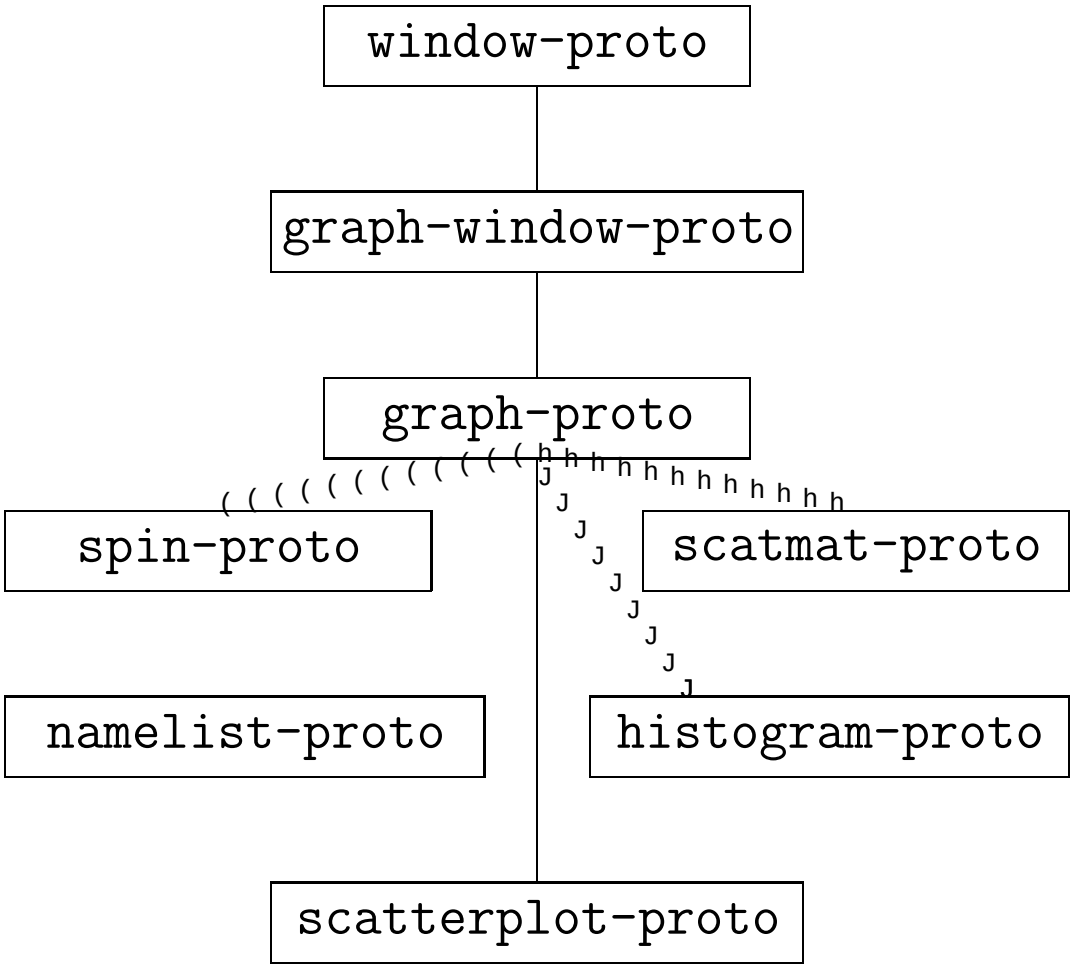
Outline of the Graphics System

Overview

The graphics window object tree:



All top-level windows share certain common features:

- a title
- a way to be moved
- a way to be resized

These common features are incorporated in the window prototype `window-proto`.

Both dialog and graphics windows inherit from the window prototype.

Graph Windows

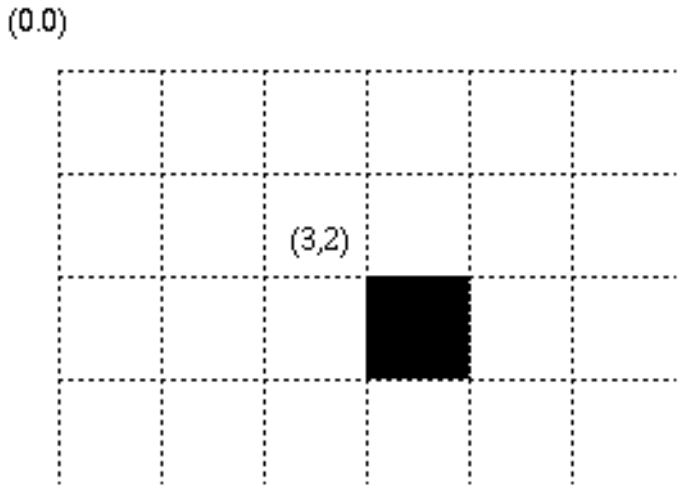
Outline of the Drawing System

A graphics window is a view onto a *drawing canvas*.

The dimensions of the canvas can be *fixed* or *elastic*.

- If a dimension is fixed, it has a scroll bar.
- If it is elastic, it fills the window.

The name list window uses one fixed dimension; all other standard plots use elastic dimensions by default.



The canvas has a coordinate system.

- coordinate units are pixels
- the origin is the top-left corner
- the x coordinate increases from left to right
- the y coordinate increases from top to bottom

A number of drawing operations are available.

Drawable objects include

- rectangles
- ovals
- arcs
- polygons

These can be

- framed
- painted
- erased

Other drawables are

- symbols
- strings
- bitmaps

The precise effect of drawing operations depends on the *state* of the drawing system.

Drawing system state components include

- colors – foreground and background
- drawing mode – normal or XOR
- line type – dashed or solid
- pen width – an integer
- use color – on or off
- buffering – on or off

Animation Techniques

There are two basic animation methods

- *XOR drawing* – drawing "inverts" the colors on the screen
- *double buffering* – a picture is built up in a background buffer and then copied to the screen

Some tradeoffs:

- XOR drawing is usually faster
- XOR drawing automatically preserves the background
- XOR drawing distorts background and object during drawing
- XOR drawing inherently involves a certain amount of flicker
- Moving several objects by XOR causes distortion – only one can move at a time
- *Inverting* is not well-defined on color displays

Lisp-Stat uses

- XOR drawing for moving the brush rectangle
- double buffering for rotation

As an example, let's move a highlighted symbol down the diagonal of a window using both methods.

A new graphics window is constructed by

```
(setf w (send graph-window-proto :new))
```

Using XOR drawing:

```
(let ((width (send w :canvas-width))
      (height (send w :canvas-height))
      (mode (send w :draw-mode)))
  (send w :draw-mode 'xor)
  (dotimes (i (min width height))
    (send w :draw-symbol 'disk t i i)
    (pause 2)
    (send w :draw-symbol 'disk t i i))
  (send w :draw-mode mode))
```

Using double Buffering:

```
(let ((width (send w :canvas-width))
      (height (send w :canvas-height)))
  (dotimes (i (min width height))
    (send w :start-buffering)
    (send w :erase-window)
    (send w :draw-symbol 'disk t i i)
    (send w :buffer-to-screen)))
```

Handling Events

User actions produce various *events* that the system handles by sending messages to the appropriate objects.

There are several types of events:

- resize events
- exposure or redraw events
- mouse events – motion and click
- key events
- idle "events"

Resize and redraw events cause `:resize` and `:redraw` messages to be sent to the window object.

Mouse and key events produce `:do-click`, `:do-motion`, and `:do-key` messages.

In idle periods, the `:do-idle` message is sent.

The `:while-button-down` message can be used to follow the mouse inside a click
(for dragging, etc.)

Graphics Window Menus

Every graphics window can have a menu.

The user interface guidelines of the window system determine how the menu is presented:

- On the Macintosh, the menu is installed in the menu bar when the window is the front window.
- In MS Windows, the menu is installed in the application's menu bar when the window is the front window.
- In *SunView*, the menu is popped up when the right mouse button is pressed in the window.
- Under *X11*, the menu is popped up when the mouse is clicked in a **Menu** button at the top of the window.

The `:menu` message retrieves a graph window's menu or installs a new menu.

An Example

As a simple example to illustrate the handling of events, let's construct a window that shows a single highlighted symbol at its center.

The window is constructed by

```
(setf w (send graph-window-proto :new))
```

We can add slots for holding the coordinates of our point:

```
(send w :add-slot 'x
      (/ (send w :canvas-width) 2))
(send w :add-slot 'y
      (/ (send w :canvas-height) 2))

(defmeth w :x (&optional (val nil set))
  (if set (setf (slot-value 'x) (round val)))
  (slot-value 'x))

(defmeth w :y (&optional (val nil set))
  (if set (setf (slot-value 'y) (round val)))
  (slot-value 'y))
```

New coordinate values are rounded since drawing operations require integer arguments.

The `:resize` method positions the point at the center of the canvas:

```
(defmeth w :resize ()
  (let ((width (send self :canvas-width))
        (height (send self :canvas-height)))
    (send self :x (/ width 2))
    (send self :y (/ height 2))))
```

The `:redraw` method erases the window and redraws the symbol at the location specified by the coordinate values:

```
(defmeth w :redraw ()
  (let ((x (send self :x))
        (y (send self :y)))
    (send self :erase-window)
    (send self :draw-symbol 'disk t x y)))
```

The `:do-click` message positions the symbol at the click and then allows it to be dragged:

```
(defmeth w :do-click (x y m1 m2)
  (flet ((set-sym (x y)
           (send self :x x)
           (send self :y y)
           (send self :redraw)))
    (set-symbol x y)
    (send self :while-button-down #'set-sym)))
```

The `:do-idle` method can be used to move the symbol in a random walk:

```
(defmeth w :do-idle ()
  (let ((x (send self :x))
        (y (send self :y)))
    (case (random 4)
      (0 (send self :x (- x 5)))
      (1 (send self :x (+ x 5)))
      (2 (send self :y (- y 5)))
      (3 (send self :y (+ y 5))))
    (send self :redraw)))
```

We can turn the random walk on by typing

```
(send w :idle-on t)
```

and we can turn it off with

```
(send w :idle-on nil)
```

A better solution is to use a menu item:

```
(setf run-item
      (send menu-item-proto :new "Run"
            :action
            #'(lambda ()
              (send w :idle-on
                    (not (send w :idle-on))))))
```

To put a check mark on this item when the walk is running, define an `:update` method:

```
(defmeth run-item :update ()
   (send self :mark (send w :idle-on)))
```

It would also be nice to have a menu item for restarting the walk:

```
(setf restart-item
      (send menu-item-proto :new "Restart"
            :action
            #'(lambda () (send w :restart))))
```

The `:restart` method is defined as

```
(defmeth w :restart ()
  (let ((width (send self :canvas-width))
        (height (send self :canvas-height)))
    (send self :x (/ width 2))
    (send self :y (/ height 2))
    (send self :redraw)))
```

and a menu with the two items is installed by

```
(setf menu
      (send menu-proto :new "Random Walk"))
(send menu :append-items restart-item run-item)
(send w :menu menu)
```

There may be some flickering when the random walk is running.

This flickering can be eliminated by modifying the `:redraw` method to use double buffering:

```
(defmeth w :redraw ()
  (let ((x (round (send self :x)))
        (y (round (send self :y))))
    (send self :start-buffering)
    (send self :erase-window)
    (send self :draw-symbol 'disk t x y)
    (send self :buffer-to-screen)))
```

Statistical Graphics Windows

`graph-proto` is the statistical graphics prototype.

It inherits from `graph-window-proto`.

The graph prototype is responsible for managing the data used by all statistical graphs.

Variations in how these data are displayed are implemented in separate prototypes for the standard graphs.

The graph prototype is a view into m dimensional space.

It allows the display of both points and connected line segments

The default methods in this prototype implement a simple scatterplot of two of the m dimensions.

Many features of graphics windows are enhanced to simplify adding new features to graphs.

The graph prototype adds the following features:

- m-dimensional point and *line start* data
- affine transformations consisting of
  - centering and scaling
  - a linear transformation
- ranges for raw, scaled and canvas coordinates
- mouse modes for controlling interaction
- linking strategy
- window layout management
  - margin, content and aspect
  - background (axes)
  - overlays
  - content
- standard menus and menu items

Data and Axes

The `:isnew` method for the graph prototype requires one argument, the number of variables:

```
> (setf w (send graph-proto :new 4))
#<Object: 302823396, prototype = GRAPH-PROTO>
```

Using the stack loss data as an illustration, we can add data

```
> (send w :add-points
        (list air temp conc loss))
NIL
```

and adjust scaling to make the data visible:

```
> (send w :adjust-to-data)
NIL
```

We can also add line segments:

```
> (send w :add-lines (list air temp conc loss))
NIL
```

You can control whether axes are drawn with the `:x-axis` and `:y-axis` messages:

```
> (send w :x-axis t)
(T NIL 4)
```

The range shown can be accessed and changed:

```
> (send w :range 0)
(50 80)
> (send w :range 1)
(17 27)
> (send w :range 1 15 30)
(15 30)
```

The function `get-nice-range` helps choosing a range and the number of ticks:

```
> (get-nice-range 17 27 4)
(16 28 7)
```

To remove the axis:

```
> (send w :x-axis nil)
(NIL NIL 4)
```

Initially, the plot shows the first two variables:

```
> (send w :current-variables)
(0 1)
```

This can be changed:

```
> (send w :current-variables 2 3)
(2 3)
> (send w :current-variables 0 1)
(0 1)
```

Plot data can be cleared by several messages:

```
(send w :clear-points)
(send w :clear-lines)
(send w :clear)
```

If the `:draw` keyword argument is `nil` the plot is not redrawn.

The default value is `t`.

Scaling and Transformations

The scale type controls the action of the default `:adjust-to-data` method.

The initial scale type is `nil`:

```
> (send w :scale-type)
NIL
> (send w :range 0)
(50 80)
> (send w :scaled-range 0)
(50 80)
```

Two other scale types are `variable` and `fixed`.

For variable scaling:

```
> (send w :scale-type 'variable)
VARIABLE
> (send w :range 0)
(35 95)
> (send w :scaled-range 0)
(-2 2)
```

The `:scale`, `:center`, and `:adjust-to-data` messages let you build your own scale types.

Initially there is no transformation:

```
> (send w :transformation)
NIL
```

If the current variables are 0 and 1, a rotation can be applied to replace `air` by `conc` and `temp` by `loss`:

```
(send w :transformation '#2A((0  0 -1  0)
                             (0  0  0 -1)
                             (1  0  0  0)
                             (0  1  0  0)))
```

The transformation can be removed by

```
(send w :transformation nil)
```

A transformation can also be applied incrementally:

```
(let* ((c (cos (/ pi 20)))
       (s (sin (/ pi 20)))
       (m (+ (* c (identity-matrix 4))
             (* s '#2A((0  0 -1  0)
                      (0  0  0 -1)
                      (1  0  0  0)
                      (0  1  0  0))))))
  (dotimes (i 10)
    (send w :apply-transformation m)))
```

A simpler message allows rotation within coordinate planes:

```
(dotimes (i 10)
  (send w :rotate-2 0 2 (/ pi 20) :draw nil)
  (send w :rotate-2 1 3 (/ pi 20)))
```

Several messages are available for accessing data values in raw, scaled, and screen coordinates.

Other messages are available for converting among coordinate systems.

Mouse Events and Mouse Modes

The graph prototype organizes mouse interactions into *mouse modes*.

Each mouse mode includes

- a symbol for choosing the mode from a program
- a title string used in the mode selection dialog
- a cursor to visually identify the mode
- mode-specific click and motion messages

It should not be necessary to override a graph's `:do-click` or `:do-motion` methods.

Initially there are two mouse modes, `selecting` and `brushing`.

We can add a new mouse mode by

```
(send w :add-mouse-mode 'identify
      :title "Identify"
      :click :do-identify
      :cursor 'finger)
```

The `:do-identify` method can be defined as

```
(defmeth w :do-identify (x y m1 m2)
  (let* ((cr (send self :click-range))
         (p (first
             (send self :points-in-rect
                   (- x 2) (- y 2) 4 4))))
    (if p
        (let ((mode (send self :draw-mode))
              (lbl (send self :point-label p)))
          (send self :draw-mode 'xor)
          (send self :draw-string lbl x y)
          (send self :while-button-down
                #'(lambda (x y) nil))
          (send self :draw-string lbl x y)
          (send self :draw-mode mode)))))
```

The button down action does nothing; it just waits.

An alternative is to allow the label to be dragged, perhaps to make it easier to read:

```
(defmeth w :do-identify (x y m1 m2)
  (let* ((cr (send self :click-range))
         (p (first
             (send self :points-in-rect
                   (- x 2) (- y 2) 4 4))))
    (if p
        (let ((mode (send self :draw-mode))
              (lbl (send self :point-label p)))
          (send self :draw-mode 'xor)
          (send self :draw-string lbl x y)
          (send self :while-button-down
                #'(lambda (new-x new-y)
                    (send self :draw-string lbl x y)
                    (setf x new-x)
                    (setf y new-y)
                    (send self :draw-string lbl x y)))
          (send self :draw-string lbl x y)
          (send self :draw-mode mode)))))
```

Standard Mouse Modes and Linking

The click and motion methods of the two standard modes use a number of messages.

In `selecting` mode, a click

- Sends `:unselect-all-points`, unless the extend modifier is used.
- Sends `:adjust-points-in-rect` with click x and y coordinates, width and height returned by `:click-range`, and the symbol `selected` as arguments.
- While the button is down, a dashed rectangle is stretched from the click to the mouse.

  When the button is released, `:adjust-points-in-rect` is sent with the rectangle coordinates and `selected` as arguments.

In `brushing` mode, a click

- Sends `:unselect-all-points` unless the extend modifier is used.
- While the mouse is dragged, sends `:adjust-points-in-rect` with the brush rectangle and `selected` as arguments.

In `brushing` mode, moving the mouse

- Sends `:adjust-points-in-rect` with the brush rectangle and `hilited` as arguments.

Points can be in four states:

`invisible`

`normal`

`hilited`

`selected`.

Linking is based on a *loose linking* model:

- points are related by index number
- only point states are adjusted

The system uses two messages to determine which plots are linked:

- `:links` returns a list of plots linked to the plot (possibly including the plot itself)
- `:linked` determines if the plot is linked, and turns linking on and off.

When a point's state is changed in a plot,

- Each linked plot (and the plot itself) is sent the `:adjust-screen-point` message with the index as argument.
- The action taken by the method for `:adjust-screen-point` may depend on both current and previous states.

Since it is not always feasible to redraw single points,

- the `:needs-adjusting` method can be used to check or set a flag
- the `:adjusting-screen` method can redraw the entire plot if the flag is set

The easiest, though not necessarily the most efficient, way to augment standard mouse modes is to define a new `:adjust-screen` method

Some useful messages:

For points specified by index:

`:point-showing`

`:point-hilited`

`:point-selected`

For sets of indices:

`:selection` or `:points-selected`

`:points-hilited`

`:points-showing`

Other operations:

`:erase-selection`

`:show-all-points`

`:focus-on-selection`

`:adjust-screen`

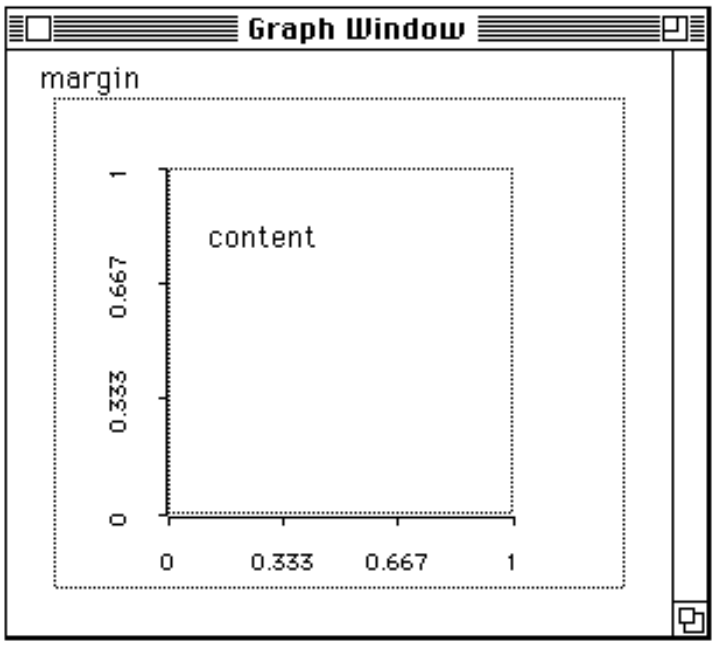
Some useful predicates:

`:any-points-selected-p`

`:all-points-showing-p`

Window Layout and Redrawing

The `:resize` method maintains a margin and a content rectangle



- The plot is surrounded by a margin, used by plot controls.
- The content rectangle can use a fixed or a variable aspect ratio.
- The size of the content depends on the aspect ratio and the axes.
- The plot can be covered by overlays, resized with `:resize-overlays`.

The aspect type used can be changed by

```
(send w :fixed-aspect t)
```

The `:redraw` method sends three messages:

- `:redraw-background` – erases the canvas and draws the axes
- `:redraw-overlays` – sends each overlay the `:redraw` message
- `:redraw-content` – redraws points, lines, etc.

Many methods, like `:rotate-2`, also send `:redraw-content`.

Plot overlays are useful for holding controls.

- Overlays inherit from `graph-overlay-proto`.
- Overlays are like transparent sheets of plastic.
- Overlays are drawn from the bottom up.
- Overlays can intercept mouse clicks.
- Clicks are processed from the top down:
  - Each overlay is sent the `:do-click` message until one returns a non-`nil` result.
  - Only if no overlay accepts a click is the click passed to the current mouse mode.

The controls of a rotating plot are implemented as an overlay.

Other examples of overlays are in `plotcontrols.lsp` in the **Examples** folder.

Menus and Menu Items

To help construct standard menus

- `:menu-title` returns the title to use
- `:menu-template` returns a list of items or symbols
- `:new-menu` constructs and installs the new menu

The `:isnew` method sends the plot the `:new-menu` message when it is created.

Standard items can be specified as symbols in the template:

- `color`
- `dash`
- `focus-on-selection`
- `link`
- `mouse`
- `options`
- `redraw`
- `erase-selection`
- `rescale`
- `save-image`
- `selection`
- `show-all`
- `showing-labels`
- `symbol`

Standard Statistical Graphs

Each of the standard plot prototypes needs only a few new methods.

The main additional or changes methods are:

`scatterplot-proto`:

Overrides: `:add-points`, `:add-lines`, `:adjust-to-data`.

New messages: `:add-boxplot`, `:add-function-contours`, `:add-surface-contour`, `:add-surface-contours`.

`scatmat-proto`:

Overrides: `:add-lines`, `:add-points`, `:adjust-points-in-rect`, `:adjust-screen-point`, `:do-click`, `:do-motion`, `:redraw-background`, `:redraw-content`, `:resize`.

`spin-proto`:

Overrides: `:adjust-to-data`, `:current-variables`, `:do-idle`, `:isnew`, `:resize`, `:redraw-content`.

New methods: `:abcplane`, `:add-function`, `:add-surface`, `:angle`, `:content-variables`, `:depth-cuing`, `:draw-axes`, `:rotate`, `:rotation-type`, `:showing-axes`

`histogram-proto`:

Overrides: `:add-points`, `:adjust-points-in-rect`, `:adjust-screen`, `:adjust-screen-point`, `:adjust-to-data`, `:clear-points`, `:drag-point`, `:isnew`, `:redraw-content`, `:resize`.

New methods: `:num-bins`, `:bin-counts`.

`name-list-proto`:

Overrides: `:add-points`, `:adjust-points-in-rect`, `:adjust-screen-point`, `:redraw-background`, `:redraw-content`.