Graphics: Legacy Library

Version 5.3.2

January 31, 2013

```
(require graphics/graphics)
```

The viewport graphics library is a relatively simple toolbox of graphics commands. The library is not very powerful; it is intended as a simplified alternative to `racket/gui`'s full graphical toolbox.

The graphics library originated as SIXlib, a library of X Windows commands available within Chez Scheme at Rice University. The functionality of that library has been reproduced (with backward compatibility) in this version.

Contents

1 Basic Commands

```
(open-graphics) → void?
```

Initializes the library's graphics routines. It must be called before any other graphics operations.

```
(close-graphics) → void?
```

Closes all of the windows. Until `open-graphics` is called again, no graphics routines will work.

```
(graphics-open?) → boolean?
```

Determines if the graphics have been opened (or not).

```
(open-viewport name horiz vert) → viewport?
  name : string?
  horiz : exact-nonnegative-integer?
  vert : exact-nonnegative-integer?
(open-viewport name dimensions) → viewport?
  name : string?
  dimensions : posn?
```

Creates a new window called *name*. The window is *horiz* pixels wide and *vert* pixels high. For backward compatibility, a single `posn` value can be submitted in the place of *horiz* and *vert*. The result is a viewport descriptor.

```
(open-pixmap name horiz vert) → viewport?
  name : string?
  horiz : exact-nonnegative-integer?
  vert : exact-nonnegative-integer?
(open-pixmap name dimensions) → viewport?
  name : string?
  dimensions : posn?
```

Like `open-viewport`, but the resulting viewport is not displayed on the screen. Offscreen pixmaps are useful for executing a sequence of drawing commands and displaying them all at once with `copy-viewport`.

Offscreen pixmaps are also useful in conjunction with viewport->snip (see below). This allows functions to compute with graphical objects and view the graphics when results are returned to the interactions window.

```
(close-viewport viewport) → void?
  viewport : viewport?
```

Removes the viewport from the screen and makes subsequent operations dealing with the viewport illegal.

```
(viewport? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a viewport (i.e., a destination for drawing), `#f` otherwise.

2 Position Operations

A position is a pixel location within a viewport. The upper-left corner is pixel (0, 0), and positions increase to the right and down.

```
(struct posn (x y)
        #:extra-constructor-name make-posn)
  x : real?
  y : real?
```

Represents a positions.

```
((get-pixel viewport) p) → (one-of/c 0 1)
  viewport : viewport?
  p : posn?
```

Returns the color of the pixel at position *p* in *viewport*; 0 denotes white and 1 denotes not white.

```
((get-color-pixel viewport) p) → rgb?
  viewport : viewport?
  p : posn?
```

Returns an rgb value for color of the pixel at position *p* in *viewport*.

```
((test-pixel viewport) color) → rgb?
  viewport : viewport?
          (or/c (integer-in 0 299)
  color :       string?
                rgb?)
```

Returns the color that will actually be used if *color* is used to draw.

3 Color Operations

A color can be represented in three ways: as a color index (an integer in 0 to 299, inclusive), as a color name string, or as a `rgb` value. All drawing functions which take a color argument accept colors in any form. An `rgb` value is assigned to an index with `change-color`.

```
(struct rgb (red green blue)
        #:extra-constructor-name make-rgb)
  red : (real-in 0 1)
  green : (real-in 0 1)
  blue : (real-in 0 1)
```

Takes three values in the range 0 (dark) to 1 (bright) and returns an `rgb` (a color).

```
(change-color index rgb) → void?
  index : (integer-in 0 299)
  rgb : rgb?
```

Changes the color at *index* in the color table to the color specified in *rgb*. Only the first twenty-one indices are initialized; a color index should not be used until it has been initialized.

```
(default-display-is-color?) → boolean?
```

Returns `#t` if the default display screen for viewports is in color or `#f` otherwise.

4 Draw, Clear, and Flip Operations

The following are the basic graphics operations for drawing to a viewport. Each function takes a viewport as its argument and returns a function operating within that viewport. Further arguments, if any, are curried. For example, `(draw-line viewport)` returns a function, that can then be applied to the proper arguments to draw a line in the viewport corresponding to viewport descriptor *viewport*.

In general, `draw-` functions make pixels black or colored, `clear-` functions make them white, and `flip-` commands *invert* pixels (which makes black white, white black, and is otherwise ill-defined).

4.1 Viewports

```
((draw-viewport viewport) [color]) → void?
  viewport : viewport?
  color :       (or/c (integer-in 0 299)
                      string?            = "black"
                      rgb?)
```

Colors the entire contents of *viewport* with *color*.

```
((clear-viewport viewport)) → void?
  viewport : viewport?
```

Whitens the entire contents of *viewport*.

```
((flip-viewport viewport)) → void?
  viewport : viewport?
```

Inverts the entire contents of *viewport*.

```
(copy-viewport source dest) → void?
  source : viewport?
  dest : viewport?
```

Copies the content of *source* into *dest*.

4.2 Pixels

```
((draw-pixel viewport) p [color]) → void?
```

```
  viewport : viewport?
  p : posn?
          (or/c (integer-in 0 299)
  color :       string?            = "black"
                rgb?)
```

Colors the pixel in *viewport* at *p*.

```
((clear-pixel viewport) p) → void?
  viewport : viewport?
  p : posn?
```

Whitens the pixel in *viewport* at *p*.

```
((flip-pixel viewport) p) → void?
  viewport : viewport?
  p : posn?
```

Inverts the pixel in *viewport* at *p*.

4.3 Lines

```
((draw-line viewport) p1 p2 [color]) → void?
  viewport : viewport?
  p1 : posn?
  p2 : posn?
          (or/c (integer-in 0 299)
  color :       string?            = "black"
                rgb?)
```

Draws a line in *viewport* connecting positions *p1* and *p2*.

```
((clear-line viewport) p1 p2) → void?
  viewport : viewport?
  p1 : posn?
  p2 : posn?
```

Whitens a line in *viewport* connecting positions *p1* and *p2*.

```
((flip-line viewport) p1 p2) → void?
  viewport : viewport?
  p1 : posn?
  p2 : posn?
```

Inverts a line in *viewport* connecting positions *p1* and *p2*.

4.4 Rectangles

```
((draw-rectangle  viewport)
                  p
                  width
                  height
                 [color])    → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
         (or/c (integer-in 0 299)
  color :       string?            = "black"
                rgb?)
```

Draws a rectangle border in the *viewport* with the top-left of the rectangle at the position *p* and with sides *width* across and *height* tall.

```
((clear-rectangle viewport) p width height) → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Whitens a rectangle border in the *viewport*, analogous to draw-rectangle.

```
((flip-rectangle viewport) p width height) → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Inverts a rectangle border in the *viewport*, analogous to draw-rectangle.

```
((draw-solid-rectangle  viewport)
                        p
                        width
                        height
                       [color])    → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
         (or/c (integer-in 0 299)
  color :       string?            = "black"
                rgb?)
```

Draws a solid rectangle in the *viewport* with the top-left of the rectangle at the position *p* and with sides *width* across and *height* tall.

```
((clear-solid-rectangle viewport)
                        p
                        width
                        height)   → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Whitens a rectangle border in the *viewport*, analogous to `draw-solid-rectangle`.

```
((flip-solid-rectangle viewport)
                       p
                       width
                       height)   → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Inverts a rectangle border in the *viewport*, analogous to `draw-solid-rectangle`.

4.5 Ellipses

```
((draw-ellipse viewport) p width height [color]) → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
         (or/c (integer-in 0 299)
  color :       string?           = "black"
               rgb?)
```

Draws a ellipse border in the *viewport*. The ellipse is inscribed with a rectangle whose top-left is at position *p* and with sides *width* across and *height* tall.

```
((clear-ellipse viewport) p width height) → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Whitens a ellipse border in the *viewport*, analogous to draw-ellipse.

```
((flip-ellipse viewport) p width height) → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Inverts a ellipse border in the *viewport*, analogous to draw-ellipse.

```
((draw-solid-ellipse  viewport)
                      p
                      width
                      height
                     [color])    → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
         (or/c (integer-in 0 299)
  color :       string?            = "black"
               rgb?)
```

Draws a solid ellipse in the *viewport*. The ellipse is inscribed with a rectangle whose top-left is at position *p* and with sides *width* across and *height* tall.

```
((clear-solid-ellipse viewport)
                      p
                      width
                      height)    → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Whitens a ellipse border in the *viewport*, analogous to draw-solid-ellipse.

```
((flip-solid-ellipse viewport)
                     p
                     width
                     height)    → void?
  viewport : viewport?
  p : posn?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Inverts a ellipse border in the *viewport*, analogous to draw-solid-ellipse.

4.6 Polygons

```
((draw-polygon viewport) points offset [color]) → void?
  viewport : viewport?
  points : (listof posn?)
  offset : posn?
         (or/c (integer-in 0 299)
  color :       string?           = "black"
               rgb?)
```

Draws a polygon border in *viewport* using *points* for the polygon vertices and *offset* as an offset added to all points.

```
((clear-polygon viewport) points offset) → void?
  viewport : viewport?
  points : (listof posn?)
  offset : posn?
```

Whitens a polygon border in *viewport*, analogous to draw-polygon.

```
((flip-polygon viewport) points offset) → void?
  viewport : viewport?
  points : (listof posn?)
  offset : posn?
```

Inverts a polygon border in *viewport*, analogous to draw-polygon.

```
((draw-solid-polygon  viewport)
                      points
                      offset
                     [color])    → void?
  viewport : viewport?
  points : (listof posn?)
  offset : posn?
         (or/c (integer-in 0 299)
  color :       string?           = "black"
               rgb?)
```

Draws a solid polygon in *viewport* using *points* for the polygon vertices and *offset* as an offset added to all points.

```
((clear-solid-polygon viewport)
                      points
                      offset)    → void?
  viewport : viewport?
  points : (listof posn?)
  offset : posn?
```

Whitens a polygon border in *viewport*, analogous to draw-solid-polygon.

```
((flip-solid-polygon viewport)
                     points
                     offset)   → void?
  viewport : viewport?
  points : (listof posn?)
  offset : posn?
```

Inverts a polygon border in *viewport*, analogous to draw-solid-polygon.

4.7 Strings

```
((draw-string viewport) p str [color]) → void?
  viewport : viewport?
  p : posn?
  str : string?
        (or/c (integer-in 0 299)
  color :       string?           = "black"
                rgb?)
```

Draws a string at a specified location in the *viewport*. The lower left of the string begins at *p*.

```
((clear-string viewport) p str) → void?
  viewport : viewport?
  p : posn?
  str : string?
```

Whitens a string at a specified location in the *viewport*. The lower left of the string begins at *p*.

```
((flip-string viewport) p str) → void?
  viewport : viewport?
  p : posn?
  str : string?
```

Inverts a string at a specified location in the *viewport*. The lower left of the string begins at *p*.

4.8 Pixmaps

```
(((draw-pixmap-posn  file
                     [type]))
                      viewport)
                      p
                     [color])    → void?
  file : path-string?
         (one-of/c 'unknown 'unknown/mask
  type :           'gif 'gif/mask 'jpeg 'png 'png/mask
                   'xbm 'xpm 'bmp)
       = 'unknown/mask
  viewport : viewport?
  p : posn?
          (or/c (integer-in 0 299)
  color :       string?            = "black"
                rgb?)
```

Draws a pixmap into *viewport* with its upper left corner at position *p*. If *type* is 'unknown or 'unknown/mask, then the content of the file is examined to determine the type. The 'gif/mask, 'png/mask, and 'unknown/mask types draw the bitmap with a transparent background if filename refers to a GIF/PNG file with a transparent background.

The argument *color* is only used when the loaded pixmap is monochrome. In that case, the color is used instead of black in the drawn image.

```
((draw-pixmap viewport) file p [color]) → void?
  viewport : viewport?
  file : path-string?
  p : posn?
          (or/c (integer-in 0 299)
  color :       string?            = "black"
                rgb?)
```

Equivalent to (((draw-pixmap-posn *file*) *viewport*) *p* *color*).

```
((save-pixmap viewport) file [type]) → void?
  viewport : viewport?
  file : path-string?
  type : (one-of/c 'gif 'jpeg 'png 'xbm 'xpm 'bmp) = 'xpm
```

Saves the current content of *viewport* to *file*. The *type* argument determines the kind of file that is written.

5 World Operations

Every canvas comes with an associated world. A client program can set the world, start the world's clock, stop the world's clock, and deal with tick events (the clock ticks) and keyboard inputs (keyevents).

```
((init-world viewport) v) → void?
  viewport : viewport?
  v : any/c
```

Sets the initial value of *viewport*'s world to *v*.

```
((set-on-tick-event viewport)
                    secs
                    update-callback) → void?
  viewport : viewport?
  secs : real?
  update-callback : (any/c . -> . any/c)
```

For *viewport*, sets *update-callback* to be invoked to transform the world value every *secs* seconds. Only one callback is installed at a time.

```
((stop-tick viewport)) → void?
  viewport : viewport?
```

Stops updating *viewport*'s world via a callback installed with set-on-tick-event.

```
((set-on-key-event viewport) key-callback) → void?
  viewport : viewport?
  key-callback : (any/c any/c . -> . any/c)
```

Sets *key-callback* as the function to call whenever a key event is received for *viewport*. The *key-callback* is given a key event and the current world, and it produces an updated world.

6 Miscellaneous Operations

```
((get-string-size viewport) str) → (list/c real? real?)
  viewport : viewport?
  str : string?
```

Returns the size of *str* as drawn into *viewport* as a list of two numbers: width and height.

```
(viewport->snip viewport) → (is-a?/c snip%)
  viewport : viewport?
```

Returns an object that can be inserted into an editor buffer to display the current image in the viewport. (Subsequent drawing to the viewport does not affect the snip's image.)

When snips are the results of computations in the interactions window, DrRacket shows the snip in the interactions window.}

```
(viewport-dc viewport) → (is-a?/c dc<%>)
  viewport : viewport?
```

Returns an object for direct drawing into *viewport*'s on-screen representation (if any). Mirror all such drawing to the result of `(viewport-buffer-dc viewport)`, too.

```
(viewport-buffer-dc viewport) → (is-a?/c dc<%>)
  viewport : viewport?
```

Returns an object for direct drawing into *viewport*'s off-screen representation. Mirror all such drawing to the result of `(viewport-dc viewport)`, too.

7 An Example

```
(open-graphics)
; nothing appears to happen, but the library is initialized...

(define w (open-viewport "practice" 300 300))
; viewport window appears

((draw-line w) (make-posn 30 30) (make-posn 100 100))
; line appears

(close-viewport w)
; viewport disappears

(close-graphics)
; again, nothing appears to happen, but
; unclosed viewports (if any) would disappear
```

8 A More Complicated Example

The use of multiple viewports, viewport descriptors, drawing operations for multiple viewports is as easy as the use of a single viewport:

```
(open-graphics)
(let* (; w1 and w2 are viewports for different windows
       [w1  (open-viewport "viewport 1" 300 300)]
       [w2  (open-viewport "viewport 2" 200 500)]
       ; d1 and d2 draw lines in different viewports
       [d1  (draw-line w1)]
       [d2  (draw-line w2)])
  ; draws a line in viewport labeled ‘‘viewport 1’’
  (d1 (make-posn 100 5) (make-posn 5 100))
  ; draws a line in viewport labeled ‘‘viewport 2’’
  (d2 (make-posn 100 100) (make-posn 101 400)))

; we no longer have access to viewports 1 and 2,
; since their descriptors did not escape the let
(close-graphics)
; removes the viewports
```

9 Protecting Graphics Operations

To guarantee the proper closing of viewports in cases of errors, especially when a program manages several viewports simultaneously, a programmer should use `dynamic-wind:`

```
(let ([w (open-viewport "hello" 100 100)])
  (dynamic-wind
    ; what we want to happen first: nothing
    void
    ; the main program (errors constrained to this piece)
    (lambda () (draw-pixel 13))  ; an error
    ; what we would like to happen, whether the main program
    ; finishes normally or not
    (lambda () (close-viewport w))))
```

10 Mouse Operations

The graphics library contains functions that determine where the mouse is, if there are any clicks, etc. The functions `get-mouse-click` and `ready-mouse-click` first return a "mouse-click descriptor," and then other functions take the descriptor and return the mouse's position, which button was pushed, etc. Mouse clicks are buffered and returned in the same order in which they occurred. Thus, the descriptors returned by `get-mouse-click` and `ready-mouse-click` may be from clicks that occurred long before these functions were called.

```
(get-mouse-click viewport) → mouse-click?
  viewport : viewport?
```

Returns the next mouse click in *viewport*, waiting for a click if necessary.

```
(ready-mouse-click viewport) → (or/c mouse-click? false/c)
  viewport : viewport?
```

Returns either a mouse click descriptor or `#f` if none is available. Unlike `get-mouse-click`, `ready-mouse-click` always returns immediately.

```
(ready-mouse-release viewport) → (or/c mouse-click? false/c)
  viewport : viewport?
```

Returns either a click descriptor from a mouse-release (button-up) event or `#f` if none is available.

```
(query-mouse-posn viewport) → (or/c posn? false/c)
  viewport : viewport?
```

Returns either the position of the mouse cursor within *viewport* or else `#f` if the cursor is currently outside *viewport*.

```
(mouse-click-posn mouse-click) → posn?
  mouse-click : mouse-click?
```

Returns the position of the pixel within a viewport where a given mouse click occurred.

```
(left-mouse-click? mouse-click) → boolean?
  mouse-click : mouse-click?
```

Returns `#t` if the mouse click occurred with the left mouse button, `#f` otherwise.

```
(middle-mouse-click? mouse-click) → boolean?
  mouse-click : mouse-click?
```

Returns `#t` if the mouse click occurred with the middle mouse button, `#f` otherwise.

```
(right-mouse-click? mouse-click) → boolean?
  mouse-click : mouse-click?
```

Returns `#t` if the mouse click occurred with the right mouse button, `#f` otherwise.

11 Keyboard Operations

The graphics library contains functions that report key presses from the keyboard. The functions `get-key-press` and `ready-key-press` return a "key-press descriptor," and then `key-value` takes the descriptor and returns a character or symbol (usually a character) representing the key that was pressed. Key presses are buffered and returned in the same order in which they occurred. Thus, the descriptors returned by `get-key-press` and `ready-key-press` may be from presses that occurred long before these functions were called.

```
(get-key-press viewport) → key-press?
  viewport : viewport?
```

Returns the next key press in the *viewport*, waiting for a key press if necessary.

```
(ready-key-press viewport) → key-press?
  viewport : viewport?
```

Returns the next key press in the *viewport* or returns `#f` if none is available. Unlike `get-key-press`, `ready-key-press` always returns immediately.

```
(key-value key-press) → (or/c character? symbol?)
  key-press : key-press?
```

Returns a character or special symbol for the key that was pressed. For example, the Enter key generates `#\return`, and the up-arrow key generates `'up`. For a complete list of possible return values, see `get-key-code`.

12 Flushing

```
(viewport-flush-input viewport) → void?
  viewport : viewport?
```

Empties all mouse and keyboard events in the input buffer of *viewport*.

13 Graphics Library as a Unit

13.1 Signatures

```
(require graphics/graphics-sig)
```

graphics^ : signature

Includes all of the bindings defined earlier in this chapter, except the posn bindings of §2 “Position Operations”.

graphics:posn^ : signature

Includes the posn bindings of §2 “Position Operations”.

13.2 Unit with posn

```
(require graphics/graphics-unit)
```

graphics@ : unit?

Imports mred^ and exports both graphics^ and graphics:posn^.

13.3 Unit without posn

```
(require graphics/graphics-posn-less-unit)
```

graphics-posn-less@ : unit?

Imports mred^ and graphics:posn^ and exports graphics^.