and maximum, respectively. For that default case, it computes $m + (M - m)[(x - m)/(M - m)]^{\gamma}$, where m and M are the minimum and maximum.

(point-operation `image function &key binsize ->`**)**
function must take a single float argument and return a float. If *binsize* is a number, this method creates a discrete-function (a type of viewable) by evaluating the function

at intervals of *binsize*. This is then used as a lookup table and the result image is computed using linear interpolation to generate values between the sample points. If binsize is nil (the default), the function is applied directly at each pixel (this can be very slow!). Example:

```
(point-operation einstein '(lambda (x) (/ 1.0 (+ 1.0 x))))
```

If the function being applied is periodic, it is advisable to use the function `periodic-point-operation` instead.

(**periodic-point-operation** `image function period &key binsize ->`)
Apply `function` with given `period` to `image` using a lookup table and linear interpolation. This is different from the `point-operation` function in that the lookup table is built to cover *one period*, with an extra point on the end. See **synth.lisp** for useful examples.

(**clip** `image below above &key ->`)
Returns an image identical to the argument except that values less than *below* are replaced with *below* and values greater than *above* are replaced with *above*.

(**round.** `&key divisor ->`)
Round to nearest integer value. If divisor is passed, divide by this value before rounding (Same as round function in common lisp).

(**floor.** `&key divisor ->`)
Compute greatest integer value less than each pixel value. If divisor is passed, divide by this value before flooring (Same as floor function in common lisp).

(**quantize** `image &key binsize origin ->`)
Image is quantized into bins. The value of each pixel is set to the *mean* (not the midpoint) of the containing bin. The default `:binsize` is the range of the image divided by the global parameter *default-number-of-bins* and the default orgin is the mean of the entire image.

Geometrical Operations

(**circular-shift** `image &key offset y x ->`)
This translates the pixels of an image, wrapping around at the edges. It is written so that the result may be the same image as the original. Note: parmeters are redunant, since `offset = (y x)`.

(**upsample** `image &key start-vector step-vector hex-start ->`)
For a full description of these keywords, see the description of `filters` in Section 4.7.

(**downsample** `image &key start-vector step-vector hex-start ->`)
For a full description of these keywords, see the description of `filters` in Section 4.7.

(**subsample** `image &key start-vector step-vector hex-start ->`)
Same as downsample.

(**resample** `image &key dimensions expansion-factor interpolator`)
Resample the image. `dimensions` is the dimension list for the resulting image. Alternatively, specify a number or pair of numbers for `expansion-factor`. `interpolator` must be either `:cubic` (the default), or `:linear`.

(**crop** `image &key offset y x dimensions y-dim x-dim ->`)
Returns the specified (rectangular) sub-image. Default `:offset` is `'(0 0)`, and default dimensions are the dimensions of the image minus the offset. Note that the parameters are redundant: you can pass `:offset` or `:x` and `:y` separately.

(**paste** `im big-im &key offset x y ->`)
Pastes the image into a larger image. Clips if im hangs over the edge. Again, parameters are redundant: `offset = (y x)`.

(**side-by-side** `im1 im2 &key space ->`)
Puts both images next to each other in a new image image twice larger. The keyword :space leaves the specified number of columns blank to separate the two images in the result.

(**transpose** `image &key ->`)
Transpose of the image: swap X and Y axes.

(**flip-x** `im &key ->`)
Flips the image around the center vertical axis.

(**flip-y** `im &key ->`)
Flips the image around the center horizontal axis.

(**rotate** `im &key angle interpolator origin`)
Rotate image about position specified by `origin` (a pair of numbers defaulting to center of image). `angle` is specified in radians and defaults to $\pi/2$. `interpolator` must be either `:cubic` (the default), or `:linear`.

(**warp** `im image-pair &key interpolator`)
Warp the image according to the image-pair. The image-pair is interpreted as a field of vectors, one associated with each pixel in the result image. Each result pixel is computed by moving to a position in the image specified by the corresponding vector and interpolating a value. `interpolator` must be either `:cubic` (the default), or `:linear`.

Comparison Point Operations

(**greater-than** `thing1 thing2 &key ->`)
equivalent macro `>`.

Arguments `thing1` and `thing1` can both be images or one of them can be a number. If they are both images, then the function does a point by point comparison. Returns a `bit-image`.

(less-than `image thing &key ->`**)**
equivalent macro `<`.

(equal-to `image thing &key result-type ->`**)**
equivalent macro `=`.

(greater-than-or-equal-to `image thing &key result-type ->`**)**
equivalent macro `>=`.

(less-than-or-equal-to `image thing &key result-type ->`**)**
equivalent macro `<=`.

(point-minimum `thing1 thing2 &key ->`**)**
`Thing1` and `thing2` may be images or numbers. Computes minimum as a pointop on two inputs. Each pixel of the result is the lesser of the two corresponding input values.

(point-maximum `thing1 thing2 &key ->`**)**

(square-error `image1 image2 &key ->`**)**
Returns the square of the difference of the two images.

(abs-error `image1 image2 &key ->`**)**
Returns the absolute value of the difference of the arguments.

(mean-square-error `image1 image2`**)**

(mean-abs-error `image1 image2`**)**

(max-abs-error `image1 image2`**)**

(zero-crossings `image &key ->`**)**
Computes a `bit-image` containing the zero-crossings of `image`.

(correlate `im1 im2 &key x y x-dim y-dim ->`**)**
Correlate images, circular-shifting one over top of the other. `x-dim` and `y-dim` specify to the area over which to correlate (these default to the dimensions of `im1`). `x` and `y` specify to the shift at which to begin correlation (that is, the value corresponding to a shift of `x` and `y` will appear in the upper-left pixel of the result). These default to zero.

Statistics

The following operations return numbers. These functions are also defined for viewables that are made up of compound images (e.g., image-sequences, image-pairs, pyramids, etc.) All but the last two store their computed values on the info-list of the viewable.

(minimum `image`**)**
Equivalent macro `min.` (unary). Returns minimum pixel value found in the image.

(maximum `image`**)**
Equivalent macro `max.` (unary).

(minimum-location `image`**)**
Returns (y,x) position of the minimum.

(maximum-location `image`**)**
Returns (y,x) position of the minimum.

(range `image`**)**
Returns difference between maximum and minimum.

(mean `image &key ignore-zeros`**)**
If `ignore-zeros` is non-nil, then the mean is computed without counting any of the zero values.

(variance `image &key ignore-zeros`**)**

(skew `image`**)**

(kurtosis `image`**)**

(entropy `image &key binsize origin numbins`**)**
The parameters are the same as for make-histogram.

Fourier Transform

(fft `image &key dimensions inverse center pre-center post-center ->`**)**
Returns a `complex-image` that is the DFT of the `image` argument. The algorithm used is a multidimensional radix-2 FFT (source code in **fft.c**. The `image` may be an `image`, an `image-pair`, or a `complex-image`. The size of the result may be specified with the `:dimensions` keyword and should be a list of two numbers, each a power of 2. By default, the X (Y) dimension of the result is the smallest power of two greater than the X (Y) dimension of the image. Computes the inverse FFT if the `:inverse`

parameter is non-nil. If the `:pre-center` argument is non-nil, the image is circular-shifted before computing the transform. If the `:post-center` argument is non-nil, the resulting transform is circular-shifted, placing the origin (DC value) in the *center* of the image. Both of these keywords default to the value of the `:center` keyword, thus allowing the user to pre- and post- center by passing a single keyword.

(power-spectrum `image &key center dimensions ->`**)**
Computes the squared magnitude of the Fourier transform. See the explanation of the `:center` and `:dimensions` arguments for the `fft` function.

(dct)
Not yet implemented.

(hilbert-transform `im &key orientation ->`**)**
This function does a ninety degree phase shift of all frequencies in the direction of a vector at angle (`orientation`) to the horizontal. Frequencies that are low with respect to this vector (ie, lie near the line perpendicular to this vector) create visual artifacts.

(derivative `im &key orientation ->`**)**
Multiply the FFT by $j\omega$, then inverse transform.

Filtering Operations

(apply-filter `thing1 thing2 &key ->(direction 0)`**)**
The argument thing1 may be a filter and thing2 an image or vice versa. This method is also defined on image-sequences. Correlates the image with the filter, (this is correlation because the filter is not flipped) downsampling as specified by the grid of the filter. Notice that the result argument (`:->`) cannot be equal to the image argument. Also, the thing must be at least the size of the filter if the edge-handler is non-nil. The `:direction` keyword argument determines the orientation of the filter. For example, if you apply a one dimensional filter to an image, it will be applied in the x direction. To apply it in the y direction, set the `:direction` argument to 1. See section 4.7 for more information.

(expand-filter `thing1 thing2 &key (zero t) ->`**)**
The argument thing1 may be a filter and thing2 an image or vice versa. Upsamples the image (as specified by the filter grid), then convolves the image with the filter. If the zero keyword parameter is nil (the default), the result of the convolution will be added into the result image; otherwise it replaces the contents of the result image. See section 4.7.

(gauss-in `image &key resample edge-handler kernel ->`**)**
Does upsampling followed by gaussian convolution (like a zoom-in with blurring). The default is to upsample by a factor of two and use a five-tap separable binomial filter.

(gauss-out `image &key resample edge-handler kernel ->`**)**

Does a gaussian convolution and downsampling (like a zoom-out with averaging). The default is to use a five-tap separable binomial filter and then downsample by a factor of two.

(blur `image &key level kernel edge-handler ->`**)**

Recursive blurring method. If level is 1 (the default), it does a gauss-out followed by a gauss-in. If level is n, it does n gauss-out's followed by n gauss-in's.

6.5 Functions for One-d-images

One-d-images are really the same as images except that one of the dimensions is 1. So, nearly all of the image operations that are defined for images work on one-d-images as well. There are some image processing operations that are not implemented for `one-d-images` including `flip-x`, `flip-y`,`upsample`, `downsample`, `subsample`, `transpose`, `side-by-side`. The following functions are particular for one-d-images:

(one-d-image-p `thing`**)**

Returns t if thing is a one-d-image, nil if not.

(make-slice `image &key x y z display-type ->`**)**

Makes a slice from `image` or `image-pair`. One (and only one) of `:x` and `:y` may be specified. This function creates a one-dimensional image that is a slice of a two dimensional image. This function may be called via a mouse click as described in Section 8.1. The function is also defined on image sequences.

6.6 Operations on Bit-Images

Bit-images are just like images except that they have only one bit of data at each pixel (the bit-image data is stored using common-lisp bit arrays). So, some of the functions and accessors are inherited from images (e.g., `iref`, `dimensions`, `x-dim`, `y-dim`, `total-size`). Other methods are functionally equivalent to those defined on images (e.g., `mul`, `copy`, `similar`, `crop`, and `paste`). Other methods are modified somewhat (e.g., add is modified to perform a logical-or operation on bit-images).

Bit-images may be made by performing any of several comparison operations on images (listed above), e.g., `greater-than`, `less-than`, `equal-to`, `zero-crossings`.

The following functions are defined for bit-images:

(bit-image-p `thing`**)**

Returns t if thing is a bit-image, nil if not.

(x-dim `im`**)**

(**y-dim** `im`)

(**iref** `im y x`)

(**make-bit-image** `dims &key display-type ->`)
Same arguments as for `make-image`.

(**image-from-bit-image** `bit-image &key ->`)

(**coerce-to-float** `bit-image &key ->`)
Same as `image-from-bit-image`.

(**similar** `image &key ->`)

(**copy** `image &key ->`)

(**crop** `image y x y-size x-size &key ->`)

(**paste** `im big-im x-offset y-offset &key ->`)

(**invert** `bit-image &key ->`)
equivalent macro `not`.

(**add** `bit-image1 bit-image2 &key ->`)
equivalent macro `or`.

(**mul** `bit-image1 bit-image2 &key ->`)
equivalent macro `and`.

6.7 Operations on Image-Pairs, Complex-Images, and Polar-Images

Since `image-pairs` are made up of lists of images, many of the image processing functions described above are defined on image-pairs. The functions `mul` and `div` are redefined for complex images to perform complex arithmetic.

The arithmetic functions (e.g., `add`, `sub`, `mul`, and `div`) are not yet defined correctly for `polar-images`. Convert them to complex images first (using `polar-to-complex`), and then do the arithmetic.

The following additional functions are defined for image pairs and complex images:

(make-image-pair `dims &key display-type ->`**)**

The argument `dims` may be either a dimension list or a list of two images. If it is a list of images then the first image in the list is taken as the y-component (or left-image), and the second is taken as the x-component (or right-image).

(make-complex-image `dims &key display-type ->`**)**

The argument `dims` may be either a dimension list or a list of two images. If it is a list of images then the first image in the list is taken as the imaginary part, and the second is taken as the real part.

(make-polar-image `dims &key display-type ->`**)**

(make-slice `image-pair &key x y display-type ->`**)**

See above description under image operations.

(image-pair-p `thing`**)**

(complex-image-p `thing`**)**

(print-values `image-pair &key y x y-size x-size`**)**

See Section 4.1.

(first-image `image-pair &key ->`**)**

If called without a result argument, this acts as a standard accessor function, returning the y component of the image pair. If called with a result argument, this copies the y component into the argument (or into a new image with the appropriate name if the argument is a symbol or string).

(second-image `image-pair &key ->`**)**

See description of first-image.

(y-component `image-pair &key ->`**)**

Same as `first-image`.

(x-component `image-pair &key ->`**)**

Same as `second-image`.

(left-image `image-pair &key ->`**)**

Same as `first-image`.

(right-image `image-pair &key ->`**)**

Same as `second-image`.

(imaginary-part `complex-image &key ->`**)**

Same as `first-image`.

(real-part `complex-image &key ->`**)**
Same as `second-image`.

(magnitude `polar-image &key ->`**)**
Same as `first-image`.

(complex-phase `polar-image &key ->`**)**
Same as `second-image`.

(switch-components `image-pair`**)**

(magnitude `image-pair &key ->`**)**
Returns the square root of the sum of the squares of the two sub-images.

(square-magnitude `image-pair &key ->`**)**
Returns the sum of the squares of the two sub-images.

(complex-phase `image-pair &key ->`**)**
Returns the arctangent of real-part/imaginary-part (values between -pi and pi).

(complex-conjugate `complex-image &key ->`**)**

(complex-to-polar `complex-image &key ->`**)**
Returns a `polar-image`.

(polar-to-complex `polar-image &key ->`**)**
Returns a `complex-image`.

(fft `complex-image &key inverse ->`**)**
Returns a complex image that is the FFT of the image. The x (y) size of the result is the smallest power of two greater than the x (y) dimension of the image. Computes the inverse FFT if the :inverse parameter is non-nil.

6.8 Operations on Discrete Functions

Most of the image processing functions (listed above) are also defined for `discrete-functions` and `complex-discrete-functions`. Most of the functions defined for `complex-images` are also defined for `complex-discrete-functions`. In addition, the following functions are defined for discrete-functions:

(make-discrete-function `func min max &key size interpolator display-type ->`**)**

Makes a discrete function of `func` from `min` to `max`. The number of bins is specified by the keyword argument `:size`. `:interpolator` is the function used to interpolate between samples. Example of use is given in Section 4.3.

(make-histogram `image &key display-type range binsize bincenter size ->)`

Returns a histogram of the image values. May be called via a mouse click (see Section 8.1). The default number of bins (`size`) may be set from the `viewable defaults` menu. The default `binsize` is the range of the image divided by the number of bins. `Bincenter` is the position of the center of any of the bins, i.e., x : x = (bincenter + i * binsize), for integer i, is the set of bin centers. `Range` can be used to specify a subrange of the image values. If range and binsize are both passed, then they determine the number of bins. For example:

```
(make-histogram im :binsize 1.0 :range '(100 200)}
```

histograms values between 100 and 200, with binsize 1.0 (i.e., size = 102, including the endpoints).

(evaluate `discrete-function value)`

Evaluates the function at value. Interpolates if value is between two entries in the table.

(distribution-mean `discrete-function)`

Mean of a df, treated as a probability distribution. For example, if you make a histogram from an image, then `(mean image)` and `(distribution-mean histogram)` are the same.

(distribution-variance `discrete-function)`

Variance of a df, treated as a probability distribution.

(distribution-mean-and-variance `discrete-function)`

Returns multiple values.

(range `discrete-function)`

Returns multiple values (max-min, min, max).

(domain `discrete-function)`

Returns multiple values, (max-min, min, max).

(make-soft-threshold-df`cutoff width &key invert ->)`

6.9 Operations on Sequences

Since image-sequences are made up of lists of images, many of the image processing functions described above are also defined for image-sequences. In addition, the following functions are particular for sequences:

(make-viewable-sequence `list &key length sub-viewable-spec display-type ->)`

List must be a list of viewables or nil. If it is nil, must provide :length and :sub-viewable-spec.

(make-image-sequence `list &key length sub-viewable-spec display-type ->`**)**
The argument `list` may be either a list of images or a dimension list or nil. If it is nil, must pass either (1) :length and :sub-viewable-spec or (2) :length and :dimensions. The dim-list can be (z y x) or (y x). In the latter case, the :length keyword must also be provided. If im-list is 2 integers it is interpreted as :dimensions. If im-list is 3 integers, it is interpreted as :dimensions=(cdr im-list) and :length=(car im-list).

(viewable-sequence-p`thing`**)**

(image-sequence-p`thing`**)**

(viewable-list`sequence`**)**
Conses a list of the viewables in the sequence.

(image-list`sequence`**)**
Conses a list of the images in the sequence.

(dimensions `sequence`**)**
Returns the (y,x) dimensions of the images in the sequence.

(x-dim `sequence`**)**

(y-dim `sequence`**)**

(sequence-length `sequence`**)**
equivalent macro `length`.

(framen `sequence &key ->`**)**
or `(frame sequence n &key ->)`
Accessor for the sub-images of a sequence. Returns the nth frame of the sequence as an image. If the result arg is nil, the existing sub-image is returned. If it is a symbol, the existing sub-image is returned, after altering its name. If an image is passed, then the appropriate sub-image is *copied* into that sub-image.

(sub-sequence `seq start-frame &optional end-frame &key ->`**)**

(append-sequence `seq1 seq2 &key ->`**)**
equivalent macro `append`.

(make-iterated-image `im func length &rest func-args`**)**
Makes a sequence of images, starting with `im`, by iteratively applying `func` (this must be a function which takes an image argument).

(make-slice `seq &key x y z display-type ->`**)**
Slices through the three-D volume of the sequence. If one coord is supplied, this will return a 2D image. If two are supplied, it will return a 1D image.

(make-synthetic-image-sequence`dims func &key length x-range y-rante z-range display-type ->`**)**
Analogous to `make-image-sequence`.

(reduce.`func seq &key start end from-end initial-value ->`**)**
Modeled on common Lisp reduce.

(map.`func seq &rest args`**)**
Modeled on common Lisp map. Args can be more sequences and a `->`argument.

6.10 Operations on Color-Images

`Color-images` are just like `image-sequences` except that the sub-images in the sequence are interpreted as different color bands. All of the methods defined for sequences apply to color-images as well.

The following additional functions are defined for color-images:

(color-image-p `thing`**)**
Returns t if thing is a color-image, nil if not.

(make-color-image `list &key length display-type ->`**)**
The argument `list` may be either a list of images or a dimension list. If it is a dimension list then the `:length` keyword must be specified.

(div `color-image array &key ->`**)**
Calls (`matrix-mul-transpose color-image array ->->`). See the section on `viewable-matrix` operations for details.

(div `color-image array &key ->`**)**
Calls color-transform with standard array for converting from rgg to yiq.

(div `color-image array &key ->`**)**
Calls color-transform with standard array for converting from yiq to rgb.

6.11 Operations on Viewable-Matrices and Image-Matrices

Operations on `viewable-Matrices` and `image-matrices` are defined in the file **viewable-matrix.lisp**. The `data` slot of a `viewable-matrix` contains an array of sub-viewables, as discussed in section 4.10. The majority of the operations defined on `viewable-matrices` are the standard image processing operations discussed above, e.g., add, mul, copy, similar, point-operation, minimum, mean, etc. (see **viewable-matrix.lisp** for complete list).

Additional functions specific for viewable-matrices are listed below. These include some matrix-mul operations, analogous to those that are defined for arrays. These matrix-mul operations are defined for any combination of a viewable-matrix, an array, and a vector. In other words, you can multiply a viewable-matrix by another viewable-matrix or you can multiply a viewable-matrix by an array (of numbers). ***Some of the methods on viewable-matrix/vector are not finished yet.

(make-viewable-matrix `data &key display-type ->`**)**
Data can be a nested-list of viewables, or an array of viewables.

(make-image-matrix `data &key size sub-viewable-spec display-type ->`**)**
Data can be a nested-list of images, an array of images, a dim-list, or nil. If data is nil, must pass either (1) :size and :sub-viewable-spec or (2) :size and :dimensions. If data is a dim-list, then :size must also be specified.

(diagonal-image-matrix `im-list &key display-type ->`**)**
Makes a new image-matrix putting images from im-list along the diagonal.

(viewable-matrix-p `thing`**)**
Returns t if thing is a viewable-matrix, nil if not.

(image-matrix-p `thing`**)**
Returns t if thing is a image-matrix, nil if not.

(dimensions `vbl-mat`**)**
Returns the dimensions of one of the sub-viewables.

(x-dim `vbl-mat`**)**
Returns the x-dim of one of the sub-viewables.

(y-dim `vbl-mat`**)**
Returns the y-dim of one of the sub-viewables.

(x-dim `vbl-mat`**)**
Returns the x-dim of one of the sub-viewables.

(y-dim `vbl-mat`**)**
Returns the y-dim of one of the sub-viewables.

(row-dim `vbl-mat`**)**
Returns the row-dim of one of the data slot (the number of rows in the matrix).

(col-dim `vbl-mat`**)**
Returns the col-dim of one of the data slot (the number of cols in the matrix).

(size `vbl-mat`**)**
Returns a list of (row-dim col-dim).

(**total-size** `vbl-mat`)
Returns (* row-dim col-dim).

(**vectorize** `vbl-mat &key size x-offset y-offset ->`)

(**columnize** `vbl-mat &key size x-offset y-offset ->`)

(**matrix-transpose** `vbl-mat &key ->`)

(**determinant** `vbl-mat &key ->`)
Works only for 2x2 or 3x3 matrices for the time-being.

(**matrix-inverse** `vbl-mat &key ->`)
Works only for 2x2 or 3x3 matrices for the time-being.

(**matrix-mul** `thing1 thing2 &key ->`)
Thing1 and thing2 may be viewable-matrices, arrays, or vectors.

(**matrix-mul-transpose** `thing1 thing2 &key ->`)

(**matrix-transpose-mul** `thing1 thing2 &key ->`)

(**dot-product** `thing1 thing2 &key ->`)

(**normalize** `thing1 thing2 &key norm ->`)

(**outer-product** `thing1 thing2 &key ->`)

6.12 Operations on Filters

(**make-filter** `kernel &key display-type edge-handler start-vector step-vector hex-start ->`)
(described in section 4.7).

(**make-separable-filter** `y-kernel x-kernel &key display-type edge-handler start-vector step-vector ->`)
(described in section 4.7).

(**x-dim** `filter`)

(y-dim `filter`**)**

(dimensions `filter`**)**

(volume `filter`**)**
Equivalent function: `sum-of`.

(filter-p `thing`**)**

(separable-filter-p `thing`**)**

(copy `filter &key ->`**)**

(negate `filter &key ->`**)**

(mul `filter thing &key ->`**)**
Thing may be a number or another filter.

(add `filter thing &key ->`**)**
Thing may be a number or another filter.

(sub `filter thing &key ->`**)**
Thing may be a number or another filter.

(shift-by-pi `filter &key ->`**)**
Defined on one-dimensional filters only. Returns a filter with taps that are frequency-modulated from the original filter taps (i.e., multiplies every other sample by -1).

(interpolate-hex-image `im &key filter ->`**)**
Does a simple interpolation from a hex image to a full-density (rectangular) image. Half of the samples in a hex image are zero (every other pixel on a line, with lines staggered).

6.13 Operations on Pyramids

The pyramid module is loaded into OBVIUS by evaluating:

```
(obv-require :pyramid)
```

Since pyramids are made up of lists of images, many of the image processing functions described above are also defined for pyramids, including `mul`, `sub`, `div`, `add`, `square-error`,

`abs-error`, `linear-xform`, `clip`, `abs-value`, `square`, `square-root`, `zero!`, `point-operation`, `apply-filter`, `expand-filter`, `blur`, `greater-than`, `greater-than-or-equal-to`, `less-than`, `less-than-or-equal-to`, `minimum`, `maximum`, `range`.

In addition, the following functions are particular for gaussian pyramids, laplacian pyramids, and qmf-pyramids. Take a look at `<obv>/tutorials/image-processing/pyramid-tutorial.l` for examples of how to use these functions.

(make-gaussian-pyramid `image &key level display-type filter ->`**)**
Make a gaussian-pyramid (a low-pass pyramid) from an image, and expand the pyramid to the specified level.

(gaussian-pyramid-p `thing`**)**

(make-laplacian-pyramid `image &key level display-type forward-filter inverse-filter ->`**)**
Make a laplacian-pyramid (a bandpass pyramid) from an image, and expand the pyramid to the specified level.

(laplacian-pyramid-p `thing`**)**

(make-separable-qmf-pyramid `image &key level display-type lo-filt edge-handler ->`**)**
Make a QMF-pyramid (a bandpass pyramid) from an image, and expands the pyramid to the specified level. Lo-filt is the kernel of an appropriate low-pass filter. The other filters are automatically contructed from this kernel.

(qmf-pyramid-p `thing`**)**

(copy `pyramid &key ->`**)**

(build `level`**)**
Extend the pyramid to the specified level.

(access `level &key ->`**)**
Returns the viewable that is at the specified level of the pyramid.

(access-band `level band &key ->`**)**
Returns the viewable that is at the specified level (and band) of the pyramid, i.e., use this to get the vertical band on the second level of a qmf pyramid.

(low-band)
Returns the low-pass band (top level) of the pyramid.

(collapse `include-low levels bands to-level &key ->`**)**
See pyramid-tutorial for details.

6.14 Operations on Arrays

Many of the image operations are also defined for arrays. These methods call C code if the arguments are all `single-float` arrays, or if they are all `fixnum` arrays. If they are some other array type, or undeclared type, or if they are mixed array types, then the operation is done in lisp (and is slower). Some of the operations only make sense for vectors or for 2d arrays. Some image processing operations that are not yet implemented for arrays.

Below, a few routines are documented in detail since they are not defined on images. At the end, a list of array operations that work like viewable operations is given.

(**almost-equal** `thing1 thing2 &key tolerance`)
: Compare two arrays and return the first array if the maximum difference between the two arrays is no more than `tolerance`. Otherwise return nil. The default tolerance is set in the global variable `*tolerace*`.

(**scalar-multiple** `thing1 thing2 &key tolerance`)
: Compare two arrays and return the first array if the first array is nearly a scalar multiple of the other. This is decided as follows. The least-squares scaling coefficient between the arrays is computed, and the maximum difference between the first array and the scaled copy of the second array is compared to `tolerance`. If the difference is no larger than `tolerance`, the first array is returned. Otherwise nil is returned. The default tolerance is set in the global variable `*tolerace*`.

(**randomize** `array number &key ->`)
: Return an array with the elements of `array` individually randomized by adding uniformly distributed values between `- number` and `number`.

(**shuffle** `array &key ->`)
: Return an array that contains the elements of `array` randomly reordered.

```
with-displaced-vectors
total-size dimensions
check-size rank z-dim y-dim x-dim similar copy
coerce-to-float print-values zero! fill! normalize
mean variance third-moment fourth-moment sum-of
product-of maximum minimum range mean-square-error
mean-abs-error max-abs-error point-minimum point-maximum
square-error abs-error add mul sub div negate square abs-value
linear-x-form square-root power natural-logarithm
point-operation periodic-point-operation quantize clip
circular-shift crop paste
greater-than less-than equal-to
greater-than-or-equal-to less-than-or-equal-to
covariance correlate array-magnitude
```

```
array-abs-value array-square-magnitude array-complex-phase
array-cross-product
```

6.15 Operations on Lists

Many arithmetic operations are also define on lists. They usually work just like the array or image equivalents. Some operations (for example, `mean`) will function on lists of viewables, arrays, etc. Many only make sense for lists of numbers (for example, `minimum`).

Below, a few routines are documented in detail since they are not defined on arrays. At the end, a list of list operations that work like array operations is given.

(elements `array &key ->`**)**
Returns a list containing the elements of `array`.

```
add sub mul div maximum minimum mean variance
square square-root natural-logarithm negate abs-value normalize
fill! dimensions total-size similar copy shuffle
vector-length vector-distance randomize almost-equal scalar-multiple
```

6.16 Operations on Matrices

The matrix module may be loaded into OBVIUS by evaluating:

```
(obv-require 'matrix)
```

The following is a list of matrix operations that are defined on arrays and vectors. Many of these methods are implemented only for float arrays. By convention, vectors are treated as row vectors. Using a row vector in a place where a column vector is required will generate a continuable error, where you will usually have the option to transpose the row-vector. To work with column vectors, you must use Nx1 arrays. Various utilities are provided to simplify row and column operations and conversions.

Note also that some of the C code called by these methods is not protected against floating-point exceptions. By default, Lisp will generate a continuable error if a floating-point exception is encountered while executing a foreign function. To turn this behavior off, evaluate `(setf (enabled-floating-point-traps) nil)`. Note that you will no longer receive *any* indication that these errors have occured!

OBVIUS Matrix operations are divided into three categories: basic matrix arithmetic, operations related to the SVD and other decompositions, and utilities (such row and column operations).

Matrix Arithmetic

(matrix-mul `thing1 thing2 &key ->`**)**
`Thing1` and `thing2` may be arrays or vectors (subject to the row-vector restrictions described above).

(matrix-transpose `array &key ->`**)**

(matrix-transpose-mularray)
thing1 thing2 &key `->` Multiplies the (tranpose of `thing1`) with `thing2`, without actually having to compute the transpose. `Thing1` and `thing2` may both be arrays, or one may be an array and the other a vector (subject to the row-vector restrictions described above).

(matrix-mul-transpose `thing1 thing2 &key ->`**)**
Multiplies the `thing1` times (tranpose of `thing2`). `Thing1` and `thing2` may both be arrays, or one may be an array and the other a vector (subject to the row-vector restrictions described above).

(outer-product `vector1 vector2 &key ->`**)**

(cross-product `vector1 vector2 &key ->`**)**

(dot-product `thing1 thing2 &key ->`**)**
Same as `matrix-transpose-mul`, but returns a scalar.

(vector-length `vector `**)**
Euclidean length of the vector.

(vector-length `vector1 vector1 `**)**
Euclidean distance between vectors.

(normalize `vector &key ->`**)**
Returns multiple values. First, a vector of unit length in the direction of the argument `vector`. Second, the length of the argument `vector`.

(matrix-inverse `array &key dimension-limit condition-number-limit singular-value-limit ->`**)**
Computes the Moore-Penrose left pseudo-inverse using the SVD. SVD returns (W, U, V) such that $M = SUV^t$. Pseudo-inverse is $M^\sharp = V1/WU^t$. Allows the calling function to restrict the dimensions of the pseudo-inverse. If `dimension-limit` is non-nil (it must be a whole number N), throw away all but the first N dimensions by setting those singular values to 0. If `singular-value-limit` is non-nil (it must be a number x), set to zero those singular values smaller than x. Default `singular-value-limit` is *machine-tolerance*. If `condition-number-limit` is non-nil (it must be a number

x), set to zero those singular values with condition numbers larger than x. The values returned are the inverse matrix, the minimum singular value (not including any zeros), the maximum singular value, and the number of nonzero singular values used (after the clipping of singular values as described above).

(determinant `matrix &key ignore-zeros u v`**)**
Uses SVD to compute determinant of a square matrix. `:Ignore-zeros` is an option to ignore any singular values that are zero, implicitly assuming that the matrix is nonsingular. The The arguments `:u` and `:v` may be used to pass u and v matrices to this function (and on to the SVD), to avoid consing.

(symmetric-p `array`**)**
**

(square-p `array`**)**
**

(diagonal-p `array`**)**
**

(identity-p `array`**)**
**

(unitary-p `array`**)**
**

(orthogonal-p `array`**)**
**

(singular-p `array`**)**
**

(identity-matrix `size &key ->`**)**
Returns a `size` by `size` identity matrix.

(diagonal-matrix `thing &key ->`**)**
`Thing` may be either a list or a vector. Returns a diagonal matrix with entries from thing.

(diagonal `matrix &key ->`**)**
Returns a vector with the diagonal entries of `matrix` as its contents.

(matrix-trace `matrix `**)**
Returns the sum of the diagonal entries of `matrix`.

Functions Related to SVD and Other Decompositions

(singular-value-decomposition `matrix &key u v s`**)**
equivalent macro `svd` Computes the singular values and complete orthogonal decomposition of a real rectangular matrix, M. The matrix is decomposed into UWV^t, where U and V are orthogonal matrices, that consist of the eigenvectors of MM^t and M^tM respectively. The entries of W are nonnegative, the singular values, which are the square roots of the eigenvalues of M^tM, ordered in decreasing sequence. If M is a square symmetric matrix, the columns of U are its eigenvectors and the elements of S are its eigenvalues. The function returns the multiple values (S, U, V) (S is returned as a vector of values on the diagonal of the W matrix). Should the function fail to compute the decomposition nil is returned and a warning is signalled. The arguments `:s`, `:u` and `:v` can be passed to avoid consing new arrays.

(with-svd `(s u v) matrix . body`**)**
** not documented

(with-static-svd `(s u v) matrix . body`**)**
** not documented

(singular-values `matrix`**)**
Returns a vector of the singular values of the matrix, in decreasing order (see description of singular-value-decomposition).

(left-singular-matrix `matrix`**)**
Returns the U matrix from the singular-value-decomposition.

(right-singular-matrix `matrix`**)**
Returns the V matrix from the singular-value-decomposition.

(principal-components `matrix &key dimension scale`**)**
Returns multiple values. First, the principal components of the rows of `matrix` in a matrix of the same dimension, with most important components first. Second, a vector with the principal values of `matrix`. Third, a vector with the singular values of `matrix`. When `scale` is t, the row of the principal components matrix are scaled by the principal values. By default they have unit length. The `dimension` argument allows you to restrict how many components (rows) you want in the resulting component matrix,

(condition-number `array`**)**
Returns square-root of quotient of minimum singular value divided by maximum singular value.

(quadratic-decomposition `matrix &key ->`**)**
Given matrix M, returns a matrix A such that $M = AA^t$. Uses A sufficient conditions is that M be positive-definite symmetric. Uses SVD. Generates an error if the matrix cannot be decomposed in this way.

(row-space `matrix &key ->singular-value-limit`**)**
Return an orthogonal basis for the space spanned by the rows of the matrix. The basis is generated using the SVD. The rows of the basis matrix returned are the principal components, and are in order of decreasing singular values. Any basis rows with singular values below `singular-value-limit` are not included in the basis. `singular-value-limit` defaults to 0.0.

(row-null-space `matrix &key ->singular-value-limit`**)**
Return an orthogonal basis for the space of vectors orthogonal to the rows of the matrix. See `row-space` for details.

(col-space `matrix &key ->singular-value-limit`**)**

(col-null-space `matrix &key ->singular-value-limit`**)**

(solve-eigenvalue `matrix &key ->`**)**
Find unit vector x such that x minimizes $x^t Mx$, for square symmetric matrix M. Finds the eigenvector corresponding to the smallest eigenvalue of M. Uses SVD.

(qr-decomposition `matrix &key q r`**)**
equivalent macro `qrd`
This function computes the QR decomposition of the matrix, such that matrix = q r, where q is orthonormal and r is upper triangular. The function returns a list of q and r. The arguments `:q` and `:r` may be used to pass q and r matrices to this function, to avoid consing.

(row-space-qr `matrix &key q r ->`**)**
Uses QR-decomposition instead of SVD.

(col-space-qr `matrix &key r ->`**)**
Uses QR-decomposition instead of SVD.

(row-null-space-qr `matrix &key q r tmp tol ->`**)**
Uses QR-decomposition instead of SVD.

(col-null-space-qr `matrix &key q r tmp tol ->`**)**
Uses QR-decomposition instead of SVD.

(determinant-qr `matrix &key ignore-zeros q r`**)**
Uses QR-decomposition instead of SVD.

Matrix Utilities

Unless otherwise noted, all row operations have corresponding column operation counterparts. Row and column indices begin at 0.

(make-matrix `&rest stuff` **)**

Return a matrix whose contents is `stuff`. The latter may be numbers, lists of numbers, lists of lists of numbers, vectors, or arrays. An `single-float` matrix of the appropriate size is made and filled with `stuff`. In the case of vectors and lists of numbers, the rows of the resulting array are filled in order.

(vectorize `matrix size x-offset y-offset`**)**

Returns a vector whose contents is the entire contents of `matrix`. The result is displaced to the original matrix. The keyword arguments allow you to displace the vector to any location and control its size.

(columnize `matrix size x-offset y-offset`**)**

Same as vectorize, except makes a `size` by 1 displaced array.

(row `row matrix` **)**

Returns a vector whose contents is a copy of the specified row of `matrix`.

(displaced-row `row matrix` **)**

Returns a vector *displaced to* the specified row of the matrix. Hence, if the resulting vector is modified, the data in the original matrix will be modified. This function has no column-operation counterpart.

(rows `row matrix` **)**

Returns a list of vectors which are copies of the rows of `matrix`.

(displaced-rows `row matrix` **)**

Returns a list vectors displaced to the rows of `matrix`. This function has no column-operation counterpart.

(col `col matrix` **)**

Returns an Nx1 matrix whose contents is a copy of the specified column of the `matrix`.

(cols `col matrix` **)**

Returns a list of copies of the columns of `matrix`.

(row-dim `matrix` **)**

Returns the number of rows of `matrix`.

(col-dim `matrix` **)**

(add-rows `matrix vector &key ->`**)**

Add a row vector to each row of a matrix. Arguments can be given in either order.

(add-rows `vector matrix &key ->`**)**

(add-cols `matrix vector &key ->`**)**

(**add-cols** `vector matrix &key ->`)

(**sub-rows** `matrix vector &key ->`)
Subtract `vector` from each row of `matrix`.

(**sub-rowsvector** `matrix`)
vector matrix &key `->` Subtract a each row of `matrix` from `vector`.

(**sub-cols** `matrix vector &key ->`)

(**sub-cols** `vector matrix &key ->`)

(**mean-rows** `matrix &key ->`)
Returns the row vector whose contents is the mean of all the rows of `matrix`.

(**mean-cols** `matrix &key ->`)

(**normalize-rows** `matrix &key ->`)
Returns a matrix with the rows of `matrix` independently normalized to a vector length of 1.0.

(**normalize-cols** `matrix &key ->`)

(**sum-rows** `matrix &key ->`)
Returns the row vector whose contents is the sum of all the rows of `matrix`.

(**sum-cols** `matrix &key ->`)

(**covariance-rows** `matrix &key ->sample`)
Calculates the covariance matrix associated with a matrix. The matrix is treated as many multivariate observations, with each observation in a row. Hence, the ijth entry of the resulting covariance matrix is the covariance between the ith and jth columns of `matrix`. If sample is t, the sample covariance is calculated, using (n-1) rather than n as the divisor.

(**covariance-cols** `matrix &key ->sample`)

(**shuffle-rows** `matrix &key ->`)
Returns a matrix with the rows of `matrix` randomly shuffled. This function has no column-operation counterpart.

(**swap-rows** `matrix first-row second-row &key ->`)

Returns a matrix identical to `matrix`, but with rows numbered `first-row` and `second-row` swapped. This function has no column-operation counterpart.

(**sort-rows** `matrix predicate &key key ->`)

Returns a matrix containing the rows of `matrix` sorted according to `predicate`, using `key`. This arguments are analagous to the Common Lisp `sort` function, except that `matrix` is not modified. This function has no column-operation counterpart.

(**paste-rows** `matrices &key ->`)

Paste a list of matrices into one big matrix, such that each row of the result is a row from one of the source matrices. In other words, paste the matrices on top of one another.

(**paste-cols** `matrices &key ->`)

(**append-rows** `&rest matrices `)

Like `paste-rows`, except that the matrices are passed directly as arguments, rather than as a list.

(**append-cols** `matrices &key ->`)

7 Data Analysis

OBVIUS has some basic tools for data analysis and simulation. These include:

Matrix Tools Singular value decomposition and related operations, and row and column operations for dealing with multivariate data.

Statistical and Simulation Tools Simple methods for sample statistics, probability distributions commonly used in modelling, and routines for generating noise drawn from particular distributions.

Model Fitting Linear least squares regression and STEPIT.

7.1 Matrix Tools

Many multivariate data analysis problems can be handled by treating data as rows or columns in matrices, and applying simple linear algebra techniques to these matrices. The matrix module contains many tools for handling matrices.

The matrix module may be loaded into OBVIUS by evaluating:

```
(obv-require :matrix)
```

The following is an outline of the matrix tools available in OBVIUS.

Matrix Algebra Most basic matrix algebra routines are available. For example:

```
;; Make a diagonal matrix, and take its inverse
(setq d (diagonal-matrix '(1 2)))
(setq d-1 (matrix-inverse d))

;; Make another matrix, and take the sum and the product
(setq m (make-matrix '((1 2) (3 4))))
(setq sum (add m d))
(setq product (matrix-mul m d))
```

Please check Section 6 for a detailed listing of matrix algebra routines.

SVD The singular value decomposition is at the heart of many numerical linear algebra algorithms. Please check Section 6 for more detail on the OBVIUS svd implementation, and related tools such as matrix inversion, subspace algorithms, other matrix decompositions, and linear least-squares regression.

An example calculation follows.

```
;; Example: Take the SVD of a matrix
(setq matrix (make-matrix '((1 2) (3 4))))
(svd matrix)
```

SVD returns the singular values of the matrix in an vector, and the left- and right-singular matrices as multiple return values.

Row and Column Operations OBVIUS has routines for doing simple row and column operations. Section 6 section documents these functions in detail. For example, you can access the rows of an example data matrix as follows:

```
;; Create a matrix and look at its row vectors
(setq matrix (make-matrix '((1 2) (3 4) (5 6))))
(setq first-row (row 0 matrix))
(setq second-row (row 1 matrix))

;; Swap the first and second rows of the matrix
(swap-rows matrix 0 1)
(setq first-row (row 0 matrix))

;; Look at the covariance matrix of the variables in each row
(setq covariance (covariance-rows matrix))
```

7.2 Statistical and Simulation Tools

The statistics module may be loaded into OBVIUS by evaluating:

```
(obv-require :statistics)
```

NOTE: The statistics module cannot be loaded unless Numerical Recipes code is set up on your system. See section A for instructions on installating Numerical Recipes in OBVIUS.

The following is a short outline of the statistical and simulation functions that are available. These functions are defined in the files **gaussian-noise.lisp** and **statistics.lisp**:

Sample statistics Some simple methods are available for standard statistics. Example:

```
;; Make a couple of vectors with data
(setq matrix-1 (make-matrix '((1 2 3 8 10 4))))
(setq matrix-2 (make-matrix '((4 5 2 2 1 8))))

;; Evaluate some statistics about the mean
(setq mean (mean matrix-1))
(setq standard-error (standard-error matrix-1))
```

```
;; Variance etc.
(setq variance (variance matrix-1))
(setq covariance (covariance matrix-1 matrix-2))

;; Examine an invented chi-square statistic
(cumulative-chi-square (sum-of (square (sub matrix-1 matrix-2))) 3)
```

Standard Functions Some commonly used probability distributions and other functions useful for modeling data are provided. These include the Gaussian, Weibull, Poisson, Binomial, Chi-Square, F, T, and others.

Noise Generation Several routines are useful for generating noise data for simulation. For example, you can generate an array filled with Poisson noise as follows:

```
;; Poisson noise example
(setq noise (make-array 100 :element-type 'single-float))
(setq lambda 7.0)
(dotimes (i (length noise))
  (setf (aref noise i) (float (poisson-noise lambda))))
(mean noise)
(variance noise)
```

Other routines exist for Gaussian, Bernoulli, and uniform noise (look through the files **gaussian-noise.lisp** and **statistics.lisp**).

7.3 Curve Fitting

Two types of curve fitting are available at present: linear least squares regression, and Chandler's STEPIT general-purpose minimization function. These are best explained by example. For more details, please refer to section 6 section, or examine the source code directly.

Linear Least Squares Regression Here is an example of how to solve a (cooked) regression problem.

```
;; Set up a matrix with the predictor variable
(setq predictor (make-array '(10 3) :element-type 'single-float))

;; Put random noise into the matrix
(randomize predictor 1.0 :-> predictor)

;; Set the observed matrix to be a linear transformation of the
;; predictor
```

```
(setq transform (make-matrix '( (1 2 3) (4 1 2) (4 9 2))))
(setq observed (matrix-mul predictor transform))

;; Solve the regression problem: the answer should be
;; very close to the transformation matrix.
(regress observed predictor)
```

Nonlinear Curve Fitting with Stepit Chandler's STEPIT program is a fast, general purpose minimization function. Conceptually, you give STEPIT a set of parameters to adjust, and an error function that depends on the values of those parameters, and STEPIT attempts to adjust the parameters so as to minimize the error function. In detail, the required arguments to `stepit-fit` are:

error-function function to be called with parameters followed by other arguments (see below). It is the return value of this function that stepit attempts to minimize.

initial-parameters starting point of parameters (a single-float vector or array).

error-function-args list of arguments that are passed (after the parameters) to the error function at each iteration. These arguments are passed to error-function one after another, not as a list.

keywords Detailed control of stepit's iterative process. See the source code for more information.

A simple example follows that illustrates the use of the OBVIUS interface to STEPIT. You are seeking the entries of the vector that is closest to a pre-set vector (the answer is obvious, the result should be the same as the pre-set vector). You set up the pre-set vector, and an initial guess for the closest vector. Then, give STEPIT the three required arguments:

```
;; Set up a dummy problem to be solved by stepit
(setq vector (make-matrix 1 2))
(setq initial-parameters (make-matrix 0 0))
(stepit-fit #'vector-distance initial-parameters (list vector))
```

STEPIT maintains its own set of parameters, call them `stepit-parameters` that it adjusts after each call to the error function. At each iteration, the error function is called as follows:

```
(vector-distance stepit-parameters vector)
```

STEPIT adjusts the entries of `stepit-parameters` until this function call is minimized. Obviously, STEPIT will figure out that the best choice of `stepit-parameters` is identical to `vector`.

Althought this problem is not very interesting, most parameter-fitting problems can be coded with this structure. STEPIT therefore provides a general method for fitting

models to data. Note that the length of time required for a stepit fit grows exponentially with the number of parameters, so if your models have more than about 5 parameters, be prepared to wait!

8 User Interface

This section describes the user interface provided for OBVIUS.

8.1 The Mouse

Mouse clicks within OBVIUS panes are captured and interpreted by OBVIUS. WARNING: Some X11 or OpenWindows window managers override the mouse bindings in obvius. In particular, twm allows the user to bind mouse-clicks in their .twmrc file. If your mouse is not responding – check your window manager manual pages.

All mouse-handling code is in the file **x-mouse.lisp**. The bindings can be altered within this file. The mouse can be used to cycle through the stacks of pictures, zoom, pan, and select images, among other things.

The following is a list of the current mouse bindings, organized according to the "bucky" keys (Shift, Control, and Meta). All actions are performed on the pane containing the mouse at the time of the button click.

buckies	left button	middle button	right button
Raw	select-viewable	refresh	select-pane/position-message
Control	previous-picture	next-picture	move-to-here
Shift	zoom-in	zoom-out	numerical-magnifier
Meta	describe-viewable	picture-parameters	picture-menu
C-Sh	pop-picture	center	drag
C-Sh-M	destroy-viewable	unbound	hardcopy

The following are picture-specific mouse bindings for gray pictures:

buckies	left button	middle button	right button
M-C	boost-contrast	reduce-contrast	histogram
M-Sh	x-slice	y-slice	crop

The following are picture-specific mouse bindings for flipbooks:

buckies	left button	middle button	right button
M-C	previous-frame	next-frame	show-movie

Raw (no bucky keys):

- Left: **Select viewable**. Selecting a viewable that is bound to a symbol prints that symbol into an emacs buffer at current prompt position (the "point"), where it is then ready to be evaluated. Of course, if the viewables have symbols bound to them, you can simply type the symbol name yourself.

  If the selected viewable has a nil name or a string name, then OBVIUS prints a more cryptic functional expression like (obvius::pid 1) that will also evaluate

to the viewable. This allows easy, interactive manipulation of viewable objects. Examples are given in the OBVIUS tutorial.

You can also access unnamed viewables via the variables *, **, and *** that are bound to the last three values returned by lisp. Also, the variables @, @@, and @@@ are bound to the last three viewables selected with a raw left mouse-click. This mouse-click also selects the pane as the current pane.

- Center: **Refresh**. The top picture on the picture stack is recomputed (if necessary) and then redisplayed. Calls (refresh pane).
- Right: **Select pane/Position Message**. The pane containing the mouse is set to be the current pane and some information that depends on the picture type is printed in the Emacs minibuffer. For example, clicking right on a grey picture of an image gives the (x,y)-position and the floating-point intensity value of the *image* (not the picture). This information is updated dynamically if you drag the mouse. Note that many other mouse clicks will also select the pane.

Control :

- Left: **Previous picture**. The stack of pictures in the pane is cycled. The pane is selected. Calls (cycle-stack pane nil).
- Middle: **Next picture**. The stack of pictures in the pane is cycled in the opposite direction as Control-Left. This pane is selected. Calls (cycle-stack pane t).
- Right: **Move to here**. The picture on top of the current pane is moved to the top of this pane. Selects this pane.

Shift :

- Left: **Zoom in**. Pixel-replication is used to produce an enlarged version of the image. The zoomed picture can be dragged using C-Sh-middle.
- Middle: **Zoom out**. This reduces the magnification of a picture. This will subsample below the resolution of the original image.
- Right: Numerical magnifying glass [Currently unimplemented].

Meta :

- Left: **Describe viewable**. This prints out a description of the viewable in the lisp listener.
- Middle: **Edit picture parameters**. Pops up a dialog to view and edit the parameters of the picture.
- Right: **Picture menu**. Pops up a menu to select (i.e., cycle to the top) any picture. Faster than using next-picture or previous-picture if the pane has a large stack.

Control-shift :

- Left: **Pop**. This mouse click calls (pop-picture pane), which is described in section 3.2. The picture on the top of the picture stack is destroyed. The viewable is not explicitly destroyed, although *it will no longer be accessible if it did not have a symbol name slot.*
- Middle: **Center**. Recenters the picture in the pane if it has been dragged.
- Right: **Drag**. Allows interactive re-positioning of pictures within the pane.

Control-meta-shift :

- Left: **Destroy**. This mouse click calls the method destroy on the viewable. All static memory used by the viewable its pictures is relinquished, the pictures are removed from the picture stacks, and if the viewable has a symbol name, it is unbound. If the viewable has inferiors that are otherwise unaccessable, then the inferiors are also destroyed. If the viewable has superiors, then this causes a continuable error that allows the user to destroy both the viewable its superior.
- Right: **Hardcopy**. Prints the picture to a postscript printer. The global parameter *default-printer* specifies which printer it goes to.

Meta-Shift : These bindings are *picture specific*!

For gray pictures:

- Left: **X-slice**. This produces a new viewable, a one-dimensional "slice" of a two-dimensional image, and pushes a one-d-graph picture of it onto the stack of the current pane. The y-coordinate of the slice is selected by the position of the mouse-click in the original picture.
- Middle: **Y-slice**. Orthogonal to x-slice.
- Right: **Interactive Crop**. Hold down the mouse and drag to define a square subregion of a picture. Releasing mouse creates a new viewable, cropped from the original.

Meta-Control : These bindings are *picture specific*!

For gray pictures:

- Left: **Boost-contrast**.
- Middle: **Reduce-Contrast**.
- Right: **Histogram**. This computes a histogram viewable, with a bin size which is controlled by the global parameter ***default-number-of-bins***, and pushes a one-d-graph picture of it onto the stack of the current pane. This calls the function make-histogram which is described in section 4.1.

For flipbook pictures:

- Left: **Previous Frame**. Cycle through flip-book picture of a sequence.
- Middle: **Next Frame**. Cycle through flip-book picture of a sequence.

- Right: **Display Sequence**. Display all frames of a sequence. See Section 5.13 for a description of the parameters that control delay between frames, etc. Any mouse button will stop the display.

8.2 The Control Panel

The control panel displays status information that tells you what the OBVIUS listener (top-level process) is doing. It also displays on-line, passive (i.e., you don't have to do anything to get it), mouse documentation. This means that when you move the mouse into an OBVIUS pane, the relevant mouse bindings are echoed in the message line; if you press a bucky key (shift, meta, or control), the documentation is updated accordingly.

To create a control panel, use the command (**make-control-panel**`&key left right top bottom`)
You may wish to put this in your `.obvius` initialization file. To destroy the control panel, call `destroy-control-panel` .

The control panel also has several menu buttons. Holding the right mouse button down on a menu button reveals the pull-down menu. Some of the menu choices bring up dialog boxes. The dialog box is created by releasing the mouse button on that choice. Some of the menu choices are set up to reveal sub-menus. The sub-menus are revealed by moving the mouse to the right while continuing to hold down the right button.

Within a dialog box, click left to move the cursor around and type as usual. Many of the dialog boxes are interfaces to function calls. The functions are executed by clicking either the `Execute` button or the `OK` button. When this is done OBVIUS prints an expression in the *lisp* buffer and evaluates it. Clicking left on an OBVIUS pane selects a viewable for the function argument, by inserting it at the cursor position in the dialog box.

At any given moment, there can only be a single function dialog box. If you try to bring up another one, OBVIUS will destroy the first one. You can add functions to the menus by adding them to particular global variables in the file `x-control-panel.lisp`. FOr example, you could add a new function `foo` to the "Misc" menu, but adding the symbol `'foo` to the value of the variable `obvius::*obvius-misc-functions*`. Changes to these variables are *dynamically* incoporated into the menus.

The menus on the control panel are:

Parameters : This menu allows you to adjust parameters of the current picture, the screen, the default slots of pictures and viewables, and global parameters. The items on the menu are described in more detail below.

Statistics : Menu of operations for computing viewable statistics (e.g., mean or variance of an image or image-sequence). Each menu choice brings up a dialog box for entering the function arguments.

Arith Ops : Menu of statistics operations (e.g., mean, variance, maximum, minimum, etc.).

Filter Ops : Menu of filtering operations (e.g., apply-filter, fft, etc.).

Geom Ops : Menu of geometric operations (e.g., crop, paste, slice, flip-x, etc.).

Compare : Menu of comparison operations (e.g., greater-than, less-than, mean-square-error, etc.).

Synth : Menu of synthetic image functions (e.g., make-impulse, make-sine-grating, make-zone-plate, etc.).

Matrix : Menu of matrix operations (e.g., matrix-multiply, singular-value-decomposition, etc.).

Misc : Menu of miscillaneous operations (e.g., print-values, hardcopy, etc.), and a submenu of optional modules that may be loaded.

8.3 Current Picture, Picture Defaults and Viewable Defaults menus

Current Picture : Brings up a dialog box to change parameters of the current picture (picture on top of the current pane). Buttons at the bottom allow user to Apply the changes to the picture (the display will be updated), revert to the current slot values, revert to the default slot values, or quit the dialog.

Viewables : Reveals a sub-menu that allows you to change the default parameters of a particular viewable class. The parameters for each viewable class are described in section 4. The default parameters are used to initialize a new viewable when it is first created. Changing the default parameter values with this menu has no effect on viewables that have already been created.

Pictures : Reveals a sub-menu that allows you to change default parameters of a particular picture class. The parameters for each picture class are described in section 4. The default parameters are used to initialize a new picture when it is first created. Changing the default parameter values with this menu has no effect on pictures that have already been created. The parameters of the current picture may be changed with the `Current Picture Parameters` menu (mentioned above).

8.4 Current Screen Parameters

The user can change the following screen parameters from the `Parameters` menu on the control panel (see above):

Gray-shades : Number of grays to allocate in the color map. If OBVIUS fails to allocate the requested number, it tries again with a smaller number.

Gray-gamma : Default gamma value in the allocated gray-shades.

Gray-dither : If gray-shades is less than gray-dither, then gray pictures will automatically be dithered.

Foreground : Foreground color used for graphics (e.g., it is the color used for each new graph pictures).

Background : Background color of each pane.

8.5 Global Parameters

There are a number of global parameters that affect how OBVIUS operates. The user can change the values of these globals from the `Parameters` menu on the control panel (see above). The function of each parameter is describe below:

`*default-printer*`: Name of a PostScript printer.

`*max-print-vals*`: Maximum number of values which the print-values method prints in a line of the image. See print-values, Section 4.1.

`*y-print-range*` and `*x-print-range*`: Default range of values in an image to be printed by the print-values function. Should be a list of two numbers between 0 and 1. See print-values, Section 4.1.

`*auto-bind-loaded-images*`: If t, automatically binds images loaded from files to symbols matching their filenames. See load-image, Section 3.3.

`*auto-update-pictures*`: If non-nil, out-of-date pictures will be updated when they are revealed. If nil (the default), user must explicitly refresh the pane to update an out-of-date picture.

`*auto-destroy-orphans*`: Slightly risky parameter causes viewables to be destroyed when 1) their names are rebound if they have no pictures or superiors, 2) their only picture is popped off of the pane and they do not have a symbol name or superiors, 3) their only superior is destroyed, and they do not have a symbol name or any pictures. These three cases are the situations in which OBVIUS can prevent the creation of orphans. Note that this does not prevent the user from creating their own orphans (e.g., make a new image with no name and don't display it). See also section 3.4 on memory management.

`*auto-display-viewables*`: If non-nil, viewables returned to the top-level listener are automatically displayed on the currently selected pane. This is the normal mode of operation.

`*heap-growth-rate*`: Number of elements to expand the heap when out of memory. See Section 3.4.

`*auto-expand-heap*`: When non-nil, expand heap automatically when out of memory. See Section 3.4.

`*div-by-zero-result*`: Default result used by /-0 and div functions when the divisor is zero. See Section 6.

8.6 Running OBVIUS on a remote host

The networking capabilities of X Windows (or OpenWindows) make it possible to open OBVIUS windows on a machine other than the one on which the OBVIUS process resides. Like other X clients, OBVIUS examines the value of the shell environment variable, DISPLAY, in order to determine which screen to open its windows on. If your X server finds more than one screen, you will be able to open obvius panes on any of them using the `:screen` keyword argument to **new-pane**. This will typically be the case if you have two monitors attached to one workstation, or if your framebuffer has a separate overlay plane. The list of screens that OBVIUS knows about is kept in the (unexported) global variable `obvius::*screen-list*`.

Because X is a server-based window system, each bitblt requires that data be transferred between the OBVIUS (lisp) process, and the X server process. Displaying a picture for the first time will be delayed by the time it takes for this transfer, but the remote (server) copy of the data is retained as an X pixmap, and subsequent redisplay, via refresh or cycling a pane, is free of this penalty.

8.7 Screens and Windows

OBVIUS maintains a list of **screens** on which its panes reside. This mechanism makes it possible to have OBVIUS panes on several screens which may have different capabilities at the same time, and leaves open the possibility of using different window systems on the various screens. System-independent code which defines windows and screens is in pane.lisp, while system-dependent window code for X is in the file **x-window.lisp**.

OBVIUS panes are described in section 3.2. These panes are built on top of LispView window objects and thus inherit all of the attributes of those windows. Accordingly, panes can be moved, resized, refreshed, hidden, closed, or opened in the same manner as any other X11 window. NOTE: if you are using X11 (as opposed to OpenWindows), you should not destroy an OBVIUS pane from the window manager as this will destroy the client process (i.e., Lisp)!! Use the OBVIUS destroy method to destroy a pane (this can be invoked from the `misc` menu on the control panel).

In LispView, mouse events are handled by a separate Lisp process. Multiprocessing poses particular difficulties for OBVIUS because some of the mouse clicks may result in lots of computation, or they may need to destructively modify data structures that are being simultaneously accessed by the main process. OBVIUS has two different solutions for

handling these problems:

1. If the mouse click involves only modification of pictures, the computation *is* performed by the mouse process. To avoid data structure collisions (e.g., the mouse process and main process both trying to display something in the same pane), OBVIUS "locks" the pane being modified, using the macro `with-locked-pane` to give exclusive access to one process (see **pane.lisp** for examples of use).

2. If the mouse click requires substantial computation or creation of viewables, OBVIUS does not let the mouse process handle the event directly. Rather, OBVIUS hands off compute-intensive evaluation to the main lisp process via an evaluation queue. The main lisp process polls the eval queue each time it goes through its read-evel-print loop. The upshot of this is that some mouse events are handled immediately (e.g., cycle stack) whereas others are not (e.g., make-slice). If the event is going to be handled in a delayed fashion, OBVIUS will notify you by putting up the status message "Enqueued form: ¡form¿".

8.8 Emacs and OBVIUS

The user interface of OBVIUS is greatly enhanced when it is run as a sub-process of the Emacs editor. The following capabilities are available:

- Quick entry of viewable arguments to commands [left mouse click (select-viewable)].
- Passive mouse documentation and system messages in the Emacs minibuffer (the bottom line of the Emacs window) (Note: this is only true in the absence of the contol panel).
- Interactive Lisp shell buffer with command history mechanism.
- Evaluation of lisp expressions from Emacs file buffers [`C-c e`], and In-package compilation of lisp functions from Emacs file buffers [`C-c c`].
- Function argument lists [`C-c C-a`], documentation strings [`C-c C-d`], symbol apropos [`C-c C-h`], symbol description [`C-c C-i`], macro expansion [`C-c C-m`], and source file editing [`C-c .`].
- Filename completion for load-image requests [`C-c l`].
- Complete record of interaction with OBVIUS (in the form of an Emacs buffer).

The extensions to Emacs that provide this functionality are contained in the `<emacs-source>` files. For more information, see the file `<emacs-source>`/cl-shell.doc.

8.9 Help

The following helpful functions are provided:

(**obvius-parameters** &key doc)
: Prints out a list of the system parameters and their current values. If the optional *doc* argument is non-nil, the documentation of each parameter will also be printed. Documentation of each parameter is also available via a call to the Common Lisp documentation function: (documentation '`<symbol>` 'variable).

(**obvius-commands**)
: Prints out a list of all OBVIUS functions which are exported.

(**obvius-variables**)
: Currently not implemented.

In addition, the source code for any OBVIUS function may be found in using the "C-c." command. Also, the Common Lisp function `(apropos <piece-of-symbol-or-string>)` will search for all symbols containing the given symbol or substring.

9 Hacking

This section is provided for users who want to write their own extensions to OBVIUS. As OBVIUS becomes larger and more people work on it, it becomes increasingly important to observe the existing programming conventions. It is generally a good idea to use existing code (copy it and modify it) as a prototype for whatever you are doing. In the following sections, we try to describe and motivate these conventions.

9.1 Writing Additional Image Processing Functions

Since OBVIUS was originally conceived as an image processing package, it contains a library of image processing functions in the file **imops.lisp**. Here is a simple example:

```
(defmethod add ((val number) (im image) &key ->)
  (with-result ((result ->) im 'add val im)
    (add (data im) val :-> (data result))
    result))
```

Note the result keyword argument, `:->`. All functions that return viewables should take this keyword argument. The standard handling of this argument is described in section 6. OBVIUS provides a macro, `with-result` that makes it easy to provide standard handling of this result argument.

The with-result macro

The `with-result` macro should be used by *all* operations returning viewables. It provides a simple mechanism for creating functions with uniform result argument handling. The syntax is as follows:

```
(with-result ((res-sym res-arg) model . history)
   . body)
```

The programmer should think of this macro as being very similar to the Common Lisp `let*` macro. The forms of the `body` are executed within a lexical environment where the symbol `res-sym` is bound to a "result" viewable. This result viewable is protected from the garbage collector, and from auto-destruction of orphans (i.e., it is placed on the `*protected-viewables*` list). The macro returns the value (or multiple-values) returned by the `body`. The `res-arg`, `model`, and `history` arguments are used to construct the result viewable.

The `model` argument must either be a plist of keyword initargs specifying the type and slots of the result, or an existing viewable which is used as a model for the result. If

a plist is used, the first two elements should be the keyword `:class` and the symbol name of a viewable class. The remaining keyword/value pairs should be those you would pass to `make-instance` in order to create the viewable. That is, they are a set of initialization arguments (initargs) for the object.

The `res-arg` is typically the result argument, `->`, as passed by the user. If this is an existing viewable, OBVIUS will check that it is consistent with the specification given by the `model` and then bind `res-sym` to it. If it is a name (nil, a symbol, or a string), then OBVIUS will create a new viewable according to the specification of the `model` and give it this name.

The `history` argument is typically the symbol name of the function (`'add` in the example above) followed by the arguments to the function (except for the result argument itself). This list is saved in the history slot of the viewable. If no `history` argument is passed, then the history of the result viewable will be set according to the operations performed in the `body`.

The macro `with-result` calls the methods `set-result` and `set-not-current`, both of which are described below in section 9.3. It also calls `set-history` to set the history slot of the resulting image according to the `history` argument. Examples of usage can be found in **imops.lisp**. Calling `macroexpand-1` on these examples may help to clarify the use of the macro.

Temporary Viewables

Often, you will need to generate intermediate viewables within the body of a function. These viewables are not needed once execution of the function is completed: they should be destroyed so that their static arrays are returned to the system for re-allocation. OBVIUS provides a macro for this purpose:

(with-local-viewables `image-list . body`**)**

This macro has the same syntax as the Common Lisp `let*` macro. It performs a local binding (using `let*`), executes the body within an `unwind-protect` and then (silently) destroys all viewables that were locally bound, thus freeing any heap memory belonging to those viewables. The returned value (or multiple values) is that of the `body`.

It is an error to return a temporary viewable from the body of `with-local-viewables`. You may, however, return inferior viewables of temporary viewables (even if `*auto-destroy-orphans*` is non-nil). For example, you may create a temporary complex image by computing an fft, and then return the real-part of this complex image. The complex image will be destroyed, but the real-part will be preserved. See the sections on destroying viewables and orphans (section 3) and the next section (section 9.2) for a more complete explanation of `*auto-destroy-orphans*`.

If you need to create temporary static arrays (i.e., arrays allocated from OBVIUS's static heap), use the the macro `with-static-arrays`, described in the next section.

Efficient Looping

In the `add` method given above, we call another method on the data arrays of the argument image and the result image. The methods on arrays are typically designed to call C functions, which run at least 20-30% on Sun SPARC machines. Despite this speed improvement, it is somewhat inconvenient to have to write C code every time you want a new image operation. It is often simpler just to write the code in LISP. Several macros, defined in the file **lucid-image-loops.lisp** are provided to make it easier to write new image processing functions in LISP:

(loop-over-image-pixels `image-list . body)`
Provides efficient looping mechanism for images. The syntax is like the Common Lisp `let*` macro. See examples in the files **lucid-image-loops.lisp**, and **imops.lisp**.

(loop-over-image-positions `image-list (ypos xpos) . body)`
Provides efficient looping mechanism for images, with access to the current coordinates. See examples in the files **lucid-image-loops.lisp**, and **synth.lisp**.

Calling Foreign Functions

If you do need to call C code (or some other language) from OBVIUS, you can look at the foreign function definitions given in `lucid-ffi.lisp`. Communication between Lisp and C can be complicated, and you should be aware of a few difficulties:

- If you pass non-static data structures to C code, you must be careful that the Lisp Garbage Collector is not invoked. See the discussion of this in section 9.2 on memory management.
- If the C code encounters a floating point exception (eg, underflow, overflow, divide-by-zero, log-of-negative-number...), then Lisp will (by default) generate a continuable error. This is appropriate if it only happens once, but if it happens in a loop over a large data structure, it is disastrous! The user can turn off this error handling by evaluating `(setf (enabled-floating-point-traps) nil)`, but then they will not know when an exception has occured. Much of the C code in obvius handles floating exceptions more gracefully (see, for example, `internal-div`), returning the number of exceptions encountered within the loop.

9.2 The Memory Manager

OBVIUS objects that contain large arrays should make use of OBVIUS's static array allocator, defined in the **memory.lisp** file. In Lucid Common Lisp, the "static" storage area is a portion of memory that is not managed by the Lisp garbage-collector. The reason for using static arrays (as opposed to normal "consed" arrays) is three-fold:

1. Lucid requires that arrays passed to foreign functions are static, and begin on even word boundaries. Technically, this is only a problem if the Lisp garbage collector is invoked while in the foreign function. In this case, the arrays will be relocated by the garbage collector, and the foreign function will write into the wrong portions of memory. This can situation can arise if the foreign function calls a Lisp function (a "callback"), *or if the foreign function calls* `malloc`!

2. We want to prevent large arrays from being garbage collected.

3. Keeping images together in memory improves virtual memory swapping behavior.

The OBVIUS static array allocation function (`allocate-array`) works by displacing sub-arrays into large static vectors. OBVIUS keeps track of the space that has been allocated, and provides a function to de-allocate arrays (`free-array`). The disadvantage of this explicit memory allocation scheme is that arrays allocated in this manner must be freed by the user or the programmer.

The following memory allocator functions are exported:

(allocate-array `dims &key element-type initial-element`**)**

This function is similar to the `malloc` function in C. Returns an array of the specified dimensions and type, allocated from OBVIUS's static heap. The args are the same as for Common Lisp `make-array`. The array returned is actually displaced to a large static array. Nevertheless, it is guaranteed to begin on an even word boundary. Remember that these arrays are not garbage collected and so must be explicitly freed. If the memory allocator does not have a large enough block of the appropriate type of static memory, and if `*auto-expand-heap*` is nil, a continuable error is given, allowing the user to expand the heap by `*heap-growth-rate*` elements. If `*auto-expand-heap*` is non-nil, this expansion will happen automatically. NOTE that Lucid is not very well protected against over-expanding the static memory area and may crash if you expand beyond the available process memory space.

(free-array `arr`**)**

This function is similar to the `free` function in C. Frees storage space for an array that has been allocated using allocate-array. After evaluating (free-array A), it will still be possible to access A, but the contents may be overwritten at any time. It is also possible to free sub-arrays of an allocated array. This function gives a warning if the argument is not an array, or if it was not allocated from the static heap.

(with-static-arrays `arr-list . body`**)**

This macro acts just like the Common Lisp `let*` macro except that all of the static-arrays bound by it are freed (silently) before exiting. If a let* was used, the local variables would become unbound on exiting, but the static storage space would not be freed.

(expand-heap `type-or-heap &optional size return-block`**)**

Expand the static heap of the given type (can also pass the heap itself) by the given

number of elements. Any valid `:element-type` argument for the argument of the Common Lisp make-array function may be used as a type. Note that image arrays are of type `'single-float`. Default value for `size` is `*heap-growth-rate*`. If `return-block` is nil (the default), returns t. Otherwise returns the newly allocated static array.

(heap-status)
Print out information about the static array allocator. For each type of static heap, prints the number of used elements, the number of free elements, the size of the largest block, and the number of contiguous blocks (i.e. fragmentation).

(run-heap-diagnostics)
Use this to verify the integrity of the static heap memory allocator. Prints out all free arrays, giving warnings if inconsistencies are found.

(ogc &key verbose)
This is the OBVIUS garbage collector, which can only be called from top-level. The `scrounge!` macro provides a top-level interface (it just calls ogc with keyword `:verbose` non-nil). When keyword `:verbose` is non-nil, it prints out information about the space reclaimed, the current heap status, and the (preserved) named viewables. `ogc` preserves all static arrays that are owned by pictures on picture stacks, and *non-orphaned* viewables (see the discussion in the next section). That is, all viewables that are displayed as pictures, all viewables that are bound to symbols in the current package or the user package, the viewables on the list `*protected-viewables*`, and all of the inferior and superior viewables of the aformentioned viewables (phew!). All arrays that are not preserved are returned to the heap for reallocation. Therefore, it is VERY IMPORTANT that the pointers from pictures to viewables, and from viewables to their superiors and inferiors be maintained. In addition, users should be discouraged from creating their own data structures containing "inaccessible" viewables, since these will not be preserved during a garbage collection. ASIDE: If Lucid ever provides a finalize-instance method (called when all references to a CLOS object have been lost and it is about to be garbage collected) we will be able to get rid of this stuff and just have the finalize-instance on viewables free their static arrays!

Orphans and OGC

In order to keep track of which allocated arrays are needed, OBVIUS contains a fairly large amount of code devoted to maintaining bi-directional pointers between viewables and pictures, and between viewables and their superiors and inferiors. For example, when viewables are destroyed, their static space is de-allocated, and their pictures are destroyed. At any given time, OBVIUS can access viewables through their pictures (which must reside on panes), through the Lisp symbol-table (if they have symbol names), and through their inferiors and superiors. Any viewable that is not accessible in this manner is called an "orphan". The OBVIUS garbage collector, `ogc`, will reclaim all allocated memory of orphans.

There are three standard situations in which an existing viewable can become an orphan. These are described in the section on "orphans" (in section 3). If the parameter `*auto-destroy-orphans*` is nil, then the static space belonging to these orphans will remain allocated. In this case, the only way to recover the static space belonging to these orphans is by running the OBVIUS garbage collector `ogc` (described below).

If the global parameter `*auto-destroy-orphans*` is non-nil, the orphans created in these three situations will be *automatically destroyed*. This is a little bit dangerous for new users: it is easy to write code that that locally creates orphans that one later expects to use. For example, you create a temporary compound-viewable (e.g., an image-pair), you bind a local variable to one of its inferiors (e.g., the x-component). Later in the function, outside of the scope of the `with-local-viewables` macro, you perform computations on the inferior. But this inferior was destroyed upon exiting the `with-local-viewables` macro! You can prevent such destruction of orphans by 1) setting `*auto-destroy-orphans*` to nil, or 2) pushing the orphan onto the list in the special variable `*protected-viewables*`. The `with-local-viewables` macro places its return value(s) on the `*protected-viewables*` (see the definition, in **viewable.lisp**.

It is important to understand, however, that setting `*auto-destroy-orphans*` to a non-nil value only ensures that non-orphan viewables that become orphaned will be destroyed. *It is still possible to create orphans.* For example, suppose the user writes a function that returns a list of viewables. If I call this function, and I don't hold onto the list returned by the function, then these viewables will be orphans: OBVIUS will have no way of accessing them. But OBVIUS cannot "know" about the creation of such orphans and thus cannot automatically destroy them. As before, the only way to recover the static space belonging to these orphans is by running the OBVIUS garbage collector `ogc` (described below).

Note also that this is a double-edged sword: if I call the function and bind the resulting list to a global variable, then these viewables are not accessible by OBVIUS (and are thus considered orphans), but I (the user) *still have access to them* (through the global variable). Calling `ogc` will therefore *de-allocate the storage belonging to these viewables.* The user must therefore be careful not to create data structures containing "hidden" (orphaned) viewables. To return multiple viewables from a function, you should use the Common Lisp multiple-values facility. See section 3.4 on memory management for more information.

Troubleshooting Memory Problems

If you find that viewables are being destroyed or over-written when you don't want them to be destroyed, there are severa things to examine. First, try running your code with the variable `*auto-destroy-orphans*` set to nil. If this fixes the problem, then you must figure out where in your code you are "orphan'ing" viewables (see the previous section). You should get warnings when you orphan viewables and the `*auto-destroy-orphans*` is nil.

You may also be creating new viewables that are orphans upon creation. In this case

the problems would not occur until you run the OBVIUS garbage collector (`ogc`). Look for places in the code that store viewables in top-level data structures that OBVIUS doesn't know about (i.e., viewable-matrices are ok, but a list of viewables is not).

Finally, if the viewables your are losing are *not* orphans, it must be that OBVIUS does not know about the static arrays of those viewables. Make sure that the `static-arrays-of` method returns alist of all of the appropriate arrays. These are the arrays that will be protected from garbage collection.

Further memory problems may arise because of foreign function calls. If you call a foreign function with non-static Lisp data structures, and the garbage collector is activated while you are in the foreign code, the Lisp data structures will be moved by the garbage collector. The foreign function, however, will still have the pointer to the old data structure and may continue to read or write data from the old memory locations. There are several situations in which the garbage collector might be called while in foreign code:

1. if the foreign code performs callbacks (that is, calls a Lisp function)
2. if the ephemeral-garbage-collection (egc) is enabled
3. if the foreign code calls `malloc`

Even worse, this behavior may be different depending on whether your code is interpreted or compiled, and which which version of the compiler (i.e., production or development) you used!! If your def-foreign function definitions are *interpreted*, egc will often run while you are in the C code!

The programmer should be aware of these situations, since these type of bugs are often not noticed when testing code on small examples. Once the code is in place and being used, these problems are very difficult to track down! Here are some possible solutions to the problem:

1. write lisp interface code that *always* passes static objects to your foreign code. You can use the OBVIUS `allocate-array` function to allocate static arrays.
2. Avoid the three items above that might cause garbage collection in lisp while executing foreign code. This means that all temporary space used in the C code must be *allocated by Common Lisp* and passed in to the foreign code.
3. Wrapping a `with-interrupts-deferred` macro around calls to the foreign code can prevent garbage collection due to the second and third items above.

9.3 Adding New Viewable Subclasses

Use the CLOS function `defclass` or the OBVIUS function `def-simple-class` to define new types of viewables. `def-simple-class` just calls defclass with args to set up the accessors

automatically. The following functions/methods must be provided for each viewable (for an example, look at the file **image-pair.lisp**). In some cases, the default methods (defined in the file **viewable.lisp**) inherited from the generic viewable class will suffice. Typically, viewable classes and their methods are placed in a file called `<viewable-class>`.lisp.

`(def-simple-class <viewable-class> ...)` The class name is usually exported.

(settable-parameters `viewable-class-name`**)**
: This method, which should dispatch on a symbol eql to the viewable class name, should return a list of the class slots which are user adjustable. It is used by the user interface code to create the dialog boxes for editing the viewable slots. Note that it should append a list of the settable slots for this class with the result of a call to the inherited method, since the superclasses may also provide settable slots. The default method returns `'(display-type)`.

(initialize-instance `vbl &rest initargs`**)**
: This is the standard CLOS initialization function. Any static arrays needed by the viewable should be allocated here using the `allocate-array` function (described in the previous section). This method must also fill the superiors-of slots of the inferior viewables if the viewable is a compound viewable (ie. contains other viewables).

(set-result `vbl model`**)**
: This method is called by the `with-result` macro (which should be used by all functions that return viewables) as described above in section 9.1. If the programmer is planning to write routines for performing computations on instances of the new viewable class, they should provide set-result methods to standardize the result argument (`:->`) behavior. The default methods are defined in the **viewable.lisp** file. There are four standard method dispatch cases:

  1. Both arguments are viewables. In this case, the method checks that the viewables are of the same type, and calls `check-size` to make sure they are the same size. The default method is usually sufficient here.
  2. The `vbl` argument is a name (nil/symbol/string), `model` is a viewable. This method creates a new viewable, as specified by the `model`, with the given name. It should copy relevant slots from the model into the result. This method may need to be redefined for viewable subclasses.
  3. The `vbl` argument is a viewable, `model` is a property list (plist). In this case, the user has passed a viewable as a result argument, and it must be checked for consistency with the initargs specified by the plist. This should be redefined for viewable subclasses to do this error-checking.
  4. The `vbl` argument is a name (nil/symbol/string), and the `model` is a plist. A new viewable is created using the plist as a list of initargs in a call to the CLOS function `make-instance`. The default method defined in **viewable.lisp** is usually sufficient for this case.

(make-¡viewable-class¿ `&rest initargs`)

The top-level constructor function for the viewable. This function creates an instance of the viewable and fills in its slots. This is redundant – the user could just as well call the CLOS function `make-instance`. It is provided to make the initargs explicit (see, for example, the definition of `make-image`). It should make use of the functionality captured by the `initialize-instance` and `set-result` methods, by calling `with-result`. This function should be exported.

(¡viewable-class¿-p `thing`)

The predicate for the class. Typically a macro. This should be exported.

(print-object `vbl stream`)

Use this to modify the printed form of this class.

(print-values `vbl`)

This should print out "values" of the viewable. It need not be defined for all viewable subclasses.

(check-size `vbl &rest vbl-list`)

This method is called by set-result (case 1) and should ensure that a list of viewables has comparable "dimensions" so that point operations performed on them will work. The meaning of "dimensions" depends on the sub-class of viewable. The method should return a viewable if the sizes are consistent, and signal an error otherwise. See example in image.lisp. The default method does no checking and returns the first argument.

(static-arrays-of `vbl`)

A method that returns a list of the static arrays allocated when the viewable was created. The list should *not* include static arrays belonging to the inferiors of the viewable. These arrays will be freed when the viewable is destroyed. They are also the arrays that are preserved when the OBVIUS garbage collector (`ogc`) is called. The default method returns an empty list (nil).

(set-not-current `vbl`)

This method is called when the viewable when it is destructively modified. It is called by the `with-result` macro. This method does not usually have to be written as the inherited one is usually sufficient. The inherited method increments the `current` slot and calls `set-not-current` on the superiors and pictures of the viewable (see viewable.lisp).

(inferiors-of `vbl`)

A method that returns a list of the inferiors of the viewable. This must be defined for all compound viewables: it is used by the `destroy` and `ogc` functions. The default (inherited) method returns nil, and is thus appropriate for atomic (non-compound) viewables.

(notify-of-inferior-destruction `vbl`)

A method that is called on a compound viewable when one of its inferior viewables

is being destroyed. The default (inherited) method signals a continuable error. If continued, it destroys both the superior viewable and the inferior (see viewable.lisp)! Some viewables may allow destruction of inferiors, but they should erase all pointers to those inferiors so as not to confuse the system memory manager. See the examples in **sequence.lisp** and **gaussian-pyramid.lisp**.

(notify-of-superior-destruction `inf-vbl sup-vbl`**)**

This method is called on a viewable when one of its superiors is being destroyed. It need not be rewritten for viewable subclasses. The standard method removes the superior from the `superiors-of` list, and if `*auto-destroy-orphans*` is non-nil and the `inf-vbl` has been orphaned, it will be destroyed.

The description of the functions and methods above should make it clear that the objects in OBVIUS are often connected by bi-directional pointers. This bi-directionality is necessary for the proper functioning of the OBVIUS garbage collector (`ogc`). These must be maintained every time an object is created or destroyed. This is especially difficult in the case of compound viewables such as complex images and image sequences. In order to localize the source of such problems, it is important that the `initialize-instance` method set up these pointers properly, and that viewables should only be destroyed using the standard destroy function (defined in the file viewable.lisp). The superiors-of and pictures-of slots must be carefully maintained. The superiors-of slot is used by set-not-current, destroy, and the garbage collector. The inferiors-of slot (method) is used by destroy and the garbage collector. The pictures-of method is used by destroy and the garbage-collector.

(present `pic viewable pane`**)**

Described in section 9.6.

Inferior Accessors

In general, compound viewables should provide accessor methods to access their sub-viewables. The convention is that these take a `:->`keyword argument. If no result is passed, they return the sub-viewable, in the style of a standard accessor function. If a result is passed, the sub-viewable is copied into that result. If a symbol is passed, a new viewable (bound to that symbol) is created and the sub-viewable is copied into it. For examples, see the definition of `frame` in **sequence.lisp** or `x-component` in **image-pair.lisp**.

History

The `history` slot of a viewable contains the a history object. This object preserves information about how the viewable was created. It keeps track of the symbol names of intermediat viewables, so that the user can print out an abbreviated version of the history. The history of a viewable is set with the `set-history` function: `(set-history vbl &rest hist-list)`. This destructively replaces viewable elements of hist-list by their histories. This function is normally called by the `with-result` macro.

9.4 Multi-Processing

Starting with version 2.2, OBVIUS takes advantage of Lucid's multi-processing capabilities. The main lisp process runs a top-level "listener" or "read-eval-print-loop" (`repl`), described below. The source code for this is in the file **repl.lisp**.

A secondary process, the `event-dispatch-process` created by LispView, handles window system events (mouse clicks, window exposure, etc). NOTE: if this process dies, you will no longer be able to use the mouse. If you encounter an error and are given a choice of killing this process, AVOID DOING SO!

Mouse Events

In order to avoid data-access collisions between processes, OBVIUS mouse events are handled in one of two ways:

1. If the a mouse event requires processing that will only effect the display (i.e., pictures and panes), and the processing is relatively simple (e.g., requiring less then one second), then that event is handled immediately by the mouse process. In order to ensure that no other process tries to access the same windows, all functions that are to be executed in immediate mode should be called within the context of the macro `with-locked-pane`. This macro sets up a "lock" on the a given pane, such that other processes executing a `with-locked-pane` macro will be forced to wait before accessing the pane.

2. More extensive event handling is placed on a queue, for execution by the listener's read-eval-print-loop (described below). This queue is executed in FIFO (first in first out). Execution will be delayed if the listener is not at top-level.

Note that if there is an error, the debugger inhibits multi-processing and therefore mouse events will not be handled.

Repl

The OBVIUS read-eval-print-loop has several purposes:

- Reads forms from *standard-input*, evaluating them and printing their value(s) to *standard-output*. If the value(s) begin printed are viewables and `*auto-display-viewables*` is non-nil, then these viewables are automatically displayed on the current pane.
- Sets the standard Common Lisp variables, `*`, `**`, `***`, `+`, `++`, `+++`, `-`, `/`, `//`, `///`, as described in Steele.
- Handles enqueued mouse events (see above).

- Sends messages to the Status line in the OBVIUS control panel, indicating when the listener is at top level, computing, or in the debugger.

Adding Menus and Dialog Boxes

The menus on the control panel are cynamically computed at "click-time". This makes their reponse time slower, but makes it very easy to modify them.

Adding new functions to the function menus is easy. You need only add the function name (a symbol) to one of the global variables `obv::*obvius-<type>-functions*` where `<type>` is one of {`stats, arith, geom, filter, compare, synth, matrix, misc`} . These menus are recomputed at "click-time" from these variables, so modifying will result in an immediate change in the menu. Function dialog boxes are computed automatically (although this is sometimes a bit flakey).

The Global parameters dialog is also computed at "click-time", and will contain all exported symbols beginning and ending with an asterisk () character.

Similarly, if you define a new viewable or picture sub-class, it will be automatically added (in alphabetical order) to the "Viewable defaults" or "Picture defaults" menus.

The "Modules" sub-menu (on the "misc" menu), is created on the fly to contain a list of all currently unloaded modules. If you load a module, it will no longer appear on this menu. If you define a new module (by adding to the obv::*obvius-module-plist*), it will appear on this menu.

9.5 Display of Pictures

The most important convention in OBVIUS is the chain of command for display of viewables, and manipulation of picture-stacks. This is illustrated in figure 1. Typically, the top-level listener loop (**(repl,)**
described above) will call the `display` function when viewable(s) are returned to top level. Other code (e.g. a function called from the mouse process) can achieve the same behavior by calling the function `print-top-level-values`.

Since OBVIUS allows destructive modification of viewables, it often happens that a picture will become out-of-date with respect to its viewable. OBVIUS keeps track of this by maintaining a `current` slot for viewables and pictures. This slot holds a version number. If the version number of the picture is equal to that of the viewable, it is up-to-date. Software that modifies a viewable should call `set-not-current` (or use the `with-result` macro). `Set-not-current` increments the `current` slot of the viewable. Below, decribe some of the functions used to manipulate and display pictures of viewables, and the use of the `current` slot.

(display `viewable &optional display-type &rest initargs &key pane make-new`**)**

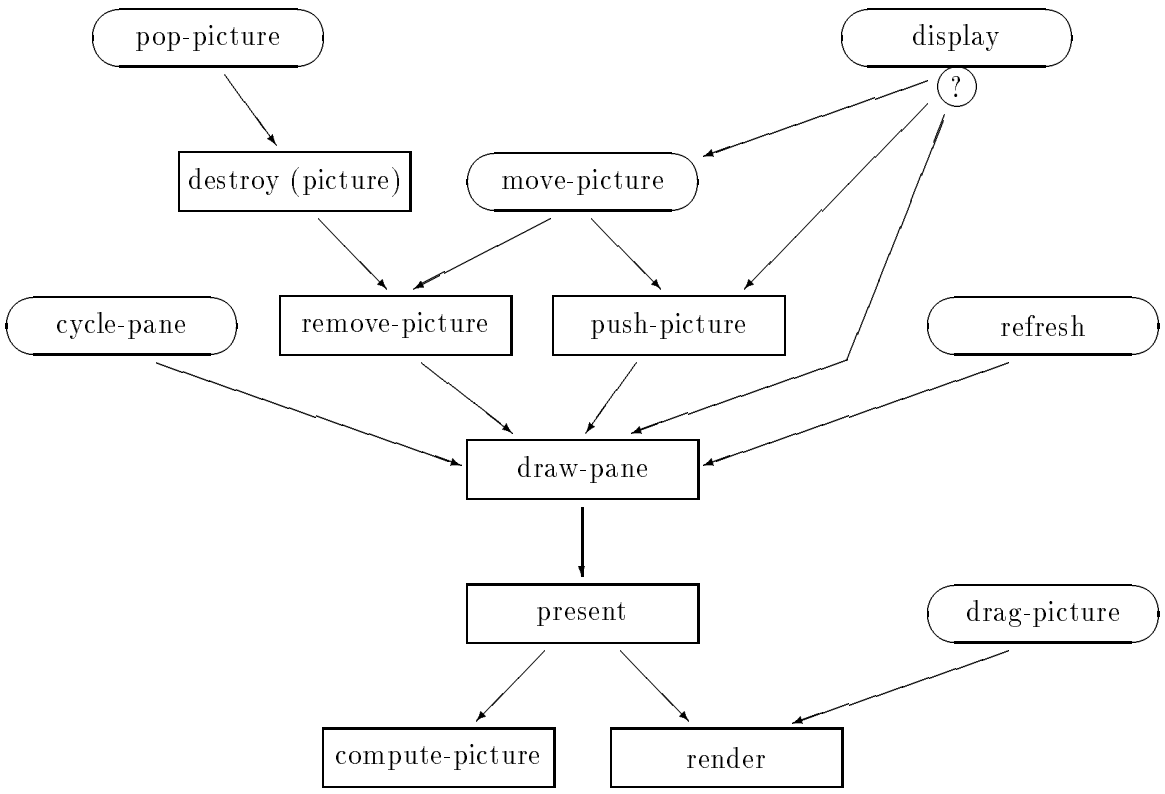

Figure 1: Chart indicating the function call hierarchy of functions used to manipulate the picture stacks of OBVIUS panes. Functions in oval boxes are top-level functions. Those in rectangular boxes are internal. The question mark enclosed in a circle indicates that the `display` function calls one of the three indicated functions.

This is the exported method for creating a picture of a viewable. This assumes *only* that the viewable exists. The default value for the :pane keyword is the current pane. If there are no panes, the function creates one using new-pane. The default value for the :display-type keyword is the display-type slot of the viewable. If this :display-type keyword is nil, then nothing is displayed. Otherwise, the value of the :display-type keyword must be a picture class (or the name of one) suitable for passing to make-instance. When the :display-type keyword is non-nil, OBVIUS searches through the picture stacks for a picture of the specified type. If no picture of this type exists or if the `:make-new` parameter is non-nil, a new one is created using `make-instance`. The `initargs` are passed along to make-instance, so the user can specify initialization arguements for the picture. If a picture of this type exists on one of the panes, it is moved to the top of the current pane, if the variable `*preserve-picture-pane*` is non-nil. Otherwise it is moved to the top of its own pane. The picture will be updated if it is out of date with the viewable and the variable `*auto-update-pictures*` is non-nil. In either case, the picture is put onto the pane using either `push-picture` or `move-picture`. If it is already displayed in the desired pane, `display` just calls `draw-pane`.

(refresh `&optional (pane *current-pane*))`
This provides an entrypoint to the draw-pane method that *always* re-computes the internal representation of the picture, whether or not it is out-of-date with respect to the viewable.

(move-picture `pic &optional (pane *current-pane*))`
This function performs bookkeeping manipulations of the picture-stack of the pane and the pane-of the picture (i.e., it is responsible for keeping consistent pointer references). If the picture is being moved to another screen, then its `system-dependent-frob` may have to be converted to be consistent with the new type of screen. It then calls `draw-pane` to update the screen.

(cycle-pane `pane &optional (num 1))`
This function performs bookkeeping manipulations of the picture-stack of the pane, rotating the list `num` places (note: num can be negative). It then calls `draw-pane` to update the display.

(pop-picture `pane)`
This function calls `destroy` on the top (visible) picture of the pane stack.

(draw-pane `pane &key clear)`
This method is the bottleneck for bringing the appearance of the pane up-to-date with the picture on top of the stack. In addition to being called the functions illustrated in figure 1, it is called internally by the window system when it receives "damage" or "exposure" events. If the `:clear` argument is non-nil, `draw-pane` clears the pane and sets the title-bar to the string `"displaying ..."`. When the pane is not empty, `draw-pane` calls the `present` method (described in the next section) on the picture,

viewable and pane, and then sets the title bar string. If an error occurs in the call to `present`, `draw-pane` will pop the bad picture from the picture-stack when the user returns from the debugger.

9.6 Adding a New Picture Subclass

Pictures are typically created by the `display` function (defined in the file **viewable.lisp**). Note that joint methods on viewables and pictures are defined in the picture file by convention since the viewable files are loaded first.

In creating a new picture type, the programmer should first write the low level code that fill's the `system-dependent-frob` slot. This slot holds a system-dependent representation of the picture that can be efficiently "rendered" to the screen or printer. For example, the frob might be an X window system offscreen bitmap, or an object containing lists of lines that need to be drawn. This abstraction also makes porting to other window systems or display devices easier.

The `drawable` picture subclass, which provides a parent class for `graph`s and `vector-field`s is a good example. The frob for this class will be either an offscreen bitmap object (called a bltable), or an object containing a description of lines to be drawn directly in the pane. The `retain-bitmap` determines which of these types of frob is used. The code in **graph.lisp** draws the graph into the frob using `draw-line` and `draw-text` commands, which are defined for both type of frob. Note also that these are described for `postscript-bltable` frobs, which are used when sending output to a postscript printer.

After writing the low-level code that draws the picture, the programmer must connect the picture with the rest of OBVIUS. In order to do this, the following functions must be provided for each new picture subclass (for examples, see the files **picture.lisp**, **gray.lisp**, and **drawing.lisp**):

(defclass `<picture-class> ...`**)**
The class name is usually exported.

(settable-parameters `picture-class-name`**)**
This method, which should dispatch on a symbol equal to the picture class name, should return a list of the class slots which are user adjustable. It is used by the user interface code to automatically create dialog boxes for editing pictures. Note that it should append a list of the settable slots for this class with the result of a call to the inherited method, since the superclasses may also provide settable slots.

(reset-picture-defaults `pic vbl &rest initargs`**)**
This method should compute appropriate values for all of the picture slots. It is called by the CLOS method `shared-initialize` which is called by both `initialize-instance` *and* `reinitialize-instance`. That is, it is called whenever the slots of the object are being set. In order to modify slots of a picture, the user will typically either

use a dialog box, or call `setp`. Both of these call `reinitialize-instance`. The `reset-picture-defaults` method should take care of all automatic computation of parameters (e.g., if the user sets the parameter to `:auto`), and should also handle parameter dependencies (i.e., when the modification of one parameter requires adjustment of another). *It must return a modifed list of initargs.* See, for example, the method for gray pictures.

(set-not-current `pic`**)**
Called when the underlying viewable is modified. The default (inherited) method puts a double asterisk (ala emacs) in the title bar if the picture is at the top of a picture stack. Specialized versions may need to destroy certain internal data, or reset certain parameters. See, for example, the method on gray pictures.

(static-arrays-of `pic`**)**
Returns the static arrays used by the picture. Note that if any of the arrays are shared with a viewable, these should *not* be returned. They should be returned only by the static-arrays-of method on the viewable (for example, see the code for gray pictures of images, which resets the :scale and :pedestal slots). The standard method just returns the `static-arrays-of` the `system-dependent-frob`.

(drag-picture `pic dy dx`**)**
This method is called while the user is dragging the picture with the mouse. It should adjust the `x-offset` and `y-offset` slots of the picture and redraw the picture in the pane. The default one does this by calling draw-pane, which is inefficient in most cases. Therefore, efficient versions are typically written for subclasses.

(zoom-picture `pic factor y x`**)**
Called by the zoom mouse clicks. `factor` is the amount to zoom by, and the `y` and `x` parameters indicate the location of the mouse when it was clicked. The default version in **picture.lisp** is often sufficient.

(position-message `pic vbl pane x y`**)**
This should present useful information to the user when they click the mouse on the pane at the given coordinates. It typically calls `pane-coord-to-viewable-coord` to compute the viewable coordinates, and then `status-message` to put the message into the OBVIUS status line. The default version does nothing.

(frob-coord-to-viewable-coord `pic frob-y frob-x`**)**
This should convert the coordinates of the frob (these are typically the same as the pane coordinates, but with the offset and zoom removed) to those of the viewable. It is called by the `pane-coord-to-viewable-coord` method.

(title-bar-string `pic`**)**
Returns a string to be put in the title bar of the pane displaying the picture. This should give to the user any information necessary for proper interpretation of the visual display. The default version returns the name of the viewable.

(present `picture viewable pane`**)**

This is a triple-method on picture, viewable, and pane sub-classes which does the actual displaying. The pane dispatch is meant to allow specialization for different types of screen. The typical method computes the internal system-dependent picture representation by calling the `compute-picture` method, and then calls `render` to display the picture in the pane. See **gray.lisp** for an example.

(compute-picture `picture viewable`**)**

This method computes the window-system-dependent intermediate picture representation (the "system-dependent-frob"). It typically calls a system-dependent picture-specific method `draw-<thing>` to actually fill the frob. Note that `compute-picture` is called by flipbooks and pasteups on the sub-pictures. See **gray.lisp** and **x-blt.lisp** for an example of how this is done.

(render `pane frob y-offset x-offset zoom`**)**

This method displays the system-dependent intermediate picture representation (the "frob") in the pane, with the given x-offset, y-offset and zoom.

9.7 Porting To Other Machines, Window Systems, or Lisp Implementations

[***** This section should be elaborated. Describe the strategy of the window-system interface layering. In particular, the generic screen, pane, and frob stuff. *****]

Porting to another window system would require replacing the source files beginning with an "x-" prefix, and would probably require a moderate amount of work. We have deliberately kept our use of LispView simple, and have tried to provide a layer of generic interface to the window system. All references to LispView code are package-prefixed with the lispview: prefix. The generic OBVIUS window system interface is accessed in the OBVIUS source files (e.g. **pane.lisp** and **drawing.lisp**). It relies on four system-dependent types of object: screen, pane, drawable, bltable. The screen and pane objects should inherit from the system-independent versions defined in pane.lisp (see, for an example, the definitions of x-screen and x-pane in x-window.lisp). The function make-pane should be provided. The drawable and bltable objects do not inherit from any system-independent objects, but make-bltable and make-drawable functions must be provided.

As for porting to another Lisp implementation, there are three Lucid-specific facilities which are heavily used in OBVIUS: 1) the foreign function interface, and 2) the multi-processing capability. We also make extensive use of Lucid's CLOS implementation, and 3) the static array allocation facility. Calls to lucid-specific functions are package-prefixed with `LCL:` (general lucid extensions), `CLOS:` (Lucid's implementation of CLOS), or `MP:` (Lucid's multi-processing facility). Changing the syntax of the foreign function calls (assuming the alternate Lisp Implementation provides a foreign function interface) would probably be relatively simple. The multi-processing is actually only used indirectly through the LispView interface to X. Only the code in **repl.lisp** relies on the multi-processing capabilities of Lucid

(which are modularized in the MP package). Porting LispView to run in another Lisp would probably be difficult. It might be easier to rewrite the window system interface from scratch (see above).

Porting to other machines (assuming the X window system and Lucid Common Lisp are both available) necessitates either porting LispView to the new machine, or providing your own X windows interface. Rumor has it that Unipress is planning to port LispView to other hardware platforms, but it is not certain when this will happen.

A Installation

This section gives step-by-step instructions for installing OBVIUS on:

- **Sun workstations**

  With Lucid (Sun) Common Lisp 4.1, with CLOS and LispView. Currently, OBVIUS runs comfortably on Suns with 8 bit frame buffers (color or gray-scale monitors), or monochrome (1bit) frame buffers.

- **SGI workstations**

  With Lucid (SGI) Common Lisp 4.1 (beta) with CLOS. In order to run OBVIUS, you will need at least this version of Common Lisp from Lucid; older versions like version 3.0.2+ will not work. The current version of OBVIUS for this platform was developed on both the Indigo and the Personal Iris. It supports both 8 and 24 bit frame buffers.

A.1 Requirements

You need a fairly large swap space (virtual memory, paging area) in order to use OBVIUS to its fullest potential (we generally allot 50 Megabytes or more). 20 Megabytes is a bare minimum to get OBVIUS to load; this amount of swap space will not allow you to do very much in the way of image processing as the image storage space restrictions will be overly stringent.

If you are accustomed to working with Lisp, these requirements will not be very remarkable. If they do seem somewhat voracious, remember that if you do image processing with individual binary programs and shell scripts, as many people do, you will end up storing all of your results as files on the disk and may soon frequently require easily this much storage space for your intermediate, temporary results.

The OBVIUS directory, which contains Lisp and C source and binary files and documentation, occupies around 4Mb of disk space. The complete lisp image (including Lucid 4.0, CLOS, OBVIUS, possibly LispView) will occupy at least 15Mb, so you should have that much disk space to devote to OBVIUS.

In order to make use of the user interface features, you should also have GNU emacs. You can obtain GNU emacs via anonymous ftp from prep.ai.mit.edu.

OBVIUS 3.0 runs under the X window system. X was developed at MIT and there are public domain servers available for a number of different architectures. You can ftp a copy of X from expo.lcs.mit.edu. Or, you can get Openwindows (available from Sun), that includes both the X and News window systems.

You also need LispView, an object-oriented lisp-interface to X, if you are running OBVIUS on a Sun. LispView is available from Sun Microsystems with Sun (Lucid) Common Lisp, version 4.0. LispView is not required for the SGI platform as OBVIUS uses the

native SGI graphics library (GL). The lisp-interface to the GL is included in the OBVIUS distribution.

A.2 Installation Instructions

The symbol `<obv>` will be used throughout this document to indicate the pathname of the primary OBVIUS directory.

1. Decide where the OBVIUS source directory tree (approx. 3-5 Mbytes) will live in your filesystem. We will refer to this directory as `<obv>`. On our machines, this is /usr/local/obvius. Also decide where the OBVIUS executable binary file (the lisp image, approximately 15 Mbytes) will live. We'll refer to this directory as `<bin-path>`. On our system, this is /usr/local/bin. This directory must be in each OBVIUS user's PATH environment variable.

2. Extract the OBVIUS directory tree from the tape or from the compressed tarfile (you have probably already done this!):

   - `mkdir <obv>`
   - `cd <obv>/..`
   - If you are using a tape, insert the tape and wait for the whirring to stop. If you are starting from a compressed tar file (obvius.tar.Z), put the tarfile in the current directory and uncompress it using `uncompress obvius.tar.Z`.
   - For a tape, execute `tar xvf /dev/nrst8`. From a tarfile, execute `tar xvf obvius.tar`.

   The distribution contains several source subdirectories (lisp-source, c-source and emacs-source), a documentation subdirectory (doc), and a set of top-level files (README, INSTALL, lucid-defsys.lisp, etc). You may move the individual source subdirectories anywhere in your file system. Also, decide where you want the Lisp and C binaries to live (see next instruction).

3. In the directory `<obv>`/c-source, there are two makefiles: Makefile-sun4 (for sun 4's) and Makefile-sgi (for SGI's). Edit these makefiles to refer to the appropriate binary pathnames. The default is for the binaries to go in subdirectories sun4-bin and sgi-bin of the `<obv>` directory. Also edit the `INCLUDES` line in these makefiles to refer to the appropriate pathnames for the Xlib.h and X.h include files (this is required for the Sun platform only). Create the necessary binary directories using mkdir.

4. In file `<obv>`/site-paths.lisp, edit the definitions for *obvius-directory-path*, *lisp-source-path*, *c-source-path*, *binary-path*. Usually these will be `<obv>`/, `<obv>`/lisp-source/, `<obv>`/c-source/, and the appropriate binary subdirectory for the given machine (must be consistent with the directory chosen in the previous item). Also, edit the defvar for *obvius-image-path*, and *numerical-recipes-library*.

5. In file <obv>/site-paths.lisp, edit the defvars for *temp-ps-directory* (usually /tmp/), *default-printer* (should be the name of a postscript printer), and *print-command-string* (this will probably need to be changed depending on your printing configuration).

6. Edit the file <obv>/site-init.lisp. This file should load any optional modules that you want to be part of the basic OBVIUS lisp image at your site. The list of modules appears in the lucid-defsys.lisp file.

7. Start up a lisp world that includes CLOS. Use "lisp -n" to start lisp without loading your personal lisp-init.lisp file.

8. If you are running on a Sun platform, load LispView by typing: `(load "<lispview>/lispview.sbin'`
   You can get this compiled lispview file (lispview.sbin) via anonymous ftp from white.stanford.edu.

   If you are compiling LispView from scratch, you must add the following lines to the file "build.lisp":

   ```
   ...
   (in-package "USER")

   ;added by Wilbur
   (proclaim '(optimize (compilation-speed 0) (speed 3) (safety 1)))
   (proclaim '(notinline make-instance))
   ...
   ```

9. To compile and load OBVIUS, type `(load "<obv>/lucid-defsys")` and then type `(obvius::run-obvius)`. This takes a while. Note that if you do not have a lot of swap space, lisp may die during the compilation. Just quit the lisp and start again. It will load the files that have already been compiled and then compile the rest of them.

10. To save an OBVIUS lisp image:

    - You should have already compiled all of the OBVIUS source (see previous instruction), and tested to see that it works (take a look at ¡obv¿/obvius-tutorial.lisp and try evaluating some of the expressions).
    - Start up a lisp world that includes CLOS.
    - Load LispView if you are running on a Sun platform by typing `(load "<lispview>/lispview.sl`
    - Type `(load "<obv>/lucid-defsys")` and then type `(obvius::make-obvius)`. WARNING: We have found that attempting a "disksave" over an NFS connection sometimes fails. If your OBVIUS image does not work correctly, then try making it from your file server.

11. OBVIUS will run without Gnu Emacs, but there are many advantages to running with it. If you will be using emacs, you will need to install the emacs extensions which are included in the tarfile in the emacs-source subdirectory. Copy the directory of emacs-extensions whereever you like in your filesystem. If you prefer, you

can put them in with your standard distribution. We will refer to this directory as <emacs-extensions>. On our machines, this is "<obv>/emacs-source". Add the following lines to your .emacs file:

```
(setq load-path (cons \abox{emacs-extensions} load-path)) \\
(setq *obvius-program* \abox{obvius-image}) \\
(autoload 'run-obvius  "cl-obvius" "" t) \\
```

The symbol <obvius-image> should be the pathname for the lisp image containing OBVIUS. To start OBVIUS from a saved OBVIUS lisp image, run emacs and type M-x run-obvius.

If you have not yet made an OBVIUS lisp image, or want to run from the compiled binary files, you may put the following definition in your .emacs file:

```
(defun run-development-obvius ()
  (interactive)
  (load "cl-obvius")
  (let ((*cl-replacement-prompt*  "OBVIUS-3.0> "))
    (run-cl "<lisp-view>"))
  (setq default-directory "<obv>/lisp-source/") ;VISCI
  (cl-send-string "(load \"<obv>/lucid-defsys\")\n") ;VISCI
  (cl-send-string "(obvius::run-obvius)\n")
  (cl-add-obvius-key-bindings)) ;add bindings to cl-shell-mode and lisp-mode
```

where <lisp-view> refers to the pathname of a lisp world containing LispView and CLOS. To run OBVIUS, you may type M-x run-development-obvius in Emacs.

12. If you want to use the same emacs environment for running Common Lisp, you may want to put the following lines in your .emacs file as well:

```
(setq *cl-program* \abox{lisp-image}) \\
(autoload 'run-cl "cl-shell" "" t) \\
```

where <lisp-image> refers to the pathname for a lisp-image. On our machines, this is in /usr/local/bin/Lucid-4.0. To start up Common Lisp inside emacs, type M-x run-cl. Documentation for the cl-shell environment is in the file <emacs-extensions>/cl-shell.doc. Further extensions to emacs are included with the source, and examples of how to use these may be found in the file <obv>/emacs-source/example.emacs.

B Release Notes

B.1 Version 2.1 Release Notes

There have been several major changes since releases 1.0 and 1.2 of OBVIUS. We have tried to keep most of the changes internal (i.e. non-exported), but some of them will affect the top-level user. The following are the major changes since release 1.2:

B.1.1 General Changes

- OBVIUS now runs under the X11 or OpenWindows window systems exclusively. We no longer support SunView or the Symbolics window systems.
- OBVIUS now runs under Lucid 4.0. This version of Lucid provides an implementation of CLOS (we no longer use PCL), and supports multi-processing.
- OBVIUS now uses a standard interface to the window system called "LispView". LispView was written at Sun, and is distributed with Sun (Lucid) Common Lisp. We have deliberately kept our interface to LispView as simple as possible so that it should be easy to port OBVIUS to other GUI platforms. Note that LispView has a few minor bugs when running under the twm window manager.

B.1.2 Specific Notes for OBVIUS Users

- The following global parameters have gone away: `*SEQ-DISPLAY-TYPE*`, `*DEFAULT-DF-SIZE*`, `*DEFAULT-NUMBER-OF-BINS*`, `*AUTO-SCALE-IMAGES*`, `*GAMMA-CORRECT*`, `*GRAY-SCALE*`, `*GRAY-PEDESTAL*`, `*AUTO-SCALE-COMPOUND-IMAGES*`, `*RETAIN-GRAPHS*`, `*INDEPENDENT-PARAMETER`
  `*DEFAULT-PASTEUP-FORMAT*`, `*AUTO-SCALE-VECTOR-FIELDS*`, `*VECTOR-FIELD-DENSITY*`, `*VECTOR-FIELD-MAGNIFY*`, `*VECTOR-FIELD-LENGTHEN*`, `*VECTOR-FIELD-SKIP*`, `*VECTOR-FIELD-AR`
  In their place, we provide a mechanism for the user to adjust the default values of slots for CLOS classes. The function `set-default` allows the user to set the default values of the slots of any class. The arguments look like: `(set-default <class-name> <slot-name-or-initarg> <default-form>)`. The function `get-defaults` is provided to get a list of the current default values for a class. For example, to auto-scale gray pictures by default, you would execute `(set-default 'gray 'scale t)`. To auto-scale an existing instance of a gray picture, you would execute `(setp scale t)`.
- The user initialization file may now be called either `obvius-init.lisp` or `.obvius`. This allows you to use a "." file if your prefer, and allows old users to keep around their former startup file (previous version of obvius did not use .obvius). Also, the user can now have an `obvius-window-init.lisp` or `.obvius-windows` file. This one will be loaded before the window system is initialized and thus allows modification of the behavior of window system resource allocation etc. Default versions of both of these files are included in the lisp-source directory.

- All pictures now have a `:zoom` slot which you can setp. Zoom factors are no longer limited to powers of two. A zoom value of one corresponds to the "natural" size of the picture. If the zoom value is set to :auto, the picture is automagically zoomed to fill the pane. If it is a list of length two, the picture is zoomed to these dimensions.

- The `:scale` in gray pictures now has a different meaning. The values in the image are *divided* by the scale (after subtracting off the pedestal). The resulting values should be in the range [0,1], where 1 will correspond to the brightest intensity of the display, and 0 the darkest. Values outside this range are clipped.

- New picture type `dither` allows display of dithered images on screens deeper than 1 bit. It inherits from gray.

- `one-d-graph` picture type is now called `graph`.

- New picture type `drawing`. This is now the parent class for `graph`s, `vector-field`s, `surface-plot`s, and `contour-plot`s.

- New slots have been added to screens. The `:gray-gamma` parameter determines the gamma correction exponent for the gray colormap. The `:gray-shades` parameter corresponds to the number of gray colors in the colormap. The `:gray-dither` parameter determines whether OBVIUS dithers float array values into gray pictures. If this holds a number, OBVIUS will dither if there are less gray shades than this number. If t, OBVIUS always dithers. The `:foreground` should hold either a keyword corresponding to one of the colors in `/usr/lib/X11/rgb.txt`, or a list of three numbers in the interval [0,1] corresponding to red, green, blue components. This color is used as a default for graphs, vector-fields, etc. The `:background` slot is similar. It determines the color used to clear the panes.

- Edge-handler arguments to `make-filter` have changed. They are now keywords instead of strings. The current list is: `nil (wrap), :dont-compute, :zero, :repeat, :reflect1, :reflect2, :extend` .

- The "point-operation" macros (`+.  *.  /.` etc) are now act like the corresponding Common Lisp functions (documented in Steele, edition I). They take any number of arguments, along with a result arg. New ones have been defined. Eventually, ALL of the numerical functions in Steele should be created. Partial list of new operations: `fill!`, `make-ramp`, `make-1-over-r`, `make-r-squared`, `expt.`, `exp.`, `log.`, `mod.`, `sin.`, `cos.`, `realpart.`, `imagpart.`, `phase.`, `incf,. decf.`, `zero-crossings`, `rect-to-polar` `polar-to-rect`, `transpose`, `max-abs-error`, `minimum-location`, `point-minimum`, `maximum-location`, `point-maximum`, `skew`, `kurtosis`.

- The `*auto-scrounge!` parameter has been eliminated. A new parameter, `*auto-destroy-orphans*`, has been added to allow OBVIUS to destroy orphaned viewables. This is described in section 3.

- The macro `with-local-images` has been replaced by `with-local-viewables`. The behavior has been slightly modified: 1) the new macro is capable of returning multiple-values, 2) the macro signals an error if you try to return one of the local viewables from the body, 3) you may return inferiors of compound viewables – these will not be destroyed (even if the global symbol `*auto-destroy-orphans*` is non-nil).

- Emacs interface has been rewritten as a much cleaner package (in use since 1/90). Support for arglists, macroexpansion, describe, etc are provided. See the file emacs-source/cl-shell.doc for more information.

- Arguments to display have changed. The new function is much more powerful and flexible. The syntax is as follows:

```
(display viewable &optional (display-type (display-type vbl))
                  &rest keys
                  &key
  (pane (or *current-pane* (new-pane)))
  make-new
                  &allow-other-keys)
```

  The differences are: 1) `:display-type` is now an optional parameter (it used to be a keyword). 2) The `:make-new` parameter allows you to force OBVIUS to create a new picture of the given type (otherwise, OBVIUS may just bring an existing picture of the given type up to the top of the current pane). 3) You can pass keywords corresponding to slot names in the picture (as in setp) to set them up. Example: `(display <image> 'gray :scale 200 :pedestal 37)`

- Arguments to `hardcopy` have changed. They are now similar to those for display:

```
(hardcopy viewable &optional (display-type (display-type vbl))
                   &rest keys
                   &key
   path printer y-location x-location y-scale x-scale
   invert-flag suppress-text make-new
                   &allow-other-keys)
```

  Also, hardcopy now does real postscript output (i.e. line drawing commands) for drawing pictures (e.g. graphs, vector-fields, surface-plots), so these will be drawn at the resolution of the printer!

- Adopted a convention for matrix multiply operations that vectors correspond to row-vectors. We used to automagically interpret vectors as either row or column depending on the operation. Now, you get a continuable error if the method is expecting a column and you give it a row (or vice versa). To work with column vectors, make Nx1 arrays. Or, use the functions vectorize and columnize to convert (via displaced arrays) between row and columns.

- Arguments to `make-sin-grating` have changed (see description in Section 6).
- Arguments to `make-histogram` and `entropy` have been changed (see description in Section 6).
- Keyword arguments to `make-<vbl>` functions can now be any initargs that you could pass to a call to make-instance. They no longer take a `:->`keyword – you should use the `:name` keyword to specify a name. The purpose of the functions is to 1) emphasize that some args are required, 2) provide result arg behavior (including history, etc).
- You can now create sequences of any type of viewable (not just images) and can display flipbooks of most types of pictures.
- Big speedup in most synthetic image generation code. (eg. `make-uniform-random-image`, `make-sin-grating`, etc).
- Optional loadable modules: look at `*obvius-module-plist*`. Current modules are `:gaussian-pyramid :qmf-pyramid :overlay :x-magnifier :matrix`. Load these by executing `(obv-require <module-name>)`.
- The `overlay` module allows you to display two graphs on one plot, or overlay a one-bit image on top of a gray image. See section 5.14.
- Mouse clicks are now received by a secondary process. Some of them are handled immediately (i.e., even when the top-level listener is computing something). Examples are the simple manipulations of panes: cycle-pane, move-to-here, etc. Other operations are put onto a queue for evaluation by the top-level (listener) process. See section 9.4 for more details.
- Mouse bindings have changed slightly. The right mouse button now displays position messages dynamically as you drag the mouse. Bindings with the M-Sh and M-C buckys down are now picture-specfic and will change depending on the type of picture in the pane. See section 8.1 for a listing of the bindings.
- We now provide a more interactive user interface: a control panel with menus. To use it, execute `(obv-require :x-control-panel)` and then `(make-control-panel)`. Status messages will appear in the control-panel if it exists, in the Emacs mini-buffer (if running in cl-shell-mode) or not at all. Dialog boxes provide pop-up help: click M-left on an item for pop-up help information (NOTE: due to a bug in LispView, this feature doesn't work sometimes. To "reset" it, click in a blank region of the dialog, and then click M-left on the desired item immediately afterward).
- New viewable, `polar-image` stores the magnitude and complex-phase of a complex-image. The functions magnitude and complex-phase do the right thing on polar images. The arithmetic functions (e.g., `add`, `sub`, `mul`, and `div`) are not yet defined correctly for `polar-images`. Convert them to complex images first (using `polar-to-complex`), and then do the arithmetic. `polar-to-rect` renamed `polar-to-complex`, and converts a polar-image to complex-image. Likewise, `rect-to-polar` renamed `complex-to-polar`.

- Defined nearly all image operations on arrays. Redefined image methods to call these array methods.
- Defined nearly all image operations on discrete-functions. Got rid of peak. Now it is same as image operations, maximum and maximum-location.
- Changed args to `circular-shift`, `paste`, and `crop` to be keywords. Defined circular-shift, paste, and crop for vectors and arrays. For vectors, ignores the "y" keywords.
- Added "x" "y" "x-dim" and "y-dim" keyword arguments to `correlate` for correlating over a restricted range of offsets.
- Defined complex-discrete-functions. All the operations for complex-images are implemented for complex-discrete-functions, including fft, magnitude, etc.
- One-d-images can now be made by specifying only the 1 relevant size, e.g., `(make-sin-grating 64)` One-d-images are now stored as vectors (not as 2D arrays).

B.1.3 Notes for OBVIUS Hackers (i.e. non-exported changes)

- OBVIUS now uses LispView, an object-oriented Lisp-to-X interface written by Sun. Most of the old X/SunView cruftiness has been thrown out. The new interface is slower, but it is also more robust and requires much less maintenance!
- Lucid-4.0 contains a nearly-up-to-spec version of CLOS. In particular, defclass and initialize-instance now adhere to the spec. We have defined a simple macro called `def-simple-class` which automatically creates accessors, an initarg and an initform (nil) for each slot. We have also moved most of the functionality of the `make-<vbl>` functions into initialize-instance methods, where it belongs!
- To avoid multi-processing collisions, any function which might be called directly from the mouse must be careful to modify the contents of the pane only within a call to the macro `with-locked-pane`. Also, note that because of the multi-processing in Lucid4.0, you can no longer compile when in the debugger. Alternatively, one can enqueue forms for execution by the top-level loop using the function `push-onto-eval-queue`.
- The memory manager has been rewritten to be noticably faster, more general, and more robust. A diagnostic tool is provided to check the integrity of the heap data structures: `obvius::run-heap-diagnostics`. Eventually, we would like to replace this with a Lucid-supported static memory allocator. Two global parameters are provided to adjust the behavior of the memory manager. If `*auto-expand-heap*` is non-nil, the static heap is automatically expanded when OBVIUS needs more memory. The number of elements by which to expand is determined by the variable `*heap-growth-rate*`.
- The `with-result` macro replaces all `with-result-<vbl>` macros. It returns the value of the last expression in the body (including multiple values). It has been rewritten

to use a model specification similar to what you would pass to make-instance. If you pass a model-plist, the first keyword must be `:class` followed by a CLOS class or the symbolic name of one. Also, if you don't pass a history list, it will not modify the history of the result. This allows the programmer to let the history be that resulting from the sub-calls made within a function. See sections 9.1 and 9.3.

- Auto-display is no longer done from the print-object method. It is now done directly by the top level loop (`repl`). In a multi-processing environment, it was too messy to determine whether the printing was being done at "top-level".
- The order of arguments in the methods `present` and `set-result` has changed.
- The keyword arguments used for the `set-result` method on `viewable-sequence` are a bit different. You can now pass a `:sub-viewable-spec` which will be used to create the sub-viewables. You must provide a `:length` parameter with this.
- The `gray-lut` (gray lookup-table) in a colormap is now a vector of length no more than 256 which indexes into the successive gray entries in the colormap. These gray values may be gamma-corrected.
- Some variable names have changed:

status-line-message	-->	status-message
draw-gray	-->	draw-float-array
draw-bitmap	-->	draw-bit-array
new-array	-->	allocate-array
memory-usage	-->	heap-status
growth-rate*	-->	*heap-growth-rate*
scrounge!	-->	ogc

  In addition, the function `default-display-type` is no longer used.
- Patches are loaded from the patches sub-directory. Files in this directory are expected to have filenames like patch-123.lisp. They are loaded at startup time in numerical order, starting with the filenumber specified by the parameter `*starting-patch-file*`. See the file `lisp-source/patches.lisp` for the code that does the loading. We will periodically put new patch files up for anonymous ftp. Users will typically have the choice of ftp'ing a whole new source tree, with the patches installed, or just the patch files. The new source tree will have an appropriate `*starting-patch-file*`, so as not to load unnecessary patches.

B.2 Version 2.2beta Release Notes

Below is a listing of major changes/additions. See the Changelog for detailed changes.

- Color-images and color-pictures added.
- Overlays totally rewritten.

- Viewable-matrices added. Sequences now inherit from viewable-matrices.
- A huge number of matrix utilites added, and new versions of svd, matrix-inverse, regrees, etc.
- Stepit added.
- List-ops added.
- Statistics and numerical-recipes added.
- Plot symbols added to graphs.
- Polar-plots added.
- Scatter-plots added.
- For flipbooks and overlays, picture parameters dialog now allows you to change parameters of the sub-pictures.
- Generic-ops added. When adding new ops, write the methods for arrays and for generic viewables first.
- eliminated gamma-correct from gray (and pasteup) pictures. Wrote new gamma-correct method.
- args to compile-load changed: Now takes an :output-file keyword like the Common Lisp compile-file command.

B.3 Version 2.2 Release Notes

Below is a listing of major changes/additions. See the Changelog for detailed changes.

- Additional keyword args :optimizations and :brief-optimize-message have been added to compile-load. :optimizations is a list that you would pass to (proclaim (optimize ...)). :brief-optimize-message overrides the Lucid :optimize-message arg and prints a one line synopsis of the current compiler optimizations.
- Arglists have been made more consistent for `crop`, `paste`, `make-slice`, and `circular-shift`. Old keywords were left in for back-compatibility.
- Overlays totally rewritten again (see obvius-tutorial.lisp for examples). Overlays is no longer an optional module.
- New version of surface plots, that can be printed using hardcopy.
- Contour plots added (an optional module).
- Fast display-sequence code added.

- Synthetic functions: Eliminated `make-normal-random-image`. Changed names of

  `make-uniform-random-image` --> make-uniform-noise

  some functions: `make-gaussian-noise-image` --> make-gaussian-noise

  `make-random-dot-image` --> make-random-dots

- `load-image` on image sequences now checks for a `class` key. Useful for image-pairs or color-images.

- `fft` now takes a `:dimensions` keyword.

- Added display for filters.

- Default args to `point-operation` have changed. The default binsize is now nil (=¿ call the function, rather than making a lookup-table). This is slower, but gives no sampling surprises in the result!

- Added :center, :post-center, and :pre-center keyword for fft and power-spectrum.

- Added `rotate` and `resample`.

- Added `round`. `truncate`. `floor`. methods

- The `warp` code is no longer an optional module (it is included in the source).

- Pane manipulation functions now exported (useful from within programs, or if mouse dies): `pop-picture`, `cycle-pane`, `get-viewable`.

- Added function `next-pane`.

- Got rid of `destroy-top-picture` (same as `pop-picture`), and `destroy-pane` which is just `(destroy *current-pane*)`. Got rid of `destroy-top-viewable`. Provided `current-viewable` instead, which returns viewable on top of current pane.

- Added `purge!` function.

- Fixed `setf iref` and lots of other annoying little bugs.

- Added `periodic-discrete-functions` and `log-discrete-functions`.

C Known Bugs

This appendix is a partial list of known bugs in Obvius. Other bugs are listed in the file `<obv>/TODO.soon` and `<obv>/TODO`. They will be fixed in a future release of the system.

1D/2D images have some inconsistencies. In particular, many of the synthetic image generating functions break on one-d-images, and you can not setf iref a one-d-image. We plan to extend Obvius so that images (and all viewables) can be any rank (like arrays).

Hardcopy still needs lots of work.

Multiple Screens are not yet supported.

24bit Displays are not yet supported.

Pyramids do not necessarily work correctly on one-d-images. Qmf-pyramids do not work at all.

Screen color does not update all pictures, e.g., pasteups and graph flipbooks will have the wrong background color.

Index

control panel 79
datfile 17
dialog boxes 96
floating-point exceptions 63
foreign functions 87
    and garbage collection 88, 91
    passing arrays to 88
    floating-point exceptions 45, 87
menus 79
    modifying 96
orphans 12, 20, 89
point operations 108
tutorial 7
*. 45
auto-bind-loaded-images 81
auto-destroy-orphans 108, 109, 12, 13, 81, 86, 90, 94
auto-display-viewables 11, 81, 95
auto-expand-heap 111, 19, 82, 88
auto-update-pictures 15, 81, 98
current-pane 15, 16
default-printer 39, 81
div-by-zero-result 45, 82
heap-growth-rate 111, 20, 81, 88
max-print-vals 23, 81
obvius-module-plist 110
preserve-picture-pane 11, 98
print-command-string 39
protected-viewables 85, 89, 90
starting-patch-file 112
temp-ps-directory 39
x-print-range 23, 81
y-print-range 23, 81
+. 40, 44
-. 44, 45
.obvius-windows 107
.obvius 107
/. 45
:-> 11, 40, 85
:auto
    pedestal 30
    scale 30
    tick-step 35
    x-range 35
    zoom 16
:background 108
:class 86
:destroyed 12
:display-type 109
:edge-handler 26
:foreground 108
:gray-dither 108
:gray-gamma 108
:gray-shades 108
:make-new 109, **15**, 38
:name 110
:reflect1 26
:reflect2 26
:repeat 26
:scale 108
:start-vector 26
:step-vector 26
:sub-viewable-spec 112
:zero 26
:zoom 108
<. 48
<=. 48
<viewable-class>-p **93**
=. 48
>. 47
>=. 48
abs-error **48**
abs-value
    image **45**
abs. 45
access-band 27
    pyramid **61**
access
    pyramid **61**
add-cols
    matrix vector **68**
    vector matrix **69**

add-rows
matrix vector **68**
vector matrix **68**
add 41, 52
bit-image **52**
filter **60**
image **44**
allocate-array **88**, 91, 92
almost-equal **62**
and. 52
append-cols **70**
append-rows **70**
append-sequence **56**
append. 56
apply-filter 26
image **50**
arrow-heads 34
aspect-ratio 35
axis-color 35, 36
back-and-forth 37
background 30
bit-image-p **51**
bit-image 25
bitmap **30**
blur
image **51**
border 32
box-color 33
box-p 33
build
pyramid **61**
change-memory-management **19**
check-size 92, **93**
circular-shift 111, 113
image **46**
clip
image **46**
coerce-to-float **52**
col-dim **68**
viewable-matrix **58**
col-null-space-qr **67**
col-null-space **67**
col-space-qr **67**
col-space **67**
collapse 27
pyramid **61**
color-image-p **57**
color-image **27**
color-picture **31**
color 33, 34, 35, 36
cols **68**
columnize **68**
viewable-matrix **59**
col **68**
complex-conjugate **54**
complex-image-p **53**
complex-image **25**
complex-phase
polar-image **54**
complex-to-polar 110, **54**
compute-picture **101**
condition-number **66**
constructor 13
contour-plot 33
copy 51
bit-image **52**
filter **60**
image **42**
pyramid **61**
correlate 111, **48**
covariance-cols **69**
covariance-rows **69**
crop 111, 113
bit-image **52**
image **47**
cross-product
vector **64**
current-viewable **15**
current 11, 96
cycle-pane **15**, **98**
dct **50**
def-simple-class 91
default-display-type 112
defclass **99**
derivative **50**
destroy-control-panel **79**
destroy 12, 93, 98
determinant-qr **67**

determinant **65**
viewable-matrix **59**
diagonal-image-matrix **58**
diagonal-matrix **65**
diagonal-p **65**
diagonal **65**
dimensions 51
filter **60**
image **41**
sequence **56**
viewable-matrix **58**
discrete-function 24
displaced-rows **68**
displaced-row **68**
display-type 10, 15, 38
display 11, **15**, 38, 96, **98**
distribution-mean-and-variance **55**
distribution-mean **55**
distribution-variance **55**
dither 108, **30**
div 52
color-transform **57**
image **45**
rgb-¿yiq **57**
yiq-¿rgb **57**
domain **55**
dot-product
vector **64**
viewable-matrix **59**
downsample 51
image **46**
drag-picture **100**
draw-line 99
draw-pane 98
draw-text 99
drawable 99
drawing 108
elements **63**
entropy 110, **49**
equal-to 51
image **48**
evaluate **55**
discrete-function **24**
event-dispatch-process 95
expand-filter 26
image **50**
expand-heap **20**, **88**
expt. 45
fft 114, 50
complex-image **54**
image **49**
fill!
image **44**
fill-symbol-p 35, 36
filter-p **60**
filter **26**
first-image **53**, 54
flip-x **47**, 51
flip-y **47**, 51
flipbook **37**
floor. 114
image **46**
foreground 30
frame-delay 37
frame 94
number viewable-sequence **56**
free-array **88**
free 88
frob-coord-to-viewable-coord **100**
gamma-correct
image **45**
gauss-in
image **50**
gauss-out
image **51**
gaussian-pyramid-p **61**
gaussian-pyramid **27**
get-defaults 107
get-viewable **16**
getp **16**
graph-type 34
graph 108, 26, **34**, 99
gray-dither 30
gray-lut 112
gray-shades 30
gray 26, 29
greater-than-or-equal-to
image **48**

greater-than 51
    image **47**
hardcopy 109, 38
heap-status **20**, **89**
hilbert-transform **50**
histogram 24
history-sexp 13
history 10, 94
identity-matrix **65**
identity-p **65**
image-from-array **42**
image-from-bit-image **52**
image-list
    image-sequence **56**
image-matrix-p **58**
image-matrix 28
image-pair-p **53**
image-pair **25**
image-p **41**
image-sequence-p **56**
image-sequence **25**, 28
image 21
imaginary-part **53**
independent-parameters 32, 37
inferiors-of **93**
info-get **11**
info-list 11
info-set **11**
initialize-instance **92**, 94, 99
internal-div 87
interpolate-hex-image **60**
invert
    bit-image **52**
iref 22, 51
    efficiency 23
    bit-image **52**
    image **41**
kurtosis **49**
laplacian-pyramid-p **61**
left-image **53**
left-singular-matrix **66**
length. 56
less-than-or-equal-to
    image **48**
less-than
    image **48**
line-width 33, 35, 36
linear-xform
    image **45**
load-image-sequence **19**, 44
load-image 114, 17, **18**, 44
log-discrete-function 114, 24
loop-over-image-pixels 22, 23, **87**
loop-over-image-positions 23, **87**
low-band
    pyramid **61**
magnitude
    image-pair **54**
    polar-image **54**
make-1-over-r **43**
make-<viewable-class> **93**
make-atan **43**
make-bit-image **52**
make-color-image **57**
make-complex-image **53**
make-control-panel **79**
make-discrete-function **24**, **54**
make-disc **43**
make-filter 108, 26, **59**
make-fractal **43**
make-gaussian-noise-image 114
make-gaussian-noise **43**
make-gaussian-pyramid 27, **61**
make-grating **42**
make-histogram 110, **55**
make-image-matrix **58**
make-image-pair **53**
make-image-sequence **56**
make-image 22, **41**, 52, 93
make-impulse **42**
make-instance 92, 93
make-iterated-image **56**
make-laplacian-pyramid **61**
make-left **43**
make-log-discrete-function **24**
make-matrix **68**
make-normal-random-image 114
make-periodic-discrete-function **24**

make-pinwheel **43**
make-polar-image **53**
make-r-squared **43**
make-ramp **42**
make-random-dot-image 114
make-random-dots **43**
make-saw-tooth-grating **42**
make-separable-filter 26, **59**
make-separable-qmf-pyramid **61**
make-sin-grating 110, **43**
make-slice 113
    image-pair **53**
    image **51**
    sequence **57**
make-soft-threshold-df **55**
make-square-grating **42**
make-synthetic-image-sequence **57**
make-synthetic-image **42**
make-top **43**
make-uniform-noise **43**
make-uniform-random-image 114
make-viewable-matrix **58**
make-viewable-sequence **55**
make-zone-plate **43**
malloc 88
map. **57**
matrix-inverse
    array **64**
    viewable-matrix **59**
matrix-mul-transpose
    array **64**
    viewable-matrix **59**
matrix-mul
    array **64**
    viewable-matrix **59**
matrix-trace **65**
matrix-transpose-mul **64**
    viewable-matrix **59**
matrix-transpose
    array **64**
    viewable-matrix **59**
max-abs-error **48**
max. 49
maximum-location
    image **49**
maximum 36
    image **49**
mean-abs-error **48**
mean-cols **69**
mean-rows **69**
mean-square-error **48**
mean
    image **49**
min. 49
minimum-location
    image **49**
minimum
    image **49**
move-picture **98**
mul 51, 52
    bit-image **52**
    filter **60**
    image **45**
name 10
natural-logarithm
    image **45**
negate
    filter **60**
    image **45**
new-pane 13
next-pane 114, **15**
normalize-cols **69**
normalize-rows **69**
normalize
    vector **64**
    viewable-matrix **59**
not. 52
notify-of-inferior-destruction **93**
notify-of-superior-destruction **94**
num-levels 33
obvius-init.lisp 107
obvius-window-init.lisp 107
obvius::*obvius-misc-functions* 79
obvius::run-heap-diagnostics 111
ogc 12, **20**, **89**, 91, 93, 94
one-d-graph 108
one-d-image-p **51**
one-d-image 23

or. 52
orthogonal-p **65**
outer-product
vector **64**
viewable-matrix **59**
overlay 110, **38**
pane-coord-to-viewable-coord 100
pane 11, **13**
paste-cols **70**
paste-rows **70**
pasteup-format 32
pasteup 25, **31**
paste 111, 113, 51
bit-image **52**
image **47**
pedestal 14, 29, 30, 31, 32
periodic-discrete-function 114, 24
periodic-point-operation
image **46**
pictures-of 10
picture 10, 11, **13**, 7
plot-symbol 35, 36
point-maximum
image **48**
point-minimum
image **48**
point-operation 114, 24
image **45**
polar-image 110, **25**
polar-plot **35**
polar-to-complex 110, 52, **54**
polar-to-rect 110
pop-picture **16**, **98**
position-message **100**
postscript-bltable 99
power-spectrum
image **50**
power
image **45**
present 112
picture **101**
viewable **94**
principal-components **66**
print-object **93**
print-top-level-values 96
print-values 23, **93**
image **41**, **53**
purge! 114, 12
qmf-pyramid-p **61**
qr-decomposition **67**
qrd 67
quadratic-decomposition **66**
quantize
image **46**
randomize **62**
range **55**
image **49**
rank **41**
real-part **54**
rect-to-polar 110
reduce. **57**
refresh **15**, **98**
reinitialize-instance 99
render **101**
repl 112, 95, **96**
resample 114
image **47**
reset-picture-defaults **99**
retain-bitmap 33, 35, 99
rgb-bits 31
right-image **53**
right-singular-matrix **66**
room **19**
rotate 114
image **47**
round. 114
image **46**
row-dim **68**
viewable-matrix **58**
row-null-space-qr **67**
row-null-space **67**
row-space-qr **67**
row-space **67**
rows **68**
row **68**
run-heap-diagnostics **89**
save-image 17
image-sequence **19**, **44**

image **18**, **44**
save-picture 44
scalar-multiple **62**
scale 14, 29, 30, 31, 32, 34
scatter-plot
image **36**
scrounge! 20, 89
second-image **53**, 54
separable-filter-p **60**
separable-filter **26**
seq-delay 37
sequence-length **56**
set-default 107
set-history 86, 94
set-not-current **100**, 11, 86, **93**, 96
set-result 112, 86, **92**, 93
setp 100, 14, **16**, 29, 34
settable-parameters **92**, **99**
shared-initialize 99
shift-by-pi **60**
shuffle-rows **69**
shuffle **62**
side-by-side **47**, 51
similar 51
bit-image **52**
image **41**
singular-p **65**
singular-value-decomposition **66**
singular-values **66**
size
viewable-matrix **58**
skew **49**
skip 33, 34
slice 23
solve-eigenvalue **67**
sort-rows **70**
sqr. 45
sqrt. 45
sqr 45
square-error **48**
square-magnitude **54**
square-p **65**
square-root
image **45**
square
image **45**
static-arrays-of **100**, 91, **93**
status-message 100
sub-cols
matrix vector **69**
vector matrix **69**
sub-display-type 37
sub-rows **69**
matrix vector **69**
sub-sequence **56**
subsample 51
image **47**
sub 52
filter **60**
image **44**
sum-cols **69**
sum-of 60
sum-rows **69**
superiors-of 10, 94
surface-plot 32
svd 66
swap-rows **70**
switch-components **54**
symbol-size 35, 36
symmetric-p **65**
system-dependent-frob 100, 98, 99
title-bar-string **100**
total-size 51
image **41**
viewable-matrix **59**
transpose **47**, 51
truncate. 114
unitary-p **65**
upsample 51
image **46**
variance **49**
vector-field 26, **34**, 99
vector-length
vector **64**
vectorize **68**
viewable-matrix **59**
viewable-list
viewable-sequence **56**

viewable-matrix-p **58**
viewable-matrix 28
viewable-sequence-p **56**
viewable-sequence 112, **25**
viewable **10**, 7
    auto display 11
    destructive modification 11
    slots of 10
volume **60**
warp 114
    image **47**
with-interrupts-deferred 91
with-local-images 109
with-local-viewables 109, **86**, 90
with-locked-pane 111, 83, 95
with-result-<vbl> 111
with-result 111, 85, 92, 93, 94, 96
with-static-arrays **88**
with-static-svd **66**
with-svd **66**
x-axis 35
x-component **53**, 94
x-dim 51
    bit-image **51**
    filter **59**
    image **41**
    sequence **56**
    viewable-matrix **58**
x-label 35
x-offset 30
x-range 35
y-axis 35
y-component **53**
y-dim 51
    bit-image **52**
    filter **60**
    image **41**
    sequence **56**
    viewable-matrix **58**
y-label 35
y-offset 30
y-range 35
y-tick-step 35
z-max 33
z-min 33
z-size 32
zero!
    image **44**
zero-crossings **48**, 51
zoom-picture **100**
zoom 30, 32, 33, 34, 35, 37
    slot 16