The Racket Drawing Toolkit

Version 6.2.1

Matthew Flatt,
Robert Bruce Findler,
and John Clements

August 10, 2015

```
(require racket/draw)
```
package: `draw-lib`

The `racket/draw` library provides all of the class, interface, and procedure bindings defined in this manual.

Contents

Index **125**

1 Overview

The racket/draw library provides a drawing API that is based on the PostScript drawing model. It supports line drawing, shape filling, bitmap copying, alpha blending, and affine transformations (i.e., scale, rotation, and translation).

See §13 “Classes and Objects” for an introduction to classes and interfaces in Racket.

Drawing with racket/draw requires a *drawing context* (*DC*), which is an instance of the dc<%> interface. For example, the post-script-dc% class implements a dc<%> for drawing to a PostScript file, while bitmap-dc% draws to a bitmap. When using the racket/gui library for GUIs, the get-dc method of a canvas returns a dc<%> instance for drawing into the canvas window.

See §1.2 “Drawing in Canvases” for an introduction to drawing in a GUI window.

1.1 Lines and Simple Shapes

To draw into a bitmap, first create the bitmap with make-bitmap, and then create a bitmap-dc% that draws into the new bitmap:

```
(define target (make-bitmap 30 30)) ; A 30x30 bitmap
(define dc (new bitmap-dc% [bitmap target]))
```

Then, use methods like draw-line on the DC to draw into the bitmap. For example, the sequence

```
(send dc draw-rectangle
      0 10   ; Top-left at (0, 10), 10 pixels down from top-left
      30 10) ; 30 pixels wide and 10 pixels high
(send dc draw-line
      0 0    ; Start at (0, 0), the top-left corner
      30 30) ; and draw to (30, 30), the bottom-right corner
(send dc draw-line
      0 30   ; Start at (0, 30), the bottom-left corner
      30 0)  ; and draw to (30, 0), the top-right corner
```

draws an “X” on top of a smaller rectangle into the bitmap target. If you save the bitmap to a file with (send target save-file "box.png" 'png), the "box.png" contains the image

in PNG format.

A line-drawing drawing operation like `draw-line` uses the DC's current *pen* to draw the line. A pen has a color, line width, and style, where pen styles include `'solid`, `'long-dash`, and `'transparent`. Enclosed-shape operations like `draw-rectangle` use both the current pen and the DC's current *brush*. A brush has a color and style, where brush styles include `'solid`, `'cross-hatch`, and `'transparent`.

For example, set the brush and pen before the drawing operations to draw a thick, red "X" on a green rectangle with a thin, blue border:

```
(send dc set-brush "green" 'solid)
(send dc set-pen "blue" 1 'solid)
(send dc draw-rectangle 0 10 30 10)
(send dc set-pen "red" 3 'solid)
(send dc draw-line 0 0 30 30)
(send dc draw-line 0 30 30 0)
```

In DrRacket, instead of saving `target` to a file viewing the image from the file, you can use `(require racket/gui)` and `(make-object image-snip% target)` to view the bitmap in the DrRacket interactions window.

To draw a filled shape without an outline, set the pen to `'transparent` mode (with any color and line width). For example,

```
(send dc set-pen "white" 1 'transparent)
(send dc set-brush "black" 'solid)
(send dc draw-ellipse 5 5 20 20)
```

By default, a `bitmap-dc%` draws solid pixels without smoothing the boundaries of shapes. To enable smoothing, set the smoothing mode to either `'smoothed` or `'aligned`:

```
(send dc set-smoothing 'aligned)
(send dc set-brush "black" 'solid)
(send dc draw-ellipse 4 4 22 22) ; a little bigger
```

The difference between `'aligned` mode and `'smoothed` mode is related to the relatively coarse granularity of pixels in a bitmap. Conceptually, drawing coordinates correspond to the lines between pixels, and the pen is centered on the line. In `'smoothed` mode, drawing on a line causes the pen to draw at half strength on either side of the line, which produces the following result for a 1-pixel black pen:

but 'aligned mode shifts drawing coordinates to make the pen fall on whole pixels, so a 1-pixel black pen draws a single line of pixels:

1.2 Pen, Brush, and Color Objects

The set-pen and set-brush methods of a DC accept pen% and brush% objects, which group together pen and brush settings.

```
(require racket/math)

(define no-pen (new pen% [style 'transparent]))
(define no-brush (new brush% [style 'transparent]))
(define blue-brush (new brush% [color "blue"]))
(define yellow-brush (new brush% [color "yellow"]))
(define red-pen (new pen% [color "red"] [width 2]))

(define (draw-face dc)
  (send dc set-smoothing 'aligned)

  (send dc set-pen no-pen)
  (send dc set-brush blue-brush)
  (send dc draw-ellipse 25 25 100 100)

  (send dc set-brush yellow-brush)
  (send dc draw-rectangle 50 50 10 10)
  (send dc draw-rectangle 90 50 10 10)

  (send dc set-brush no-brush)
  (send dc set-pen red-pen)
  (send dc draw-arc 37 37 75 75 (* 5/4 pi) (* 7/4 pi)))

(define target (make-bitmap 150 150))
(define dc (new bitmap-dc% [bitmap target]))

(draw-face dc)
```

The get-pen and get-brush methods return a DC's current pen and brush, so they can be restored after changing them temporarily for drawing.

Besides grouping settings, a pen% or brush% object includes extra settings that are not available by using set-pen or set-brush directly. For example, a pen or brush can have a *stipple*, which is a bitmap that is used instead of a solid color when drawing. For example, if "water.png" has the image

then it can be loaded with read-bitmap and installed as the stipple for blue-brush:

```
(send blue-brush set-stipple (read-bitmap "water.png"))
(send dc erase)
(draw-face dc)
```

Along similar lines, a `color%` object lets you specify a color through its red, green, and blue components instead of a built-in color name. Due to the way that `color%` initialization is overloaded, use `make-object` instead of `new` to instantiate `color%`, or use the `make-color` function:

```
(define red-pen
  (new pen% [color (make-color 200 100 150)] [width 2]))
(send dc erase)
(draw-face dc)
```

1.3 Transformations

Any coordinates or lengths supplied to drawing commends are transformed by a DC's current transformation matrix. The transformation matrix can scale an image, draw it at an offset, or rotate all drawing. The transformation can be set directly, or the current transformation can be transformed further with methods like `scale`, `translate`, or `rotate`:

```
(send dc erase)
(send dc scale 0.5 0.5)
(draw-face dc)
(send dc rotate (/ pi 2))
(send dc translate 0 150)
(draw-face dc)
(send dc translate 0 -150)
(send dc rotate (/ pi 2))
(send dc translate 150 150)
(draw-face dc)
(send dc translate -150 -150)
(send dc rotate (/ pi 2))
(send dc translate 150 0)
(draw-face dc)
```

Use the `get-transformation` method to get a DC's current transformation, and restore a saved transformation (or any affine transformation) using `set-transformation`.

1.4 Drawing Paths

Drawing functions like `draw-line` and `draw-rectangle` are actually convenience functions for the more general `draw-path` operation. The `draw-path` operation takes a *path*, which describes a set of line segments and curves to draw with the pen and—in the case of closed set of lines and curves—fill with the current brush.

An instance of `dc-path%` holds a path. Conceptually, a path has a current pen position that is manipulated by methods like `move-to`, `line-to`, and `curve-to`. The `move-to` method starts a sub-path, and `line-to` and `curve-to` extend it. The `close` method moves the pen from its current position in a straight line to its starting position, completing the sub-path and forming a closed path that can be filled with the brush. A `dc-path%` object can have multiple closed sub-paths and one final open path, where the open path is drawn only with the pen.

For example,

```
(define zee (new dc-path%))
(send zee move-to 0 0)
(send zee line-to 30 0)
(send zee line-to 0 30)
(send zee line-to 30 30)
```

creates an open path. Drawing this path with a black pen of width 5 and a transparent brush produces

Drawing a single path with three line segments is not the same as drawing three separate lines. When multiple line segments are drawn at once, the corner from one line to the next is shaped according to the pen's join style. The image above uses the default `'round` join style. With `'miter`, line lines are joined with sharp corners:

If the sub-path in `zee` is closed with `close`, then all of the corners are joined, including the corner at the initial point:

```
(send zee close)
```

Using `blue-brush` instead of a transparent brush causes the interior of the path to be filled:

When a sub-path is not closed, it is implicitly closed for brush filling, but left open for pen drawing. When both a pen and brush are available (i.e., not transparent), then the brush is used first, so that the pen draws on top of the brush.

At this point we can't resist showing an extended example using `dc-path%` to draw the Racket logo:

```
(define red-brush (new brush% [stipple (read-bitmap "fire.png")]))

(define left-lambda-path
  (let ([p (new dc-path%)])
    (send p move-to 153 44)
    (send p line-to 161.5 60)
```

```
    (send p curve-to 202.5 49 230 42 245 61)
    (send p curve-to 280.06 105.41 287.5 141 296.5 186)
    (send p curve-to 301.12 209.08 299.11 223.38 293.96 244)
    (send p curve-to 281.34 294.54 259.18 331.61 233.5 375)
    (send p curve-to 198.21 434.63 164.68 505.6 125.5 564)
    (send p line-to 135 572)
    p))

(define left-logo-path
  (let ([p (new dc-path%)])
    (send p append left-lambda-path)
    (send p arc 0 0 630 630 (* 47/72 2 pi) (* 121/360 2 pi) #f)
    p))

(define bottom-lambda-path
  (let ([p (new dc-path%)])
    (send p move-to 135 572)
    (send p line-to 188.5 564)
    (send p curve-to 208.5 517 230.91 465.21 251 420)
    (send p curve-to 267 384 278.5 348 296.5 312)
    (send p curve-to 301.01 302.98 318 258 329 274)
    (send p curve-to 338.89 288.39 351 314 358 332)
    (send p curve-to 377.28 381.58 395.57 429.61 414 477)
    (send p curve-to 428 513 436.5 540 449.5 573)
    (send p line-to 465 580)
    (send p line-to 529 545)
    p))

(define bottom-logo-path
  (let ([p (new dc-path%)])
    (send p append bottom-lambda-path)
    (send p arc 0 0 630 630 (* 157/180 2 pi) (* 47/72 2 pi) #f)
    p))

(define right-lambda-path
  (let ([p (new dc-path%)])
    (send p move-to 153 44)
    (send p curve-to 192.21 30.69 233.21 14.23 275 20)
    (send p curve-to 328.6 27.4 350.23 103.08 364 151)
    (send p curve-to 378.75 202.32 400.5 244 418 294)
    (send p curve-to 446.56 375.6 494.5 456 530.5 537)
    (send p line-to 529 545)
    p))

(define right-logo-path
  (let ([p (new dc-path%)])
```

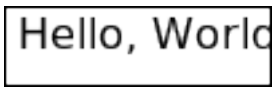
```
    (send p append right-lambda-path)
    (send p arc 0 0 630 630 (* 157/180 2 pi) (* 121/360 2 pi) #t)
    p))

(define lambda-path
  (let ([p (new dc-path%)])
    (send p append left-lambda-path)
    (send p append bottom-lambda-path)
    (let ([t (new dc-path%)])
        (send t append right-lambda-path)
        (send t reverse)
        (send p append t))
    (send p close)
    p))

(define (paint-racket dc)
  (send dc set-pen "black" 0 'transparent)
  (send dc set-brush "white" 'solid)
  (send dc draw-path lambda-path)

  (send dc set-pen "black" 4 'solid)

  (send dc set-brush red-brush)
  (send dc draw-path left-logo-path)
  (send dc draw-path bottom-logo-path)

  (send dc set-brush blue-brush)
  (send dc draw-path right-logo-path))

(define racket-logo (make-bitmap 170 170))
(define dc (new bitmap-dc% [bitmap racket-logo]))

(send dc set-smoothing 'smoothed)
(send dc translate 5 5)
(send dc scale 0.25 0.25)
(paint-racket dc)
```

In addition to the core `move-to`, `line-to`, `curve-to`, and `close` methods, a `dc-path%` includes many convenience methods, such as `ellipse` for adding a closed elliptical sub-path to the path.

1.5 Text

Draw text using the `draw-text` method, which takes a string to draw and a location for the top-left of the drawn text:

```
(define text-target (make-bitmap 100 30))
(define dc (new bitmap-dc% [bitmap text-target]))
(send dc set-brush "white" 'transparent)

(send dc draw-rectangle 0 0 100 30)
(send dc draw-text "Hello, World!" 5 1)
```



The font used to draw text is determined by the DC's current font. A font is described by a `font%` object and installed with `set-font`. The color of drawn text which is separate from either the pen or brush, can be set using `set-text-foreground`.

```
(send dc erase)
(send dc set-font (make-font #:size 14 #:family 'roman
                             #:weight 'bold))
(send dc set-text-foreground "blue")
```

```
(send dc draw-rectangle 0 0 100 30)
(send dc draw-text "Hello, World!" 5 1)
```

To compute the size that will be used by drawn text, use `get-text-extent`, which returns four values: the total width, total height, difference between the baseline and total height, and extra space (if any) above the text in a line. For example, the result of `get-text-extent` can be used to position text within the center of a box:

```
(send dc erase)
(send dc draw-rectangle 0 0 100 30)
(define-values (w h d a) (send dc get-text-extent "Hello,
World!"))
(send dc draw-text "Hello, World!" (/ (- 100 w) 2) (/ (- 30 h) 2))
```

1.6 Alpha Channels and Alpha Blending

When you create or `erase` a bitmap, the content is nothing. "Nothing" isn't the same as white; it's the absence of drawing. For example, if you take `text-target` from the previous section and copy it onto another DC using `draw-bitmap`, then the black rectangle and blue text is transferred, and the background is left alone:

```
(define new-target (make-bitmap 100 30))
(define dc (new bitmap-dc% [bitmap new-target]))
(send dc set-pen "black" 1 'transparent)
(send dc set-brush "pink" 'solid)

(send dc draw-rectangle 0 0 100 30)
(send dc draw-bitmap text-target 0 0)
```

The information about which pixels of a bitmap are drawn (as opposed to "nothing") is the bitmap's *alpha channel*. Not all DCs keep an alpha channel, but bitmaps created with `make-bitmap` keep an alpha channel by default. Bitmaps loaded with `read-bitmap` preserve transparency in the image file through the bitmap's alpha channel.

An alpha channel isn't all or nothing. When the edges text is anti-aliased by `draw-text`, for example, the pixels are partially transparent. When the pixels are transferred to another DC, the partially transparent pixel is blended with the target pixel in a process called *alpha blending*. Furthermore, a DC has an alpha value that is applied to all drawing operations; an alpha value of `1.0` corresponds to solid drawing, an alpha value of `0.0` makes the drawing have no effect, and values in between make the drawing translucent.

For example, setting the DC's alpha to `0.25` before calling `draw-bitmap` causes the blue and black of the "Hello, World!" bitmap to be quarter strength as it is blended with the destination image:

```
(send dc erase)
(send dc draw-rectangle 0 0 100 30)
(send dc set-alpha 0.25)
(send dc draw-bitmap text-target 0 0)
```

1.7 Clipping

In addition to tempering the opacity of drawing operations, a DC has a *clipping region* that constrains all drawing to inside the region. In the simplest case, a clipping region corresponds to a closed path, but it can also be the union, intersection, subtraction, or exclusive-or of two paths.

For example, a clipping region could be set to three circles to clip the drawing of a rectangle (with the 0.25 alpha still in effect):

```
(define r (new region%))
(let ([p (new dc-path%)])
  (send p ellipse 0 0 35 30)
  (send p ellipse 35 0 30 30)
  (send p ellipse 65 0 35 30)
  (send r set-path p))
(send dc set-clipping-region r)
(send dc set-brush "green" 'solid)
(send dc draw-rectangle 0 0 100 30)
```

The clipping region can be viewed as a convenient alternative to path filling or drawing with

stipples. Conversely, stippled drawing can be viewed as a convenience alternative to clipping repeated calls of draw-bitmap.

Combining regions with pen% objects that have gradients, however, is more than just a convenience, as it allows us to draw shapes in combinations we could not otherwise draw. To illustrate, here is some code that draws text with its reflection below it.

```
; First compute the size of the text we're going to draw,
; using a small bitmap that we never draw into.
(define bdc (new bitmap-dc% [bitmap (make-bitmap 1 1)]))
(define str "Racketeers, ho!")
(define the-font (make-font #:size 24 #:family 'swiss
                            #:weight 'bold))
(define-values (tw th)
  (let-values ([(tw th ta td)
                (send dc get-text-extent str the-font)])
    (values (inexact->exact (ceiling tw))
            (inexact->exact (ceiling th)))))

; Now we can create a correctly sized bitmap to
; actually draw into and enable smoothing.
(send bdc set-bitmap (make-bitmap tw (* th 2)))
(send bdc set-smoothing 'smoothed)

; next, build a path that contains the outline of the text
(define upper-path (new dc-path%))
(send upper-path text-outline the-font str 0 0)

; next, build a path that contains the mirror image
; outline of the text
(define lower-path (new dc-path%))
(send lower-path text-outline the-font str 0 0)
(send lower-path transform (vector 1 0 0 -1 0 0))
(send lower-path translate 0 (* 2 th))

; This helper accepts a path, sets the clipping region
; of bdc to be the path (but in region form), and then
; draws a big rectangle over the whole bitmap.
; The brush will be set differently before each call to
; draw-path, in order to draw the text and then to draw
; the shadow.
(define (draw-path path)
  (define rgn (new region%))
  (send rgn set-path path)
  (send bdc set-clipping-region rgn)
  (send bdc set-pen "white" 1 'transparent)
```

```
  (send bdc draw-rectangle 0 0 tw (* th 2))
  (send bdc set-clipping-region #f))

; Now we just draw the upper-path with a solid brush
(send bdc set-brush "black" 'solid)
(draw-path upper-path)

; To draw the shadow, we set up a brush that has a
; linear gradient over the portion of the bitmap
; where the shadow goes
(define stops
  (list (list 0 (make-color 0 0 0 0.4))
        (list 1 (make-color 0 0 0 0.0))))
(send bdc set-brush
      (new brush%
           [gradient
            (new linear-gradient%
                 [x0 0]
                 [y0 th]
                 [x1 0]
                 [y1 (* th 2)]
                 [stops stops])]))
(draw-path lower-path)
```

And now the bitmap in `bdc` has "Racketeers, ho!" with a mirrored version below it.

1.8 Portability and Bitmap Variants

Drawing effects are not completely portable across platforms, across different classes that implement `dc<%>`, or different kinds of bitmaps. Fonts and text, especially, can vary across platforms and types of DC, but so can the precise set of pixels touched by drawing a line.

Different kinds of bitmaps can produce different results:

- Drawing to a bitmap produced by `make-bitmap` (or instantiated from `bitmap%`)

draws in the most consistent way across platforms.

- Drawing to a bitmap produced by make-platform-bitmap uses platform-specific drawing operations as much as possible. On Windows, however, a bitmap produced by make-platform-bitmap has no alpha channel, and it uses more constrained resources than one produced by make-bitmap (due to a system-wide, per-process GDI limit).

  As an example of platform-specific difference, text is smoothed by default with subpixel anti-aliasing on Mac OS X, while text smoothing in a make-bitmap result uses only grays. Line or curve drawing may touch different pixels than in a bitmap produced by make-bitmap, and bitmap scaling may differ.

  A possible approach to dealing with the GDI limit under Windows is to draw into the result of a make-platform-bitmap and then copy the contents of the drawing into the result of a make-bitmap. This approach preserves the drawing results of make-platform-bitmap, but it retains constrained resources only during the drawing process.

- Drawing to a bitmap produced by make-screen-bitmap from racket/gui/base uses the same platform-specific drawing operations as drawing into a canvas% instance. A bitmap produced by make-screen-bitmap uses the same platform-specific drawing as make-platform-bitmap on Windows or Mac OS X, but possibly scaled, and it may be sensitive to the X11 server on Unix.

  On Mac OS X, when the main screen is in Retina mode (at the time that the bitmap is created), the bitmap is also internally scaled so that one drawing unit uses two pixels. Similarly, on Windows, when the main display's text scale is configured at the operating-system level (see §1.8 "Screen Resolution and Text Scaling"), the bitmap is internally scaled, where common configurations map a drawing unit to 1.25, 1.5, or 2 pixels.

  Use make-screen-bitmap when drawing to a bitmap as an offscreen buffer before transferring an image to the screen, or when consistency with screen drawing is needed for some other reason.

- A bitmap produced by make-bitmap in canvas% from racket/gui/base is like a bitmap from make-screen-bitmap, but on Mac OS X, the bitmap is optimized for drawing to the screen (by taking advantage of system APIs that can, in turn, take advantage of graphics hardware).

  Use make-bitmap in canvas% for similar purposes as make-screen-bitmap, particularly when the bitmap will be drawn later to a known target canvas.

2 bitmap%

```
bitmap% : class?
  superclass: object%
```

A bitmap% object is a pixel-based image, either monochrome, color, or color with an alpha channel. See also make-bitmap, make-platform-bitmap, make-screen-bitmap (from racket/gui/base), make-bitmap in canvas% (from racket/gui/base), and §1.8 “Portability and Bitmap Variants”.

A bitmap has a *backing scale*, which is the number of pixels that correspond to a drawing unit for the bitmap, either when the bitmap is used as a target for drawing or when the bitmap is drawn into another context. For example, on Mac OS X when the main monitor is in Retina mode, make-screen-bitmap returns a bitmap whose backing scale is 2.0. On Windows, the backing scale of a screen bitmap corresponds to the system-wide text scale (see §1.8 “Screen Resolution and Text Scaling”). A monochrome bitmap always has a backing scale of 1.0.

A bitmap is convertible to 'png-bytes through the file/convertible protocol.

```
(make-object bitmap%  width
                      height
                     [monochrome?
                      alpha?
                      backing-scale]) → (is-a?/c bitmap%)
  width : exact-positive-integer?
  height : exact-positive-integer?
  monochrome? : any/c = #f
  alpha? : any/c = #f
  backing-scale : (>/c 0.0) = 1.0
(make-object bitmap%  in
                     [kind
                      bg-color
                      complain-on-failure?
                      backing-scale])      → (is-a?/c bitmap%)
  in : (or/c path-string? input-port?)
  kind : (or/c 'unknown 'unknown/mask 'unknown/alpha = 'unknown
               'gif 'gif/mask 'gif/alpha
               'jpeg 'jpeg/alpha
               'png 'png/mask 'png/alpha
               'xbm 'xbm/alpha 'xpm 'xpm/alpha
               'bmp 'bmp/alpha)
  bg-color : (or/c (is-a?/c color%) #f) = #f
  complain-on-failure? : any/c = #f
  backing-scale : (>/c 0.0) = 1.0
(make-object bitmap% bits width height) → (is-a?/c bitmap%)
```

```
bits : bytes?
width : exact-positive-integer?
height : exact-positive-integer?
```

The make-bitmap, make-monochrome-bitmap, and read-bitmap functions create bitmap% instances, but they are also preferred over using make-object with bitmap% directly, because the functions are less overloaded and they enable alpha channels by default. See also §1.8 "Portability and Bitmap Variants".

When *width* and *height* are provided: Creates a new bitmap. If *monochrome?* is true, the bitmap is monochrome; if *monochrome?* is #f and *alpha?* is true, the bitmap has an alpha channel; otherwise, the bitmap is color without an alpha channel. The *backing-scale* argument sets the bitmap's backing scale, and it must be 1.0 if monochrome is true. The initial content of the bitmap is "empty": all white, and with zero alpha in the case of a bitmap with an alpha channel.

When *in* is provided: Creates a bitmap from a file format, where *kind* specifies the format. See load-file for details. The *backing-scale* argument sets the bitmap's backing scale, so that the bitmap's size (as reported by get-width and get-height) is the ceiling of the bitmap's size from *in* divided by *backing-scale*; the backing scale must be 1.0 if the bitmap is monocrhome or loaded with a mask.

When a *bits* byte string is provided: Creates a monochrome bitmap from an array of bit values, where each byte in *bits* specifies eight bits, and padding bits are added so that each bitmap line starts on a character boundary. A 1 bit value indicates black, and 0 indicates white. If *width* times *height* is larger than 8 times the length of *bits*, an exn:fail:contract exception is raised.

Changed in version 1.1 of package draw-lib: Added the *backing-scale* optional arguments.

```
(send a-bitmap get-argb-pixels  x
                                y
                                width
                                height
                                pixels
                               [just-alpha?
                                pre-multiplied?
                                #:unscaled? unscaled?]) → void?
  x : real?
  y : real?
  width : exact-nonnegative-integer?
  height : exact-nonnegative-integer?
  pixels : (and/c bytes? (not/c immutable?))
  just-alpha? : any/c = #f
  pre-multiplied? : any/c = #f
  unscaled? : any/c = #f
```

Produces the same result as get-argb-pixels in bitmap-dc% when *unscaled?* is #f, but the bitmap does not have to be selected into the DC (and this method works even if the bitmap is selected into another DC, attached as a button label, etc.).

If the bitmap has a backing scale other than 1.0 and *unscaled?* is true, then the result corresponds to the bitmap's pixels ignoring the backing scale. In that case, *x*, *y*, *width*, and *height* are effectively in pixels instead of drawing units.

Changed in version 1.1 of package draw-lib: Added the #:unscaled? optional argument.

```
(send a-bitmap get-backing-scale) → (>/c 0.0)
```

Returns the bitmap's backing scale.

Added in version 1.1 of package draw-lib.

```
(send a-bitmap get-depth) → exact-nonnegative-integer?
```

Gets the color depth of the bitmap, which is 1 for a monochrome bitmap and 32 for a color bitmap. See also is-color?.

```
(send a-bitmap get-handle) → cpointer?
```

Returns a low-level handle to the bitmap content. Currently, on all platforms, a handle is a cairo_surface_t. For a bitmap created with make-bitmap, the handle is specifically a Cairo image surface.

```
(send a-bitmap get-height) → exact-positive-integer?
```

Gets the height of the bitmap in drawing units (which is the same as pixels if the backing scale is 1.0).

```
(send a-bitmap get-loaded-mask) → (or/c (is-a?/c bitmap%) #f)
```

Returns a mask bitmap that is stored with this bitmap.

When a GIF file is loaded with 'gif/mask or 'unknown/mask and the file contains a transparent "color," a mask bitmap is generated to identify the transparent pixels. The mask bitmap is monochrome, with white pixels where the loaded bitmap is transparent and black pixels everywhere else.

When a PNG file is loaded with 'png/mask or 'unknown/mask and the file contains a mask or alpha channel, a mask bitmap is generated to identify the mask or alpha channel. If the file contains a mask or an alpha channel with only extreme values, the mask bitmap is monochrome, otherwise it is grayscale (representing the alpha channel inverted).

When an XPM file is loaded with `'xpm/mask` or `'unknown/mask`, a mask bitmap is generated to indicate which pixels are set.

When `'unknown/alpha` and similar modes are used to load a bitmap, transparency information is instead represented by an alpha channel, not by a mask bitmap.

Unlike an alpha channel, the mask bitmap is *not* used automatically by drawing routines. The mask bitmap can be extracted and supplied explicitly as a mask (e.g., as the sixth argument to `draw-bitmap`). The mask bitmap is used by `save-file` when saving a bitmap as `'png` if the mask has the same dimensions as the saved bitmap. The mask bitmap is also used automatically when the bitmap is a control label.

```
(send a-bitmap get-width) → exact-positive-integer?
```

Gets the width of the bitmap in drawing units (which is the same as pixels of the backing scale is 1.0).

```
(send a-bitmap has-alpha-channel?) → boolean?
```

Returns `#t` if the bitmap has an alpha channel, `#f` otherwise.

```
(send a-bitmap is-color?) → boolean?
```

Returns `#f` if the bitmap is monochrome, `#t` otherwise.

```
(send a-bitmap load-file  in
                          [kind
                           bg-color
                           complain-on-failure?]) → boolean?
  in : (or/c path-string? input-port?)
  kind : (or/c 'unknown 'unknown/mask 'unknown/alpha = 'unknown
               'gif 'gif/mask 'gif/alpha
               'jpeg 'jpeg/alpha
               'png 'png/mask 'png/alpha
               'xbm 'xbm/alpha 'xpm 'xpm/alpha
               'bmp 'bmp/alpha)
  bg-color : (or/c (is-a?/c color%) #f) = #f
  complain-on-failure? : any/c = #f
```

Loads a bitmap from a file format that read from *in*, unless the bitmap was produced by `make-platform-bitmap`, `make-screen-bitmap`, or `make-bitmap` in `canvas%` (in which case an `exn:fail:contract` exception is raised). If the bitmap is in use by a `bitmap-dc%` object or a control, the image data is not loaded. The bitmap changes its size and depth to match that of the loaded image. If an error is encountered when reading the file format, an exception is raised only if *complain-on-failure?* is true (which is *not* the default).

The *kind* argument specifies the file's format:

- `'unknown` — examine the file to determine its format; creates either a monochrome or color bitmap without an alpha channel
- `'unknown/mask` — like `'unknown`, but see `get-loaded-mask`
- `'unknown/alpha` — like `'unknown`, but if the bitmap is color, it has an alpha channel, and transparency in the image file is recorded in the alpha channel
- `'gif` — load a GIF bitmap file, creating a color bitmap
- `'gif/mask` — like `'gif`, but see `get-loaded-mask`
- `'gif/alpha` — like `'gif`, but with an alpha channel
- `'jpeg` — load a JPEG bitmap file, creating a color bitmap
- `'jpeg/alpha` — like `'jpeg`, but with an alpha channel
- `'png` — load a PNG bitmap file, creating a color or monochrome bitmap
- `'png/mask` — like `'png`, but see `get-loaded-mask`
- `'png/alpha` — like `'png`, but always color and with an alpha channel
- `'xbm` — load an X bitmap (XBM) file; creates a monochrome bitmap
- `'xbm/alpha` — like `'xbm`, but creates a color bitmap with an alpha channel
- `'xpm` — load an XPM bitmap file, creating a color bitmap
- `'xpm/alpha` — like `'xpm`, but with an alpha channel
- `'bmp` — load a Windows bitmap file, creating a color bitmap
- `'bmp/alpha` — like `'bmp`, but with an alpha channel

An XBM image is always loaded as a monochrome bitmap. A 1-bit grayscale PNG without a mask or alpha channel is also loaded as a monochrome bitmap. An image in any other format is always loaded as a color bitmap.

For PNG loading, if *`bg-color`* is not `#f`, then it is combined with the file's alpha channel or mask (if any) while loading the image; in this case, no separate mask bitmap is generated and the alpha channel fills the bitmap, even if `'unknown/mask`, `'png/mask` is specified for the format. If the format is specified as `'unknown` or `'png` and *`bg-color`* is not specified, the PNG file is consulted for a background color to use for loading, and white is used if no background color is indicated in the file.

In all PNG-loading modes, gamma correction is applied when the file provides a gamma value, otherwise gamma correction is not applied. The current display's gamma factor is determined by the `SCREEN_GAMMA` environment variable if it is defined. If the preference and environment variable are both undefined, a platform-specific default is used.

After a bitmap is created, `load-file` can be used only if the bitmap's backing scale is `1.0`.

```
(send a-bitmap make-dc) → (is-a?/c bitmap-dc%)
```

Return (make-object bitmap-dc% this).

```
(send a-bitmap ok?) → boolean?
```

Returns #t if the bitmap is valid in the sense that an image file was loaded successfully. If ok? returns #f, then drawing to or from the bitmap has no effect.

```
(send a-bitmap save-file  name
                          kind
                         [quality
                          #:unscaled? unscaled?]) → boolean?
  name : (or/c path-string? output-port?)
  kind : (or/c 'png 'jpeg 'xbm 'xpm 'bmp)
  quality : (integer-in 0 100) = 75
  unscaled? : any/c = #f
```

Writes a bitmap to the named file or output stream.

The *kind* argument determined the type of file that is created, one of:

- 'png — save a PNG file
- 'jpeg — save a JPEG file
- 'xbm — save an X bitmap (XBM) file
- 'xpm — save an XPM bitmap file
- 'bmp — save a Windows bitmap file

The *quality* argument is used only for saving as 'jpeg, in which case it specifies the trade-off between image precision (high quality matches the content of the bitmap% object more precisely) and size (low quality is smaller).

When saving as 'png, if get-loaded-mask returns a bitmap of the same size as this one, a grayscale version is included in the PNG file as the alpha channel.

A monochrome bitmap saved as 'png without a mask bitmap produces a 1-bit grayscale PNG file (which, when read with load-file, creates a monochrome bitmap% object.)

If the bitmap has a backing scale other than 1.0, then it is effectively converted to a single pixel per drawing unit before saving unless *unscaled?* is true.

Changed in version 1.1 of package draw-lib: Added the #:unscaled? optional argument.

```
(send a-bitmap set-argb-pixels  x
                                y
                                width
                                height
                                pixels
                               [just-alpha?
                                pre-multiplied?
                                #:unscaled? unscaled?]) → void?
  x : real?
  y : real?
  width : exact-nonnegative-integer?
  height : exact-nonnegative-integer?
  pixels : bytes?
  just-alpha? : any/c = #f
  pre-multiplied? : any/c = #f
  unscaled? : any/c = #f
```

The same as `set-argb-pixels` in `bitmap-dc%` when *unscaled?* is `#f`, but the bitmap does not have to be selected into the DC.

If the bitmap has a backing scale other than `1.0` and *unscaled?* is true, then pixel values are installed ignoring the backing scale. In that case, *x*, *y*, *width*, and *height* are effectively in pixels instead of drawing units.

Changed in version 1.1 of package `draw-lib`: Added the `#:unscaled?` optional argument.

```
(send a-bitmap set-loaded-mask mask) → void?
  mask : (is-a?/c bitmap%)
```

See `get-loaded-mask`.

3 bitmap-dc%

```
bitmap-dc% : class?
  superclass: object%
  extends: dc<%>
```

A bitmap-dc% object allows drawing directly into a bitmap. A bitmap% object must be supplied at initialization or installed into a bitmap DC using set-bitmap before any other method of the DC is called, except get-text-extent, get-char-height, or get-char-width. If any other bitmap-dc% method is called before a bitmap is selected, the method call is ignored.

Drawing to a bitmap-dc% with a color bitmap is guaranteed to produce the same result as drawing into a canvas% instance (with appropriate clipping and offsets). Thus, a bitmap-dc% can be used for offscreen staging of canvas content.

```
(new bitmap-dc% [bitmap bitmap]) → (is-a?/c bitmap-dc%)
  bitmap : (or/c (is-a?/c bitmap%) #f)
```

Creates a new bitmap DC. If *bitmap* is not #f, it is installed into the DC so that drawing commands on the DC draw to *bitmap*. Otherwise, no bitmap is installed into the DC and set-bitmap must be called before any other method of the DC is called.

```
(send a-bitmap-dc draw-bitmap-section-smooth  source
                                              dest-x
                                              dest-y
                                              dest-width
                                              dest-height
                                              src-x
                                              src-y
                                              src-width
                                              src-height
                                             [style
                                              color
                                              mask])
 → boolean?
  source : (is-a?/c bitmap%)
  dest-x : real?
  dest-y : real?
  dest-width : (and/c real? (not/c negative?))
  dest-height : (and/c real? (not/c negative?))
  src-x : real?
  src-y : real?
  src-width : (and/c real? (not/c negative?))
  src-height : (and/c real? (not/c negative?))
  style : (or/c 'solid 'opaque 'xor) = 'solid
```

```
  color : (is-a?/c color%)
        = (send the-color-database find-color "black")
  mask : (or/c (is-a?/c bitmap%) #f) = #f
```

The same as draw-bitmap-section, except that *dest-width* and *dest-height* cause the DC's transformation to be adjusted while drawing the bitmap so that the bitmap is scaled; and, if the DC's smoothing mode is 'unsmoothed, it is changed to 'aligned while drawing.

```
(send a-bitmap-dc get-argb-pixels  x
                                   y
                                   width
                                   height
                                   pixels
                                  [just-alpha?
                                   pre-multiplied?]) → void?
  x : real?
  y : real?
  width : exact-nonnegative-integer?
  height : exact-nonnegative-integer?
  pixels : (and/c bytes? (not/c immutable?))
  just-alpha? : any/c = #f
  pre-multiplied? : any/c = #f
```

Gets a rectangle of pixels in the bitmap, subject to the same rules and performance characteristics of get-pixel, except that the block get is likely to be faster than the sequence of individual gets. Also, the bitmap% class also provides the same method directly, so it is not necessary to select a bitmap into a DC to extracts its pixel values.

The pixel RGB values and alphas are copied into *pixels* (or just alpha values if *just-alpha?* is true). The first byte represents an alpha value of the pixel at (*x*, *y*), the second byte represents a red value of the pixel at (*x*, *y*), the third byte is the green value, etc. In this way, the first *width* * *height* * 4 bytes of *pixels* are set to reflect the current pixel values in the DC. The pixels are in row-major order, left to right then top to bottom.

If the bitmap has an alpha channel, then the alpha value for each pixel is always set in *pixels*. If *just-alpha?* is false and the bitmap does not have an alpha channel, then the alpha value for each pixel is set to 255. If *just-alpha?* is true, then *only* the alpha value is set for each pixel; if the bitmap has no alpha channel, then the alpha value is based on each pixel's inverted RGB average. Thus, when a bitmap has a separate mask bitmap, the same *pixels* byte string is in general filled from two bitmaps: one (the main image) for the pixel values and one (the mask) for the alpha values.

If *pre-multiplied?* is true, *just-alpha?* is false, and the bitmap has an alpha channel, then RGB values in the result are scaled by the corresponding alpha value (i.e., multiplied by the alpha value and then divided by 255).

If the bitmap has a backing scale other than 1.0, the result of get-argb-pixels is as if the bitmap is drawn to a bitmap with a backing scale of 1.0 and the pixels of the target bitmap are returned.

```
(send a-bitmap-dc get-bitmap) → (or/c (is-a?/c bitmap%) #f)
```

Gets the bitmap currently installed in the DC, or #f if no bitmap is installed. See set-bitmap for more information.

```
(send a-bitmap-dc get-pixel x y color) → boolean?
  x : real?
  y : real?
  color : (is-a?/c color%)
```

Fills *color* with the color of the current pixel at position (*x*, *y*) in the drawing context. If the color is successfully obtained, the return value is #t, otherwise the result is #f.

```
(send a-bitmap-dc set-argb-pixels  x
                                   y
                                   width
                                   height
                                   pixels
                                  [just-alpha?
                                   pre-multiplied?]) → void?
  x : real?
  y : real?
  width : exact-nonnegative-integer?
  height : exact-nonnegative-integer?
  pixels : bytes?
  just-alpha? : any/c = #f
  pre-multiplied? : any/c = #f
```

Sets a rectangle of pixels in the bitmap, unless the DC's current bitmap was produced by make-screen-bitmap or make-bitmap in canvas% (in which case an exn:fail:contract exception is raised).

The pixel RGB values are taken from *pixels*. The first byte represents an alpha value, the second byte represents a red value to used for the pixel at (*x*, *y*), the third byte is a blue value, etc. In this way, the first *width* * *height* * 4 bytes of *pixels* determine the new pixel values in the DC. The pixels are in row-major order, left to right then top to bottom.

If *just-alpha?* is false, then the alpha value for each pixel is used only if the DC's current bitmap has an alpha channel. If *just-alpha?* is true and the bitmap has an alpha channel, then the bitmap is not modified. If *just-alpha?* is true and the bitmap has no alpha channel, then each pixel is set based *only* on the alpha value, but inverted to serve as a mask. Thus, when working with bitmaps that have an associated mask bitmap instead of an alpha channel,

the same *pixels* byte string is used with two bitmaps: one (the main image) for the pixel values and one (the mask) for the alpha values.

If *pre-multiplied?* is true, *just-alpha?* is false, and the bitmap has an alpha channel, then RGB values in *pixels* are interpreted as scaled by the corresponding alpha value (i.e., multiplied by the alpha value and then divided by 255). If an R, G, or B value is greater than its corresponding alpha value (which is not possible if the value is properly scaled), then it is effectively reduced to the alpha value.

If the bitmap has a backing scale other than `1.0`, then *pixels* are effectively scaled by the backing scale to obtain pixel values that are installed into the bitmap.

```
(send a-bitmap-dc set-bitmap bitmap) → void?
  bitmap : (or/c (is-a?/c bitmap%) #f)
```

Installs a bitmap into the DC, so that drawing operations on the bitmap DC draw to the bitmap. A bitmap is removed from a DC by setting the bitmap to `#f`.

A bitmap can be selected into at most one bitmap DC, and only when it is not used by a control (as a label) or in a `pen%` or `brush%` (as a stipple). If the argument to `set-bitmap` is already in use by another DC, a control, a `pen%`, or a `brush%`, an `exn:fail:contract` exception is raised.

```
(send a-bitmap-dc set-pixel x y color) → void?
  x : real?
  y : real?
  color : (is-a?/c color%)
```

Sets a pixel in the bitmap.

The current clipping region might not affect the pixel change. Under X, interleaving drawing commands with `set-pixel` calls (for the same `bitmap-dc%` object) incurs a substantial performance penalty, except for interleaved calls to `get-pixel`, `get-argb-pixels`, and `set-argb-pixels`.

4 brush%

```
brush% : class?
  superclass: object%
```

A brush is a drawing tool with a color and a style that is used for filling in areas, such as the interior of a rectangle or ellipse. In a monochrome destination, all non-white brushes are drawn as black.

In addition to its color and style, a brush can have a *brush stipple* bitmap. Painting with a stipple brush is similar to calling `draw-bitmap` with the stipple bitmap in the filled region.

As an alternative to a color, style, and stipple, a brush can have a *gradient* that is a `linear-gradient%` or `radial-gradient%`. When a brush has a gradient and the target for drawing is not monochrome, then other brush settings are ignored. With a gradient, for each point in a drawing destination, the gradient associates a color to the point based on starting and ending colors and starting and ending lines (for a linear gradient) or circles (for a radial gradient); a gradient-assigned color is applied for each point that is touched when drawing with the brush.

By default, coordinates in a stipple or gradient are transformed by the drawing context's transformation when the brush is used, but a brush can have its own *brush transformation* that is used, instead. A brush transformation has the same representation and meaning as for `get-transformation` in `dc<%>`.

A *brush style* is one of the following (but is ignored if the brush has a gradient and the target is not monochrome):

- `'transparent` — Draws with no effect (on the interior of the drawn shape).
- `'solid` — Draws using the brush's color. If a monochrome brush stipple is installed into the brush, black pixels from the stipple are transferred to the destination using the brush's color, and white pixels from the stipple are not transferred.
- `'opaque` — The same as `'solid` for a color brush stipple. For a monochrome stipple, white pixels from the stipple are transferred to the destination using the destination's background color.
- `'xor` — The same as `'solid`, accepted only for partial backward compatibility.
- `'hilite` — Draws with black and a `0.3` alpha.
- `'panel` — The same as `'solid`, accepted only for partial backward compatibility.
- The following modes correspond to built-in brush stipples drawn in `'solid` mode:
  - `'bdiagonal-hatch` — diagonal lines, top-left to bottom-right
  - `'crossdiag-hatch` — crossed diagonal lines

- `'fdiagonal-hatch` — diagonal lines, top-right to bottom-left
- `'cross-hatch` — crossed horizontal and vertical lines
- `'horizontal-hatch` — horizontal lines
- `'vertical-hatch` — vertical lines

However, when a specific brush stipple is installed into the brush, the above modes are ignored and `'solid` is used, instead.

To draw outline shapes (such as unfilled boxes and ellipses), use the `'transparent` brush style.

To avoid creating multiple brushes with the same characteristics, use the global `brush-list%` object `the-brush-list`, or provide a color and style to `set-brush` in `dc<%>`.

See also `make-brush`.

```
(new brush%
   [[color color]
    [style style]
    [stipple stipple]
    [gradient gradient]
    [transformation transformation]])
→ (is-a?/c brush%)
 color : (or/c string? (is-a?/c color%)) = "black"
 style : brush-style/c = 'solid
 stipple : (or/c #f (is-a?/c bitmap%)) = #f
 gradient : (or/c #f                          = #f
                  (is-a?/c linear-gradient%)
                  (is-a?/c radial-gradient%))
 transformation :  (or/c #f (vector/c (vector/c real? real? real?
                                                 real? real? real?)
                                      real? real? real? real? real?))
                 = #f
```

Creates a brush with the given color, brush style, brush stipple, gradient, and brush transformation (which is kept only if the gradient or stipple is non-`#f`). For the case that the color is specified using a name, see `color-database<%>` for information about color names; if the name is not known, the brush's color is black.

```
(send a-brush get-color) → (is-a?/c color%)
```

Returns the brush's color.

```
(send a-brush get-gradient) → (or/c (is-a?/c linear-gradient%)
                                    (is-a?/c radial-gradient%)
                                    #f)
```

Gets the gradient, or #f if the brush has no gradient.

```
(send a-brush get-handle) → (or/c cpointer? #f)
```

Returns a low-level handle for the brush content, but only for brushes created with make-handle-brush; otherwise, the result is #f.

```
(send a-brush get-stipple) → (or/c (is-a?/c bitmap%) #f)
```

Gets the brush stipple bitmap, or #f if the brush has no stipple.

```
(send a-brush get-style) → brush-style/c
```

Returns the brush style. See brush% for information about brush styles.

```
(send a-brush get-transformation)
 → (or/c #f (vector/c (vector/c real? real? real? real? real? real?)
                      real? real? real? real? real?))
```

Returns the brush's brush transformation, if any.

If a brush with a stipple or gradient also has a transformation, then the transformation applies to the stipple or gradient's coordinates instead of the target drawing context's transformation; otherwise, the target drawing context's transformation applies to stipple and gradient coordinates.

```
(send a-brush is-immutable?) → boolean?
```

Returns #t if the brush object is immutable.

```
(send a-brush set-color color) → void?
  color : (is-a?/c color%)
(send a-brush set-color color-name) → void?
  color-name : string?
(send a-brush set-color red green blue) → void?
  red : byte?
  green : byte?
  blue : byte?
```

Sets the brush's color. A brush cannot be modified if it was obtained from a brush-list% or while it is selected into a drawing context.

For the case that the color is specified using a string, see color-database<%> for information about color names.

```
(send a-brush set-stipple  bitmap
                           [transformation]) → void?
  bitmap : (or/c (is-a?/c bitmap%) #f)
  transformation : (or/c #f (vector/c (vector/c real? real? real?
                                                real? real? real?)
                                      real? real? real? real? real?))
                 = #f
```

Sets or removes the brush stipple bitmap, where `#f` removes the stipple. The brush transformation is set at the same time to *transformation*. See brush% for information about drawing with stipples.

If *bitmap* is modified while is associated with a brush, the effect on the brush is unspecified. A brush cannot be modified if it was obtained from a brush-list% or while it is selected into a drawing context.

```
(send a-brush set-style style) → void?
  style : brush-style/c
```

Sets the brush style. See brush% for information about the possible styles.

A brush cannot be modified if it was obtained from a brush-list% or while it is selected into a drawing context.

5 brush-list%

```
brush-list% : class?
  superclass: object%
```

A brush-list% object maintains a list of brush% objects to avoid creating brushes repeatedly. A brush% object in a brush list cannot be mutated.

A global brush list, the-brush-list, is created automatically.

```
(new brush-list%) → (is-a?/c brush-list%)
```

Creates an empty brush list.

```
(send a-brush-list find-or-create-brush color
                                        style)
 → (is-a?/c brush%)
  color : (or/c string? (is-a?/c color%))
  style : (or/c 'transparent 'solid 'opaque
                'xor 'hilite 'panel
                'bdiagonal-hatch 'crossdiag-hatch
                'fdiagonal-hatch 'cross-hatch
                'horizontal-hatch 'vertical-hatch)
(send a-brush-list find-or-create-brush color-name
                                        style)
 → (or/c (is-a?/c brush%) #f)
  color-name : string?
  style : (or/c 'transparent 'solid 'opaque
                'xor 'hilite 'panel
                'bdiagonal-hatch 'crossdiag-hatch
                'fdiagonal-hatch 'cross-hatch
                'horizontal-hatch 'vertical-hatch)
```

Finds a brush of the given specification, or creates one and adds it to the list. See brush% for a further explanation of the arguments, which are the same as brush%'s initialization arguments.

6 color%

```
color% : class?
  superclass: object%
```

A color is an object representing a red-green-blue (RGB) combination of primary colors plus an "alpha" for opacity. Each red, green, or blue component of the color is an exact integer in the range 0 to 255, inclusive, and the alpha value is a real number between 0 and 1, inclusive. For example, (0, 0, 0, 1.0) is solid black, (255, 255, 255, 1.0) is solid white, (255, 0, 0, 1.0) is solid red, and (255, 0, 0, 0.5) is translucent red.

See color-database<%> for information about obtaining a color object using a color name, and see also make-color.

```
(make-object color%) → (is-a?/c color%)
(make-object color% red green blue [alpha]) → (is-a?/c color%)
  red : byte?
  green : byte?
  blue : byte?
  alpha : (real-in 0 1) = 1.0
(make-object color% color-name-or-obj) → (is-a?/c color%)
  color-name-or-obj : (or/c string? (is-a?/c color%))
```

Creates a new color.

If three or four arguments are supplied to the constructor, the color is created with those RGB and alpha values.

If a single color% object is supplied, the color is created with the same RGB and alpha values as the given color.

If a string is supplied, then it is passed to the color-database<%>'s find-color method to find a color (signaling an error if the color is not in the color-database<%>'s get-names method's result).

If no arguments are supplied, the new color is black.

```
(send a-color red) → byte?
```

Returns the red component of the color.

```
(send a-color green) → byte?
```

Returns the green component of the color.

```
(send a-color blue) → byte?
```

Returns the blue component of the color.

```
(send a-color alpha) → (real-in 0 1)
```

Returns the alpha component (i.e., opacity) of the color.

```
(send a-color set red green blue [alpha]) → void?
  red : byte?
  green : byte?
  blue : byte?
  alpha : (real-in 0 1) = 1.0
```

Sets the four (red, green, blue, and alpha) component values of the color.

```
(send a-color copy-from src) → (is-a?/c color%)
  src : (is-a?/c color%)
```

Copies the RGB values of another color object to this one, returning this object as the result.

```
(send a-color is-immutable?) → boolean?
```

Returns `#t` if the color object is immutable.

See also `make-color` and `find-color` in `color-database<%>`.

```
(send a-color ok?) → #t
```

Returns `#t` to indicate that the color object is valid.

(Historically, the result could be `#f`, but color objects are now always valid.)

7 color-database<%>

color-database<%> : interface?

The global the-color-database object is an instance of color-database<%>. It maintains a database of standard RGB colors for a predefined set of named colors (such as “black” and “light gray”).

The following colors are in the database:

Orange Red
OrangeRed
Tomato
DarkRed
Red
Firebrick
Crimson
DeepPink
Maroon
Indian Red
IndianRed
Medium Violet Red
MediumVioletRed
Violet Red
VioletRed
LightCoral
HotPink
PaleVioletRed
LightPink
RosyBrown
Pink
Orchid
LavenderBlush
Snow
Chocolate
SaddleBrown
Brown
DarkOrange
Coral
Sienna
Orange
Salmon
Peru
DarkGoldenrod
Goldenrod
SandyBrown

LightSalmon
DarkSalmon
Gold
Yellow
Olive
Burlywood
Tan
NavajoWhite
PeachPuff
Khaki
DarkKhaki
Moccasin
Wheat
Bisque
PaleGoldenrod
BlanchedAlmond
Medium Goldenrod
MediumGoldenrod
PapayaWhip
MistyRose
LemonChiffon
AntiqueWhite
Cornsilk
LightGoldenrodYellow
OldLace
Linen
LightYellow
SeaShell
Beige
FloralWhite
Ivory
Green
LawnGreen
Chartreuse
Green Yellow
GreenYellow
Yellow Green
YellowGreen
Medium Forest Green
OliveDrab
MediumForestGreen
Dark Olive Green
DarkOliveGreen
DarkSeaGreen
Lime
Dark Green

DarkGreen
Lime Green
LimeGreen
Forest Green
ForestGreen
Spring Green
SpringGreen
Medium Spring Green
MediumSpringGreen
Sea Green
SeaGreen
Medium Sea Green
MediumSeaGreen
Aquamarine
LightGreen
Pale Green
PaleGreen
Medium Aquamarine
MediumAquamarine
Turquoise
LightSeaGreen
Medium Turquoise
MediumTurquoise
Honeydew
MintCream
RoyalBlue
DodgerBlue
DeepSkyBlue
CornflowerBlue
Steel Blue
SteelBlue
LightSkyBlue
Dark Turquoise
DarkTurquoise
Cyan
Aqua
DarkCyan
Teal
Sky Blue
SkyBlue
Cadet Blue
CadetBlue
Dark Slate Gray
DarkSlateGray
LightSlateGray
SlateGray

Light Steel Blue
LightSteelBlue
Light Blue
LightBlue
PowderBlue
PaleTurquoise
LightCyan
AliceBlue
Azure
Medium Blue
MediumBlue
DarkBlue
Midnight Blue
MidnightBlue
Navy
Blue
Indigo
Blue Violet
BlueViolet
Medium Slate Blue
MediumSlateBlue
Slate Blue
SlateBlue
Purple
Dark Slate Blue
DarkSlateBlue
DarkViolet
Dark Orchid
DarkOrchid
MediumPurple
Cornflower Blue
Medium Orchid
MediumOrchid
Magenta
Fuchsia
DarkMagenta
Violet
Plum
Lavender
Thistle
GhostWhite
White
WhiteSmoke
Gainsboro
Light Gray
LightGray

Silver
Gray
Dark Gray
DarkGray
Dim Gray
DimGray
Black

The names are not case-sensitive.

See also color%.

```
(send a-color-database find-color color-name)
 → (or/c (is-a?/c color%) #f)
  color-name : string?
```

Finds a color by name (character case is ignored). If no color is found for the name, #f is returned, otherwise the result is an immutable color object.

```
(send a-color-database get-names) → (listof string?)
```

Returns an alphabetically sorted list of case-folded color names for which find-color returns a color% value.

8 dc<%>

```
dc<%> : interface?
```

A dc<%> object is a drawing context for drawing graphics and text. It represents output devices in a generic way; e.g., a canvas has a drawing context, as does a printer.

```
(send a-dc cache-font-metrics-key) → exact-integer?
```

Returns an integer that, if not 0, corresponds to a particular kind of device and scaling factor, such that text-extent information (from get-text-extent, get-char-height, etc.) is the same. The key is valid across all dc<%> instances, even among different classes.

A 0 result indicates that the current configuration of a-dc does not fit into a common category, and so no key is available for caching text-extent information.

```
(send a-dc clear) → void?
```

Clears the drawing region (fills it with the current background color, as determined by get-background). See also erase.

```
(send a-dc copy x y width height x2 y2) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  x2 : real?
  y2 : real?
```

Copies the rectangle defined by *x*, *y*, *width*, and *height* of the drawing context to the same drawing context at the position specified by *x2* and *y2*.

The result is undefined if the source and destination rectangles overlap.

```
(send a-dc draw-arc x
                    y
                    width
                    height
                    start-radians
                    end-radians)  → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  start-radians : real?
  end-radians : real?
```

Draws a counter-clockwise circular arc, a part of the ellipse inscribed in the rectangle specified by *x* (left), *y* (top), *width*, and *height*. The arc starts at the angle specified by *start-radians* (0 is three o'clock and half-pi is twelve o'clock) and continues counter-clockwise to *end-radians*. If *start-radians* and *end-radians* are the same, a full ellipse is drawn.

The current pen is used for the arc. If the current brush is not transparent, it is used to fill the wedge bounded by the arc plus lines (not drawn) extending to the center of the inscribed ellipse. If both the pen and brush are non-transparent, the wedge is filled with the brush before the arc is drawn with the pen.

The wedge and arc meet so that no space is left between them, but the precise overlap between the wedge and arc is platform- and size-specific. Typically, the regions drawn by the brush and pen overlap. In unsmoothed or aligned mode, the path for the outline is adjusted by shrinking the bounding ellipse width and height by, after scaling, one drawing unit divided by the alignment scale.

```
(send a-dc draw-bitmap  source
                        dest-x
                        dest-y
                       [style
                        color
                        mask]) → boolean?
  source : (is-a?/c bitmap%)
  dest-x : real?
  dest-y : real?
  style : (or/c 'solid 'opaque 'xor) = 'solid
  color : (is-a?/c color%)
        = (send the-color-database find-color "black")
  mask : (or/c (is-a?/c bitmap%) #f) = #f
```

Displays the *source* bitmap. The *dest-x* and *dest-y* arguments are in DC coordinates.

For color bitmaps, the drawing style and color arguments are ignored. For monochrome bitmaps, draw-bitmap uses the style and color arguments in the same way that a brush uses its style and color settings to draw a monochrome stipple (see brush% for more information).

If a *mask* bitmap is supplied, it must have the same width and height as *source*, and its ok? must return true, otherwise an exn:fail:contract exception is raised. The *source* bitmap and *mask* bitmap can be the same object, but if the drawing context is a bitmap-dc% object, both bitmaps must be distinct from the destination bitmap, otherwise an exn:fail:contract exception is raised.

The effect of *mask* on drawing depends on the type of the *mask* bitmap:

- If the *mask* bitmap is monochrome, drawing occurs in the target dc<%> only where

the mask bitmap contains black pixels (independent of *style*, which controls how the white pixels of a monochrome *source* are handled).

- If the *mask* bitmap is color with an alpha channel, its alpha channel is used as the mask for drawing *source*, and its color channels are ignored.
- If the *mask* bitmap is color without an alpha channel, the color components of a given pixel are averaged to arrive at an inverse alpha value for the pixel. In particular, if the *mask* bitmap is grayscale, then the blackness of each mask pixel controls the opacity of the drawn pixel (i.e., the mask acts as an inverted alpha channel).

The current brush, current pen, and current text for the DC have no effect on how the bitmap is drawn, but the bitmap is scaled if the DC has a scale, and the DC's alpha setting determines the opacity of the drawn pixels (in combination with an alpha channel of *source*, any given *mask*, and the alpha component of *color* when *source* is monochrome).

For post-script-dc% and pdf-dc% output, opacity from an alpha channel in *source*, from *mask*, or from *color* is rounded to full transparency or opacity.

The result is #t if the bitmap is successfully drawn, #f otherwise (possibly because the bitmap's ok? method returns #f).

See also draw-bitmap-section.

```
(send a-dc draw-bitmap-section  source
                                dest-x
                                dest-y
                                src-x
                                src-y
                                src-width
                                src-height
                               [style
                                color
                                mask])      → boolean?
  source : (is-a?/c bitmap%)
  dest-x : real?
  dest-y : real?
  src-x : real?
  src-y : real?
  src-width : (and/c real? (not/c negative?))
  src-height : (and/c real? (not/c negative?))
  style : (or/c 'solid 'opaque 'xor) = 'solid
  color : (is-a?/c color%)
        = (send the-color-database find-color "black")
  mask : (or/c (is-a?/c bitmap%) #f) = #f
```

Displays part of a bitmap.

The *src-x*, *src-y*, *src-width*, and *src-height* arguments specify a rectangle in the source bitmap to copy into this drawing context.

See draw-bitmap for information about *dest-x*, *dest-y*, *style*, *color*, and *mask*.

```
(send a-dc draw-ellipse x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Draws an ellipse contained in a rectangle with the given top-left corner and size. The current pen is used for the outline, and the current brush is used for filling the shape. If both the pen and brush are non-transparent, the ellipse is filled with the brush before the outline is drawn with the pen.

Brush filling and pen outline meet so that no space is left between them, but the precise overlap between the filling and outline is platform- and size-specific. Thus, the regions drawn by the brush and pen may partially overlap. In unsmoothed or aligned mode, the path for the outline is adjusted by, after scaling, shrinking the ellipse width and height by one drawing unit divided by the alignment scale.

```
(send a-dc draw-line x1 y1 x2 y2) → void?
  x1 : real?
  y1 : real?
  x2 : real?
  y2 : real?
```

Draws a line from one point to another. The current pen is used for drawing the line.

In unsmoothed mode, the points correspond to pixels, and the line covers both the start and end points. For a pen whose scaled width is larger than 1, the line is drawn centered over the start and end points.

See also set-smoothing for information on the 'aligned smoothing mode.

```
(send a-dc draw-lines  points
                      [xoffset
                       yoffset]) → void?
  points : (or/c (listof (is-a?/c point%))
                 (listof (cons/c real? real?)))
  xoffset : real? = 0
  yoffset : real? = 0
```

Draws lines using a list *points* of points, adding *xoffset* and *yoffset* to each point. A pair is treated as a point where the car of the pair is the x-value and the cdr is the y-value. The current pen is used for drawing the lines.

See also set-smoothing for information on the 'aligned smoothing mode.

```
(send a-dc draw-path  path
                     [xoffset
                      yoffset
                      fill-style]) → void?
  path : (is-a?/c dc-path%)
  xoffset : real? = 0
  yoffset : real? = 0
  fill-style : (or/c 'odd-even 'winding) = 'odd-even
```

Draws the sub-paths of the given dc-path% object, adding *xoffset* and *yoffset* to each point. (See dc-path% for general information on paths and sub-paths.) The current pen is used for drawing the path as a line, and the current brush is used for filling the area bounded by the path.

If both the pen and brush are non-transparent, the path is filled with the brush before the outline is drawn with the pen. The filling and outline meet so that no space is left between them, but the precise overlap between the filling and outline is platform- and size-specific. Thus, the regions drawn by the brush and pen may overlap. More generally, the pen is centered over the path, rounding left and down in unsmoothed mode.

The *fill-style* argument specifies the fill rule: 'odd-even or 'winding. In 'odd-even mode, a point is considered enclosed within the path if it is enclosed by an odd number of sub-path loops. In 'winding mode, a point is considered enclosed within the path if it is enclosed by more or less clockwise sub-path loops than counter-clockwise sub-path loops.

See also set-smoothing for information on the 'aligned smoothing mode.

```
(send a-dc draw-point x y) → void?
  x : real?
  y : real?
```

Plots a single point using the current pen.

```
(send a-dc draw-polygon  points
                        [xoffset
                         yoffset
                         fill-style]) → void?
  points : (or/c (listof (is-a?/c point%))
                 (listof (cons/c real? real?)))
  xoffset : real? = 0
  yoffset : real? = 0
  fill-style : (or/c 'odd-even 'winding) = 'odd-even
```

Draw a filled polygon using a list *points* of points, adding *xoffset* and *yoffset* to each point. A pair is treated as a point where the car of the pair is the x-value and the cdr is the

y-value. The polygon is automatically closed, so the first and last point can be different. The current pen is used for drawing the outline, and the current brush for filling the shape.

If both the pen and brush are non-transparent, the polygon is filled with the brush before the outline is drawn with the pen. The filling and outline meet so that no space is left between them, but the precise overlap between the filling and outline is platform- and shape-specific. Thus, the regions drawn by the brush and pen may overlap. More generally, the pen is centered over the polygon lines, rounding left and down in unsmoothed mode.

The *fill-style* argument specifies the fill rule: 'odd-even or 'winding. In 'odd-even mode, a point is considered enclosed within the polygon if it is enclosed by an odd number of loops. In 'winding mode, a point is considered enclosed within the polygon if it is enclosed by more or less clockwise loops than counter-clockwise loops.

See also set-smoothing for information on the 'aligned smoothing mode.

```
(send a-dc draw-rectangle x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Draws a rectangle with the given top-left corner and size. The current pen is used for the outline and the current brush for filling the shape. If both the pen and brush are non-transparent, the rectangle is filled with the brush before the outline is drawn with the pen.

In unsmoothed or aligned mode, when the pen is size 0 or 1, the filling precisely overlaps the entire outline. More generally, in unsmoothed or aligned mode, the path for the outline is adjusted by shrinking the rectangle width and height by, after scaling, one drawing unit divided by the alignment scale.

See also set-smoothing for information on the 'aligned smoothing mode.

```
(send a-dc draw-rounded-rectangle  x
                                   y
                                   width
                                   height
                                  [radius]) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  radius : real? = -0.25
```

Draws a rectangle with the given top-left corner, and with the given size. The corners are quarter-circles using the given radius. The current pen is used for the outline and the current

brush for filling the shape. If both the pen and brush are non-transparent, the rectangle is filled with the brush before the outline is drawn with the pen.

If *radius* is positive, the value is used as the radius of the rounded corner. If *radius* is negative, the absolute value is used as the *proportion* of the smallest dimension of the rectangle.

If *radius* is less than -0.5 or more than half of *width* or *height*, an exn:fail:contract exception is raised.

Brush filling and pen outline meet so that no space is left between them, but the precise overlap between the filling and outline is platform- and size-specific. Thus, the regions drawn by the brush and pen may partially overlap. In unsmoothed or aligned mode, the path for the outline is adjusted by, after scaling, shrinking the rectangle width and height by one drawing unit divided by the alignment scale.

See also set-smoothing for information on the 'aligned smoothing mode.

```
(send a-dc draw-spline x1 y1 x2 y2 x3 y3) → void?
  x1 : real?
  y1 : real?
  x2 : real?
  y2 : real?
  x3 : real?
  y3 : real?
```

Draws a spline from (*x1*, *y1*) to (*x3*, *y3*) using (*x2*, *y2*) as the control point.

See also set-smoothing for information on the 'aligned smoothing mode. See also dc-path% and draw-path for drawing more complex curves.

```
(send a-dc draw-text  text
                      x
                      y
                     [combine?
                      offset
                      angle])  → void?
  text : string?
  x : real?
  y : real?
  combine? : any/c = #f
  offset : exact-nonnegative-integer? = 0
  angle : real? = 0
```

Draws a text string at a specified point, using the current text font, and the current text foreground and background colors. For unrotated text, the specified point is used as the

starting top-left point for drawing characters (e.g, if “W” is drawn, the point is roughly the location of the top-left pixel in the “W”). Rotated text is rotated around this point.

The *text* string is drawn starting from the *offset* character, and continuing until the end of *text* or the first null character.

If *combine?* is #t, then *text* may be measured with adjacent characters combined to ligature glyphs, with Unicode combining characters as a single glyph, with kerning, with right-to-left rendering of characters, etc. If *combine?* is #f, then the result is the same as if each character is measured separately, and Unicode control characters are ignored.

The string is rotated by *angle* radians counter-clockwise. If *angle* is not zero, then the text is always drawn in transparent mode (see set-text-mode).

The current brush and current pen settings for the DC have no effect on how the text is drawn.

See get-text-extent for information on the size of the drawn text.

See also set-text-foreground, set-text-background, and set-text-mode.

(send a-dc end-doc) → void?

Ends a document, relevant only when drawing to a printer, PostScript, PDF, or SVG device.

For relevant devices, an exception is raised if end-doc is called when the document is not started with start-doc, when a page is currently started by start-page and not ended with end-page, or when the document has been ended already.

(send a-dc end-page) → void?

Ends a single page, relevant only when drawing to a printer, PostScript, PDF, or SVG device.

For relevant devices, an exception is raised if end-page is called when a page is not currently started by start-page.

(send a-dc erase) → void?

For a drawing context that has an alpha channel, erase sets all alphas to zero. Similarly, for a transparent canvas, erase erases all drawing to allow the background window to show through. For other drawing contexts that have no alpha channel or transparency, erase fills the drawing context with white.

(send a-dc flush) → void?

Calls the flush in canvas<%> method for canvas<%> output, and has no effect for other kinds of drawing contexts.

```
(send a-dc get-alpha) → (real-in 0 1)
```

Gets the current opacity for drawing; see set-alpha.

```
(send a-dc get-background) → (is-a?/c color%)
```

Gets the color used for painting the background. See also set-background.

```
(send a-dc get-brush) → (is-a?/c brush%)
```

Gets the current brush. See also set-brush.

```
(send a-dc get-char-height) → (and/c real? (not/c negative?))
```

Gets the height of a character using the current font.

Unlike most methods, this method can be called for a bitmap-dc% object without a bitmap installed.

```
(send a-dc get-char-width) → (and/c real? (not/c negative?))
```

Gets the average width of a character using the current font.

Unlike most methods, this method can be called for a bitmap-dc% object without a bitmap installed.

```
(send a-dc get-clipping-region) → (or/c (is-a?/c region%) #f)
```

Gets the current clipping region, returning #f if the drawing context is not clipped (i.e., the clipping region is the entire drawing region).

```
(send a-dc get-device-scale) → (and/c real? (not/c negative?))
                               (and/c real? (not/c negative?))
```

Gets an "external" scaling factor for drawing coordinates to the target device. For most DCs, the result is 1.0 and 1.0.

A post-script-dc% or pdf-dc% object returns scaling factors determined via get-scaling in ps-setup% at the time that the DC was created. A printer-dc% may also have a user-configured scaling factor.

```
(send a-dc get-font) → (is-a?/c font%)
```

Gets the current font. See also set-font.

```
(send a-dc get-gl-context) → (or/c (is-a?/c gl-context<%>) #f)
```

Returns a gl-context<%> object for this drawing context if it supports OpenGL, #f otherwise.

See gl-context<%> for more information.

```
(send a-dc get-initial-matrix)
 → (vector/c real? real? real? real? real? real?)
```

Returns a transformation matrix that converts logical coordinates to device coordinates. The matrix applies before additional origin offset, scaling, and rotation.

The vector content corresponds to a transformation matrix in the following order:

- *xx*: a scale from the logical *x* to the device *x*
- *xy*: a scale from the logical *x* added to the device *y*
- *yx*: a scale from the logical *y* added to the device *x*
- *yy*: a scale from the logical *y* to the device *y*
- *x0*: an additional amount added to the device *x*
- *y0*: an additional amount added to the device *y*

See also set-initial-matrix and get-transformation.

```
(send a-dc get-origin) → real? real?
```

Returns the device origin, i.e., the location in device coordinates of (0,0) in logical coordinates. The origin offset applies after the initial transformation matrix, but before scaling and rotation.

See also set-origin and get-transformation.

```
(send a-dc get-pen) → (is-a?/c pen%)
```

Gets the current pen. See also set-pen.

```
(send a-dc get-path-bounding-box path type)
 → real? real? real? real?
  path : (is-a?/c dc-path%)
  type : (or/c 'path 'stroke 'fill)
```

Returns a rectangle that encloses the path's points. The return values are the left, top, width, and, height of the rectangle. The numbers are in logical coordinates.

For the type `'stroke` the rectangle covers the area that would be affected ("inked") when drawn with the current pen by draw-path in the drawing context (with a transparent brush). If the pen width is zero, then an empty rectangle will be returned. The size and clipping of the drawing context is ignored.

For the type `'fill` the rectangle covers the area that would be affected ("inked") by draw-path in the drawing context (with a non-transparent pen and brush). If the line width is zero, then an empty rectangle will be returned. The size and clipping of the drawing context are ignored.

For the type `'path` the rectangle covers the path, but the pen and brush are ignored. The size and clipping of the drawing context are also ignored. More precisely: The result is defined as the limit of the bounding boxes returned by the `'stroke` type for line widths approaching 0 with a round pen cap. The "limit process" stops when an empty rectangle is returned. This implies that zero-area segments contributes to the rectangle.

For all types if the path is empty, then an empty rectangle `(values 0 0 0 0)` will be returned.

```
(send a-dc get-rotation) → real?
```

Returns the rotation of logical coordinates in radians to device coordinates. Rotation applies after the initial transformation matrix, origin offset, and scaling.

See also `set-rotation` and `get-transformation`.

```
(send a-dc get-scale) → real? real?
```

Returns the scaling factor that maps logical coordinates to device coordinates. Scaling applies after the initial transformation matrix and origin offset, but before rotation.

See also `set-scale` and `get-transformation`.

```
(send a-dc get-size) → (and/c real? (not/c negative?))
                       (and/c real? (not/c negative?))
```

Gets the size of the destination drawing area. For a `dc<%>` object obtained from a `canvas<%>`, this is the (virtual client) size of the destination window; for a `bitmap-dc%` object, this is the size of the selected bitmap (or 0 if no bitmap is selected); for a `post-script-dc%` or `printer-dc%` drawing context, this gets the horizontal and vertical size of the drawing area.

```
(send a-dc get-smoothing)
 → (or/c 'unsmoothed 'smoothed 'aligned)
```

Returns the current smoothing mode. See set-smoothing.

```
(send a-dc get-text-background) → (is-a?/c color%)
```

Gets the current text background color. See also set-text-background.

```
(send a-dc get-text-extent  string
                           [font
                            combine?
                            offset])
 → (and/c real? (not/c negative?))
   (and/c real? (not/c negative?))
   (and/c real? (not/c negative?))
   (and/c real? (not/c negative?))
  string : string?
  font : (or/c (is-a?/c font%) #f) = #f
  combine? : any/c = #f
  offset : exact-nonnegative-integer? = 0
```

Returns the size of str at it would be drawn in the drawing context, starting from the *offset* character of str, and continuing until the end of str or the first null character. The *font* argument specifies the font to use in measuring the text; if it is #f, the current font of the drawing area is used. (See also set-font.)

The result is four real numbers:

- the total width of the text (depends on both the font and the text);
- the total height of the font (depends only on the font);
- the distance from the baseline of the font to the bottom of the descender (included in the height, depends only on the font); and
- extra vertical space added to the font by the font designer (included in the height, and often zero; depends only on the font).

The returned width and height define a rectangle is that guaranteed to contain the text string when it is drawn, but the fit is not necessarily tight. Some undefined number of pixels on the left, right, top, and bottom of the drawn string may be "whitespace," depending on the whims of the font designer and the platform-specific font-scaling mechanism.

If *combine?* is #t, then text may be drawn with adjacent characters combined to ligature glyphs, with Unicode combining characters as a single glyph, with kerning, with right-to-left ordering of characters, etc. If *combine?* is #f, then the result is the same as if each character is drawn separately, and Unicode control characters are ignored.

Unlike most methods, this method can be called for a bitmap-dc% object without a bitmap installed.

```
(send a-dc get-text-foreground) → (is-a?/c color%)
```

Gets the current text foreground color. See also set-text-foreground.

```
(send a-dc get-text-mode) → (or/c 'solid 'transparent)
```

Reports how text is drawn; see set-text-mode.

```
(send a-dc get-transformation)
 → (vector/c (vector/c real? real? real? real? real? real?)
             real? real? real? real? real?)
```

Returns the current transformation setting of the drawing context in a form that is suitable for restoration via set-transformation.

The vector content is as follows:

- the initial transformation matrix; see get-initial-matrix;
- the X and Y origin; see get-origin;
- the X and Y scale; see get-origin;
- a rotation; see get-rotation.

```
(send a-dc glyph-exists? c) → boolean?
  c : char?
```

Returns #t if the given character has a corresponding glyph for this drawing context, #f otherwise.

Due to automatic font substitution when drawing or measuring text, the result of this method does not depend on the given font, which merely provides a hint for the glyph search. If the font is #f, the drawing context's current font is used. The result depends on the type of the drawing context, but the result for canvas% dc<%> instances and bitmap-dc% instances is always the same for a given platform and a given set of installed fonts.

See also screen-glyph-exists? .

```
(send a-dc ok?) → boolean?
```

Returns #t if the drawing context is usable.

```
(send a-dc resume-flush) → void?
```

Calls the resume-flush in canvas<%> method for canvas<%> output, and has no effect for other kinds of drawing contexts.

```
(send a-dc rotate angle) → void?
  angle : real?
```

Adds a rotation of *angle* radians to the drawing context's current transformation.

Afterward, the drawing context's transformation is represented in the initial transformation matrix, and the separate origin, scale, and rotation settings have their identity values.

```
(send a-dc scale x-scale y-scale) → void?
  x-scale : real?
  y-scale : real?
```

Adds a scaling of *x-scale* in the X-direction and *y-scale* in the Y-direction to the drawing context's current transformation.

Afterward, the drawing context's transformation is represented in the initial transformation matrix, and the separate origin, scale, and rotation settings have their identity values.

```
(send a-dc set-alignment-scale scale) → void?
  scale : (>/c 0.0)
```

Sets the drawing context's *alignment scale*, which determines how drawing coordinates and pen widths are adjusted for unsmoothed or aligned drawing (see set-smoothing).

The default alignment scale is 1.0, which means that drawing coorinates and pen sizes are aligned to integer values.

An alignment scale of 2.0 aligns drawing coordinates to half-integer values. A value of 2.0 could be suitable for a bitmap-dc% whose destination is a bitmap with a backing scale of 2.0, since half-integer values coorespond to pixel boundaries. Even when a destinate context has a backing scale of 2.0, however, an alignment scale of 1.0 may be desirable to maintain consistency with drawing contexts that have a backing scale and alignment scale of 1.0.

Added in version 1.1 of package draw-lib.

```
(send a-dc set-alpha opacity) → void?
  opacity : (real-in 0 1)
```

Determines the opacity of drawing. A value of 0.0 corresponds to completely transparent (i.e., invisible) drawing, and 1.0 corresponds to completely opaque drawing. For intermediate values, drawing is blended with the existing content of the drawing context. A color (e.g. for a brush) also has an alpha value; it is combined with the drawing context's alpha by multiplying.

```
(send a-dc set-background color) → void?
  color : (is-a?/c color%)
(send a-dc set-background color-name) → void?
  color-name : string?
```

Sets the background color for drawing in this object (e.g., using `clear` or using a stippled `brush%` with the mode `'opaque`). For monochrome drawing, all non-black colors are treated as white.

```
(send a-dc set-brush brush) → void?
  brush : (is-a?/c brush%)
(send a-dc set-brush color style) → void?
  color : (is-a?/c color%)
  style : (or/c 'transparent 'solid 'opaque
                'xor 'hilite 'panel
                'bdiagonal-hatch 'crossdiag-hatch
                'fdiagonal-hatch 'cross-hatch
                'horizontal-hatch 'vertical-hatch)
(send a-dc set-brush color-name style) → void?
  color-name : string?
  style : (or/c 'transparent 'solid 'opaque
                'xor 'hilite 'panel
                'bdiagonal-hatch 'crossdiag-hatch
                'fdiagonal-hatch 'cross-hatch
                'horizontal-hatch 'vertical-hatch)
```

Sets the current brush for drawing in this object. While a brush is selected into a drawing context, it cannot be modified. When a color and style are given, the arguments are as for `find-or-create-brush` in `brush-list%`.

```
(send a-dc set-clipping-rect x
                             y
                             width
                             height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Sets the clipping region to a rectangular region.

See also `set-clipping-region` and `get-clipping-region`.

```
(send a-dc set-clipping-region rgn) → void?
  rgn : (or/c (is-a?/c region%) #f)
```

Sets the clipping region for the drawing area, turning off all clipping within the drawing region if `#f` is provided.

The clipping region must be reset after changing a `dc<%>` object's origin or scale (unless it is `#f`); see `region%` for more information.

See also `set-clipping-rect` and `get-clipping-region`.

```
(send a-dc set-font font) → void?
  font : (is-a?/c font%)
```

Sets the current font for drawing text in this object.

```
(send a-dc set-initial-matrix m) → void?
  m : (vector/c real? real? real? real? real? real?)
```

Set a transformation matrix that converts logical coordinates to device coordinates. The matrix applies before additional origin offset, scaling, and rotation.

See `get-initial-matrix` for information on the matrix as represented by a vector *m*.

See also `transform`, which adds a transformation to the current transformation, instead of changing the transformation composition in the middle.

```
(send a-dc set-origin x y) → void?
  x : real?
  y : real?
```

Sets the device origin, i.e., the location in device coordinates of (0,0) in logical coordinates. The origin offset applies after the initial transformation matrix, but before scaling and rotation.

See also `translate`, which adds a translation to the current transformation, instead of changing the transformation composition in the middle.

```
(send a-dc set-pen pen) → void?
  pen : (is-a?/c pen%)
(send a-dc set-pen color width style) → void?
  color : (is-a?/c color%)
  width : (real-in 0 255)
  style : (or/c 'transparent 'solid 'xor 'hilite
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash)
(send a-dc set-pen color-name width style) → void?
  color-name : string?
```

```
  width : (real-in 0 255)
  style : (or/c 'transparent 'solid 'xor 'hilite
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash)
```

Sets the current pen for this object. When a color, width, and style are given, the arguments are as for `find-or-create-pen` in `pen-list%`.

The current pen does not affect text drawing; see also `set-text-foreground`.

While a pen is selected into a drawing context, it cannot be modified.

```
(send a-dc set-rotation angle) → void?
  angle : real?
```

Set the rotation of logical coordinates in radians to device coordinates. Rotation applies after the initial transformation matrix, origin offset, and scaling.

See also `rotate`, which adds a rotation to the current transformation, instead of changing the transformation composition.

```
(send a-dc set-scale x-scale y-scale) → void?
  x-scale : real?
  y-scale : real?
```

Sets a scaling factor that maps logical coordinates to device coordinates. Scaling applies after the initial transformation matrix and origin offset, but before rotation. Negative scaling factors have the effect of flipping.

See also `scale`, which adds a scale to the current transformation, instead of changing the transformation composition in the middle.

```
(send a-dc set-smoothing mode) → void?
  mode : (or/c 'unsmoothed 'smoothed 'aligned)
```

Enables or disables anti-aliased smoothing for drawing. (Text smoothing is not affected by this method, and is instead controlled through the `font%` object.)

The smoothing mode is either `'unsmoothed`, `'smoothed`, or `'aligned`. Both `'aligned` and `'smoothed` are smoothing modes that enable anti-aliasing, while both `'unsmoothed` and `'aligned` adjust drawing coordinates to match pixel boundaries. For most applications that draw to the screen or bitmaps, `'aligned` mode is the best choice.

Conceptually, integer drawing coordinates correspond to the boundary between pixels, and pen-based drawing is centered over a given line or curve. Thus, drawing with pen width `1` from (0, 10) to (10, 10) in `'smoothed` mode draws a 2-pixel wide line with 50% opacity.

In 'unsmoothed and 'aligned modes, drawing coordinates are truncated based on the alignment scale of the drawing context. Specifically, when the alignment scale is 1.0, drawing coordinates are truncated to integer coordinates. More generally, drawing coordinates are shifted toward zero so that the result multipled by the alignment scale is integral. For line drawing, coordinates are further shifted based on the pen width and the alignment scale, where the shift corrsponds to half of the pen width (reduced to a value such that its multiplication times the alignment scale times two produces an integer). In addition, for pen drawing through draw-rectangle, draw-ellipse, draw-rounded-rectangle, and draw-arc, the given width and height are each decreased by 1.0 divided by the alignment scale.

```
(send a-dc set-text-background color) → void?
  color : (is-a?/c color%)
(send a-dc set-text-background color-name) → void?
  color-name : string?
```

Sets the current text background color for this object. The text background color is painted behind text that is drawn with draw-text, but only for the 'solid text mode (see set-text-mode).

For monochrome drawing, all non-white colors are treated as black.

```
(send a-dc set-text-foreground color) → void?
  color : (is-a?/c color%)
(send a-dc set-text-foreground color-name) → void?
  color-name : string?
```

Sets the current text foreground color for this object, used for drawing text with draw-text.

For monochrome drawing, all non-black colors are treated as white.

```
(send a-dc set-text-mode mode) → void?
  mode : (or/c 'solid 'transparent)
```

Determines how text is drawn:

- 'solid — Before text is drawn, the destination area is filled with the text background color (see set-text-background).
- 'transparent — Text is drawn directly over any existing image in the destination, as if overlaying text written on transparent film.

```
(send a-dc set-transformation t) → void?
  t : (vector/c (vector/c real? real? real? real? real? real?)
                real? real? real? real? real?)
```

Sets the draw context's transformation. See get-transformation for information about *t*.

```
(send a-dc start-doc message) → void?
  message : string?
```

Starts a document, relevant only when drawing to a printer, PostScript, PDF, or SVG device. For some platforms, the *message* string is displayed in a dialog until end-doc is called.

For relevant devices, an exception is raised if start-doc has been called already (even if end-doc has been called as well). Furthermore, drawing methods raise an exception if not called while a page is active as determined by start-doc and start-page.

```
(send a-dc start-page) → void?
```

Starts a page, relevant only when drawing to a printer, PostScript, SVG, or PDF device.

Relevant devices, an exception is raised if start-page is called when a page is already started, or when start-doc has not been called, or when end-doc has been called already. In addition, in the case of PostScript output, Encapsulated PostScript (EPS) cannot contain multiple pages, so calling start-page a second time for a post-script-dc% instance raises an exception; to create PostScript output with multiple pages, supply #f as the as-eps initialization argument for post-script-dc%.

```
(send a-dc suspend-flush) → void?
```

Calls the suspend-flush in canvas<%> method for canvas<%> output, and has no effect for other kinds of drawing contexts.

```
(send a-dc transform m) → void?
  m : (vector/c real? real? real? real? real? real?)
```

Adds a transformation by *m* to the drawing context's current transformation.

See get-initial-matrix for information on the matrix as represented by a vector *m*.

Afterward, the drawing context's transformation is represented in the initial transformation matrix, and the separate origin, scale, and rotation settings have their identity values.

```
(send a-dc translate dx dy) → void?
  dx : real?
  dy : real?
```

Adds a translation of *dx* in the X-direction and *dy* in the Y-direction to the drawing context's current transformation.

Afterward, the drawing context's transformation is represented in the initial transformation matrix, and the separate origin, scale, and rotation settings have their identity values.

```
(send a-dc try-color try result) → void?
  try : (is-a?/c color%)
  result : (is-a?/c color%)
```

Determines the actual color used for drawing requests with the given color. The *result* color is set to the RGB values that are actually produced for this drawing context to draw the color *try*.

9 dc-path%

```
dc-path% : class?
  superclass: object%
```

A path is a set of figures defined by curves. A path can be used with the `draw-path` method of a `dc<%>` object to draw the path's curves as lines, fill the region bounded by the path's curves, or both. A path can also be used with the `set-path` method of a `region%` object to generate a region bounded by the path's curves.

A path consists of zero or more *closed sub-paths*, and possibly one *open sub-path*. Some `dc-path%` methods extend the open sub-path, some `dc-path%` methods close the open sub-path, and some `dc-path%` methods add closed sub-paths. This approach to drawing formulation is inherited from PostScript [Adobe99].

When a path is drawn as a line, a closed sub-path is drawn as a closed figure, analogous to a polygon. An open sub-path is drawn with disjoint start and end points, analogous lines drawn with `draw-lines` in `dc<%>`.

When a path is filled or used as a region, the open sub-path (if any) is treated as if it were closed. The content of a path is determined either through the `'even-odd` rule or the `'winding` rule, as selected at the time when the path is filled or used to generate a region.

A path is not connected to any particular `dc<%>` object, so setting a `dc<%>` origin or scale does not affect path operations. Instead, a `dc<%>`'s origin and scale apply at the time that the path is drawn or used to set a region.

```
(new dc-path%) → (is-a?/c dc-path%)
```

Creates a new path that contains no sub-paths (and no open sub-path).

```
(send a-dc-path append path) → void?
  path : (is-a?/c dc-path%)
```

Adds the sub-paths of *path* to *a-dc-path*. Closed sub-paths of *path* are added as closed sub-paths to *a-dc-path*. If both paths have an open sub-path, then this path's sub-path is extended by the given path's open sub-path, adding a line from this path's current ending point to the given path's starting point. If only one of the paths has an open sub-path, then it becomes (or remains) this path's open sub-path.

```
(send a-dc-path arc  x
                     y
                     width
                     height
                     start-radians
                     end-radians
                    [counter-clockwise?]) → void?
```

```
  x : real?
  y : real?
  width : real?
  height : real?
  start-radians : real?
  end-radians : real?
  counter-clockwise? : any/c = #t
```

Extends or starts the path's open sub-path with a curve that corresponds to a section of an ellipse. If *width* and *height* are non-negative, the ellipse is the one bounded by a rectangle whose top-left corner is (*x*, *y*) and whose dimensions are *width* by *height*; if *width* is negative, then the rectangle's right edge is *x*, and the ellipse width is (abs *width*), while a negative *height* similarly makes *y* is the bottom edge of the ellipse and the height (abs *height*). The ellipse section starts a the angle *start-radians* (0 is three o'clock and half-π is twelve o'clock) and continues to the angle *end-radians*; if *counter-clockwise?* is true, then the arc runs counter-clockwise from *start-radians* to *end-radians*, otherwise it runs clockwise.

Support for negative *width* and *height* helps avoid round-off problems for aligned drawing in an eventual destination, since `arc` reduces its input to a sequence of curves. In contrast, `draw-arc` in `dc<%>` can automatically correct for round off, since the drawing mode is known immediately.

If the path has no open sub-path, a new one is started with the arc's starting point. Otherwise, the arc extends the existing sub-path, and the existing path is connected with a line to the arc's starting point.

```
(send a-dc-path close) → void?
```

Closes the path's open sub-path. If the path has no open sub-path, an `exn:fail:contract` exception is raised.

```
(send a-dc-path curve-to x1 y1 x2 y2 x3 y3) → void?
  x1 : real?
  y1 : real?
  x2 : real?
  y2 : real?
  x3 : real?
  y3 : real?
```

Extends the path's open sub-path with a Bezier curve to the given point (*x3*,*y3*), using the points (*x1*, *y1*) and (*x2*, *y2*) as control points. If the path has no open sub-path, an `exn:fail:contract` exception is raised.

```
(send a-dc-path ellipse x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Closes the open sub-path, if any, and adds a closed sub-path that represents an ellipse bounded by a rectangle whose top-left corner is (*x*, *y*) and whose dimensions are *width* by *height*. (This convenience method is implemented in terms of `close` and `arc`.)

```
(send a-dc-path get-bounding-box) → real? real? real? real?
```

Returns a rectangle that encloses the path's points. The return values are the left, top, width, and height of the rectangle.

For curves within the path, the bounding box enclosed the two control points as well as the start and end points. Thus, the bounding box does not always tightly bound the path.

```
(send a-dc-path line-to x y) → void?
  x : real?
  y : real?
```

Extends the path's open sub-path with a line to the given point. If the path has no open sub-path, an `exn:fail:contract` exception is raised.

```
(send a-dc-path lines  points
                      [xoffset
                       yoffset]) → void?
  points : (or/c (listof (is-a?/c point%))
                 (listof (cons/c real? real?)))
  xoffset : real? = 0
  yoffset : real? = 0
```

Extends the path's open sub-path with a sequences of lines to the given points. A pair is treated as a point where the `car` of the pair is the x-value and the `cdr` is the y-value. If the path has no open sub-path, an `exn:fail:contract` exception is raised. (This convenience method is implemented in terms of `line-to`.)

```
(send a-dc-path move-to x y) → void?
  x : real?
  y : real?
```

After closing the open sub-path, if any, starts a new open sub-path with the given initial point.

```
(send a-dc-path open?) → boolean?
```

Returns `#t` if the path has an open sub-path, `#f` otherwise.

```
(send a-dc-path rectangle x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Closes the open sub-path, if any, and adds a closed path that represents a rectangle whose top-left corner is (*x*, *y*) and whose dimensions are *width* by *height*. (This convenience method is implemented in terms of `close`, `move-to`, and `line-to`.)

```
(send a-dc-path reset) → void?
```

Removes all sub-paths of the path.

```
(send a-dc-path reverse) → void?
```

Reverses the order of all points in all sub-paths. If the path has an open sub-path, the starting point becomes the ending point, and extensions to the open sub-path build on this new ending point. Reversing a closed sub-path affects how it combines with other sub-paths when determining the content of a path in `'winding` mode.

```
(send a-dc-path rotate radians) → void?
  radians : real?
```

Adjusts all points within the path (including all sub-paths), rotating them *radians* counterclockwise around (0, 0). Future additions to the path are not rotated by this call.

```
(send a-dc-path rounded-rectangle  x
                                   y
                                   width
                                   height
                                  [radius]) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  radius : real? = -0.25
```

Closes the open sub-path, if any, and adds a closed sub-path that represents a round-cornered rectangle whose top-left corner is (*x* *y*) and whose dimensions are *width* by *height*. (This convenience method is implemented in terms of `close`, `move-to`, `arc`, and `line-to`.)

If *radius* is positive, the value is used as the radius of the rounded corner. If *radius* is negative, the absolute value is used as the *proportion* of the smallest dimension of the rectangle.

If *radius* is less than `-0.5` or more than half of *width* or *height*, an `exn:fail:contract` exception is raised.

```
(send a-dc-path scale x y) → void?
  x : real?
  y : real?
```

Adjusts all points within the path (including all sub-paths), multiplying each x-coordinate by *x* and each y-coordinate by *y*. Scaling by a negative number flips the path over the corresponding axis. Future additions to the path are not scaled by this call.

```
(send a-dc-path text-outline  font
                              str
                              x
                              y
                             [combine?]) → void?
  font : (is-a?/c font%)
  str : string?
  x : real?
  y : real?
  combine? : any/c = #f
```

Closes the open sub-path, if any, and adds a closed sub-path to outline *str* using *font*. The top left of the text is positioned at *x* and *y*. The *combine?* argument enables kerning and character combinations as for draw-text in dc<%>.

```
(send a-dc-path transform m) → void?
  m : (vector/c real? real? real? real? real? real?)
```

Adjusts all points within the path (including all sub-paths) by applying the transformation represented by *m*.

See get-initial-matrix for information on the matrix as represented by a vector *m*.

```
(send a-dc-path translate x y) → void?
  x : real?
  y : real?
```

Adjusts all points within the path (including all sub-paths), shifting then *x* to the right and *y* down. Future additions to the path are not translated by this call.

10 font%

font% : class?
  superclass: object%

A *font* is an object which determines the appearance of text, primarily when drawing text to a device context. A font is determined by seven properties:

- size — The size of the text, either in points (the default) or logical drawing units, depending on the "size-in-pixels?" property (see below).
- family — A platform- and device-independent font designation. The families are:
  - 'default
  - 'decorative
  - 'roman
  - 'script
  - 'swiss
  - 'modern (fixed width)
  - 'symbol (Greek letters and more)
  - 'system (similar to the font to draw control labels, but see normal-control-font)
- face — A string face name, such as "Courier". The format and meaning of a face name is platform- and device-specific. If a font's face name is #f, then the font's appearance depends only on the family. If a face is provided but no mapping is available for the face name (for a specific platform or device), then the face name is ignored and the family is used. See font-name-directory<%> for information about how face names are mapped for drawing text.
- style — The slant style of the font, one of:
  - 'normal
  - 'slant (a.k.a "oblique")
  - 'italic
- weight — The weight of the font, one of:
  - 'normal
  - 'light
  - 'bold
- underline? — #t for underlined, #f for plain.

The terminology "family" and "face" is mangled relative to its usual meaning. A font% "face" is really used more like a font family in the usual terminology or more generally as a face-description string that is combined with other font% attributes to arrive at a face. A font% "family" is a kind of abstract font family that is mapped to a particular font family on a given platform.

- smoothing — Amount of anti-alias smoothing, one of:
  - `'default` (platform-specific, sometimes user-configurable)
  - `'partly-smoothed` (gray anti-aliasing)
  - `'smoothed` (sub-pixel anti-aliasing)
  - `'unsmoothed`
- size-in-pixels? — `#t` if the size of the font is in logical drawing units (i.e., pixels for an unscaled screen or bitmap drawing context), `#f` if the size of the font is in "points", where a "point" is equal to 1 pixel on Mac OS X and `(/ 96 72)` pixels on Windows and Unix
- hinting — Whether font metrics should be rounded to integers:
  - `'aligned` (the default) — rounds to integers to improve the consistency of letter spacing for pixel-based targets, but at the expense of making metrics unscalable
  - `'unaligned` — disables rounding

To avoid creating multiple fonts with the same characteristics, use the global `font-list%` object `the-font-list`.

See also `font-name-directory<%>`.

Changed in version 1.2 of package `draw-lib`: Defined "points" as `(/ 96 72)` pixels on Windows, independent of the screen resolution.

```
(make-object font%) → (is-a?/c font%)
(make-object font%  size
                    family
                   [style
                    weight
                    underline?
                    smoothing
                    size-in-pixels?
                    hinting])       → (is-a?/c font%)
  size : (real-in 0.0 1024.0)
  family : font-family/c
  style : font-style/c = 'normal
  weight : font-weight/c = 'normal
  underline? : any/c = #f
  smoothing : font-smoothing/c = 'default
  size-in-pixels? : any/c = #f
  hinting : font-hinting/c = 'aligned
```

```
(make-object font%  size
                    face
                    family
                   [style
                    weight
                    underline?
                    smoothing
                    size-in-pixels?
                    hinting])       → (is-a?/c font%)
  size : (real-in 0.0 1024.0)
  face : string?
  family : font-family/c
  style : font-style/c = 'normal
  weight : font-weight/c = 'normal
  underline? : any/c = #f
  smoothing : font-smoothing/c = 'default
  size-in-pixels? : any/c = #f
  hinting : font-hinting/c = 'aligned
```

When no arguments are provided, creates an instance of the default font. If no face name is provided, the font is created without a face name.

See font% for information about *family*, *style*, *weight*, *smoothing*, *size-in-pixels?*, and *hinting*. font-name-directory<%>.

See also make-font.

Changed in version 1.4 of package draw-lib: Changed *size* to allow non-integer and zero values.

```
(send a-font get-face) → (or/c string? #f)
```

Gets the font's face name, or #f if none is specified.

```
(send a-font get-family) → font-family/c
```

Gets the font's family. See font% for information about families.

```
(send a-font get-font-id) → exact-integer?
```

Gets the font's ID, for use with a font-name-directory<%>. The ID is determined by the font's face and family specifications, only.

```
(send a-font get-hinting) → font-hinting/c
```

Gets the font's hinting. See font% for information about hinting.

```
(send a-font get-point-size) → (integer-in 1 1024)
```

Gets the font's size rounded to the nearest non-zero integer. Despite the method's name, the result is in either logical units or points, depending on the result of `get-size-in-pixels`.

See `get-size`, instead. The `get-point-size` method is provided for backward compatibility.

```
(send a-font get-size) → (real-in 0.0 1024.0)
```

Gets the font's size (roughly the height). The size is in either logical units or points, depending on the result of `get-size-in-pixels`.

Due to space included in a font by a font designer, a font tends to generate text that is slightly taller than the nominal size.

Added in version 1.4 of package `draw-lib`.

```
(send a-font get-size-in-pixels) → boolean?
```

Returns `#t` if the size reported by `get-point-size` is in logical drawing units, `#f` if it is in points.

For a size in points and a screen or bitmap drawing context, the logical height depends on the resolution of the screen.

```
(send a-font get-smoothing) → font-smoothing/c
```

Gets the font's anti-alias smoothing mode. See `font%` for information about smoothing.

```
(send a-font get-style) → font-style/c
```

Gets the font's slant style. See `font%` for information about styles.

```
(send a-font get-underlined) → boolean?
```

Returns `#t` if the font is underlined or `#f` otherwise.

```
(send a-font get-weight) → font-weight/c
```

Gets the font's weight. See `font%` for information about weights.

```
(send a-font screen-glyph-exists? c
                                  [for-label?]) → boolean?
  c : char?
  for-label? : any/c = #f
```

Returns `#t` if the given character has a corresponding glyph when drawing to the screen or a bitmap, `#f` otherwise.

If the second argument is true, the result indicates whether the glyph is available for control labels. Otherwise, it indicates whether the glyph is available for `dc<%>` drawing.

For `dc<%>` drawing, due to automatic font substitution when drawing or measuring text, the result of this method does not depend on this font's attributes (size, face, etc.). The font's attributes merely provide a hint for the glyph search.

See also `glyph-exists?` .

11 font-list%

```
font-list% : class?
  superclass: object%
```

A font-list% object maintains a list of font% objects to avoid repeatedly creating fonts.

A global font list, the-font-list, is created automatically.

```
(new font-list%) → (is-a?/c font-list%)
```

Creates an empty font list.

```
(send a-font-list find-or-create-font  size
                                       family
                                       style
                                       weight
                                      [underline?
                                       smoothing
                                       size-in-pixels?
                                       hinting])
 → (is-a?/c font%)
  size : (real-in 0.0 1024.0)
  family : (or/c 'default 'decorative 'roman 'script
                 'swiss 'modern 'symbol 'system)
  style : (or/c 'normal 'italic 'slant)
  weight : (or/c 'normal 'bold 'light)
  underline? : any/c = #f
  smoothing : (or/c 'default 'partly-smoothed 'smoothed 'unsmoothed)
            = 'default
  size-in-pixels? : any/c = #f
  hinting : (or/c 'aligned 'unaligned) = 'aligned
(send a-font-list find-or-create-font  size
                                       face
                                       family
                                       style
                                       weight
                                      [underline
                                       smoothing
                                       size-in-pixels?
                                       hinting])
 → (is-a?/c font%)
  size : (real-in 0.0 1024.0)
  face : string?
  family : (or/c 'default 'decorative 'roman 'script
                 'swiss 'modern 'symbol 'system)
```

```
style : (or/c 'normal 'italic 'slant)
weight : (or/c 'normal 'bold 'light)
underline : any/c = #f
smoothing : (or/c 'default 'partly-smoothed 'smoothed 'unsmoothed)
          = 'default
size-in-pixels? : any/c = #f
hinting : (or/c 'aligned 'unaligned) = 'aligned
```

Finds an existing font in the list or creates a new one (that is automatically added to the list). The arguments are the same as for creating a `font%` instance.

Changed in version 1.4 of package `draw-lib`: Changed *size* to allow non-integer and zero values.

12 font-name-directory<%>

```
font-name-directory<%> : interface?
```

There is one font-name-directory<%> object: the-font-name-directory. It implements a mapping from font specifications (face, family, style, and weight) to information for rendering text on a specific device. Programmers rarely need to directly invoke methods of the-font-name-directory. It is used automatically when drawing text to a dc<%> object. Nevertheless, the-font-name-directory is available so that programmers can query or modify the mapping manually. A programmer may also need to understand how the face-and-family mapping works.

To extract mapping information from the-font-name-directory, first obtain a *font ID*, which is an index based on a family and optional face string. Font IDs are returned by find-or-create-font-id and get-font-id . A Font ID can be combined with a weight and style to obtain a specific mapping value via get-screen-name or get-post-script-name.

For a family without a face string, the corresponding font ID has a useful built-in mapping for every platform and device. For a family with a face string, the-font-name-directory interprets the string (in a platform-specific way) to generate a mapping for drawing (to a canvas's dc<%>, a bitmap-dc%, or a printer-dc%).

Currently, on all platforms, a face string is interpreted as a Pango font description when it contains a comma, otherwise it is treated as a family name. A face can thus be just a family name such as "Helvetica", a family followed by a comma and font modifiers as in "Helvetica, Bold", or a sequence of comma-separated familie names followed by space-separated font options as an "Helvetica, Arial, bold italic". Any size in a font description is overridden by a given font%'s size. Any (slant) style or weight options in a font description are overridden by a non-'normal value for a given font%'s style or weight, respectively.

```
(send a-font-name-directory find-family-default-font-id family)
 → exact-integer?
  family : (or/c 'default 'decorative 'roman 'script
                 'swiss 'modern 'symbol 'system)
```

Gets the font ID representing the default font for a family. See font% for information about font families.

```
(send a-font-name-directory find-or-create-font-id name
                                                    family)
 → exact-integer?
  name : string?
  family : (or/c 'default 'decorative 'roman 'script
                 'swiss 'modern 'symbol 'system)
```

Gets the face name for a font ID, initializing the mapping for the face name if necessary.

Font ID are useful only as mapping indices for the-font-name-directory.

```
(send a-font-name-directory get-face-name font-id)
 → (or/c string? #f)
  font-id : exact-integer?
```

Gets the face name for a font ID. If the font ID corresponds to the default font for a particular family, #f is returned.

```
(send a-font-name-directory get-family font-id)
 → (or/c 'default 'decorative 'roman 'script
         'swiss 'modern 'symbol 'system)
  font-id : exact-integer?
```

Gets the family for a font ID. See font% for information about font families.

```
(send a-font-name-directory get-font-id name
                                        family)
 → exact-integer?
  name : string?
  family : (or/c 'default 'decorative 'roman 'script
                 'swiss 'modern 'symbol 'system)
```

Gets the font ID for a face name paired with a default family. If the mapping for the given pair is not already initialized, 0 is returned. See also find-or-create-font-id.

Font ID are useful only as mapping indices for the-font-name-directory.

```
(send a-font-name-directory get-post-script-name font-id
                                                 weight
                                                 style)
 → (or/c string? #f)
  font-id : exact-integer?
  weight : (or/c 'normal 'bold 'light)
  style : (or/c 'normal 'italic 'slant)
```

Gets a PostScript font description for a font ID, weight, and style combination.

See font% for information about *weight* and *style*.

```
(send a-font-name-directory get-screen-name font-id
                                            weight
                                            style)
 → (or/c string? #f)
```

```
  font-id : exact-integer?
  weight : (or/c 'normal 'bold 'light)
  style : (or/c 'normal 'italic 'slant)
```

Gets a platform-dependent screen font description (used for drawing to a canvas's dc<%>, a bitmap-dc%, or a printer-dc%) for a font ID, weight, and style combination.

See font% for information about *weight* and *style*.

```
(send a-font-name-directory set-post-script-name font-id
                                                 weight
                                                 style
                                                 name)
 → void?
  font-id : exact-integer?
  weight : (or/c 'normal 'bold 'light)
  style : (or/c 'normal 'italic 'slant)
  name : string?
```

Sets a PostScript font description for a font ID, weight, and style combination. See also get-post-script-name.

See font% for information about *weight* and *style*.

```
(send a-font-name-directory set-screen-name font-id
                                            weight
                                            style
                                            name)    → void?
  font-id : exact-integer?
  weight : (or/c 'normal 'bold 'light)
  style : (or/c 'normal 'italic 'slant)
  name : string?
```

Sets a platform-dependent screen font description (used for drawing to a canvas's dc<%>, a bitmap-dc%, or a printer-dc%) for a font ID, weight, and style combination.

See font% for information about *weight* and *style*.

13 gl-config%

```
gl-config% : class?
  superclass: object%
```

A gl-config% object encapsulates configuration information for an OpenGL drawing context. Use a gl-config% object as an initialization argument for canvas% or provide it to make-gl-bitmap.

```
(new gl-config%) → (is-a?/c gl-config%)
```

Creates a GL configuration that indicates legacy OpenGL, double buffering, a depth buffer of size one, no stencil buffer, no accumulation buffer, no multisampling, and not stereo.

```
(send a-gl-config get-accum-size) → (integer-in 0 256)
```

Reports the accumulation-buffer size (for each of red, green, blue, and alpha) that the configuration requests, where zero means no accumulation buffer is requested.

```
(send a-gl-config get-depth-size) → (integer-in 0 256)
```

Reports the depth-buffer size that the configuration requests, where zero means no depth buffer is requested.

```
(send a-gl-config get-double-buffered) → boolean?
```

Reports whether the configuration requests double buffering or not.

```
(send a-gl-config get-multisample-size) → (integer-in 0 256)
```

Reports the multisampling size that the configuration requests, where zero means no multisampling is requested.

```
(send a-gl-config get-share-context)
 → (or/c #f (is-a?/c gl-context<%>))
```

Returns a gl-context<%> object that shares certain objects (textures, display lists, etc.) with newly created OpenGL drawing contexts, or #f is none is set.

See also set-share-context.

```
(send a-gl-config get-stencil-size) → (integer-in 0 256)
```

Reports the stencil-buffer size that the configuration requests, where zero means no stencil buffer is requested.

```
(send a-gl-config get-stereo) → boolean?
```

Reports whether the configuration requests stereo or not.

```
(send a-gl-config set-accum-size on?) → void?
  on? : (integer-in 0 256)
```

Adjusts the configuration to request a particular accumulation-buffer size for every channel (red, green, blue, and alpha), where zero means no accumulation buffer is requested.

```
(send a-gl-config set-depth-size on?) → void?
  on? : (integer-in 0 256)
```

Adjusts the configuration to request a particular depth-buffer size, where zero means no depth buffer is requested.

```
(send a-gl-config set-double-buffered on?) → void?
  on? : any/c
```

Adjusts the configuration to request double buffering or not.

```
(send a-gl-config set-multisample-size on?) → void?
  on? : (integer-in 0 256)
```

Adjusts the configuration to request a particular multisample size, where zero means no multisampling is requested. If a multisampling context is not available, this request will be ignored.

```
(send a-gl-config set-share-context context) → void?
  context : (or/c #f (is-a?/c gl-context<%>))
```

Determines a `gl-context<%>` object that shares certain objects (textures, display lists, etc.) with newly created OpenGL drawing contexts, where `#f` indicates that no sharing should occur.

When a context *B* shares objects with context *A*, it is also shares objects with every other context sharing with *A*, and vice versa.

If an OpenGL implementation does not support sharing, *context* is effectively ignored when a new context is created. Sharing should be supported in all versions of Mac OS X. On Windows and Linux, sharing is provided by the presence of the `WGL_ARB_create_context` and `GLX_ARB_create_context` extensions, respectively (and OpenGL 3.2 requires both).

```
(send a-gl-config set-stencil-size on?) → void?
  on? : (integer-in 0 256)
```

Adjusts the configuration to request a particular stencil-buffer size, where zero means no stencil buffer is requested.

```
(send a-gl-config set-stereo on?) → void?
  on? : any/c
```

Adjusts the configuration to request stereo or not.

```
(send a-gl-config get-legacy?) → boolean?
```

Determines whether to use legacy, "Compatibility" OpenGL or "Core" OpenGL. Core OpenGL profiles are currently supported on OS X (versions 10.7 and up) and Linux (if the graphics drivers support them).

Added in version 1.2 of package `draw-lib`.

```
(send a-gl-config set-legacy? legacy?) → void?
  legacy? : any/c
```

Adjusts the configuration to request legacy or not.

Added in version 1.2 of package `draw-lib`.

14 gl-context<%>

```
gl-context<%> : interface?
```

A gl-context<%> object represents a context for drawing with OpenGL to a specific dc<%> instance. To obtain a gl-context<%> object, call get-gl-context of the target drawing context.

Only canvas dc<%> and bitmap-dc% objects containing a bitmap from make-gl-bitmap support OpenGL (always on Windows and Mac OS X, sometimes on Unix). Normal dc<%> drawing and OpenGL drawing can be mixed in a bitmap-dc%, but a canvas that uses the 'gl style to support OpenGL does not reliably support normal dc<%> drawing; use a bitmap if you need to mix drawing modes, and use a canvas to maximize OpenGL performance.

When the target bitmap for a bitmap-dc% context is changed via set-bitmap, the associated gl-context<%> changes. Canvas contexts are normally double buffered, and bitmap contexts are single buffered.

The racket/gui/base library provides no OpenGL routines. Instead, they must be obtained from a separate library, such as sgl. The facilities in racket/gui/base merely manage the current OpenGL context, connecting it to windows and bitmaps.

Only one OpenGL context can be active at a time across all threads and eventspaces. OpenGL contexts are not protected against interference among threads; that is, if a thread selects one of its OpenGL contexts, then other threads can write into the context via OpenGL commands. However, if all threads issue OpenGL commands only within a thunk passed to call-as-current, then drawing from the separate threads will not interfere, because call-as-current uses a lock to serialize context selection across all threads in Racket.

```
(send a-gl-context call-as-current  thunk
                                    [alternate
                                     enable-breaks?]) → any/c
  thunk : (-> any)
  alternate : evt? = never-evt
  enable-breaks? : any/c = #f
```

Calls a thunk with this OpenGL context as the current context for OpenGL commands.

The method blocks to obtain a lock that protects the global OpenGL context, and it releases the lock when the thunk returns or escapes. The lock is re-entrant, so a nested use of the method in the same thread with the same OpenGL context does not obtain or release the lock.

The lock prevents interference among OpenGL-using threads. If a thread is terminated while holding the context lock, the lock is released. Continuation jumps into the thunk do not grab the lock or set the OpenGL context. See gl-context<%> for more information on interference.

The method accepts an alternate synchronizable event for use while blocking for the context lock; see also sync.

The result of the method call is the result of the thunk if it is called, or the result of the alternate event if it is chosen instead of the context lock.

If ok? returns #f at the time that this method is called, then an exn:fail:contract exception is raised.

If *enable-breaks?* is true, then the method uses sync/enable-break while blocking for the context-setting lock instead of sync.

(send *a-gl-context* get-handle) → cpointer?

Returns a handle to the platform's underlying context. The value that the pointer represents depends on the platform:

- Windows: HGLRC
- Mac OS X: NSOpenGLContext
- Unix: GLXContext

Note that these values are not necessary the most "low-level" context objects, but are instead the ones useful to Racket. For example, a NSOpenGLContext wraps a CGLContextObj.

(send *a-gl-context* ok?) → boolean?

Returns #t if this context is available OpenGL drawing, #f otherwise.

A context is unavailable if OpenGL support is disabled at compile time or run time, if the context is associated with a bitmap-dc% with no selected bitmap or with a monochrome selected bitmap, if the context is for a canvas that no longer exists, or if there was a low-level error when preparing the context.

(send *a-gl-context* swap-buffers) → void?

Swaps the front (visible) and back (OpenGL-drawing) buffer for a context associated with a canvas, and has no effect on a bitmap context.

This method implicitly uses call-as-current to obtain the context lock. Since the lock is re-entrant, however, the swap-buffers method can be safely used within a call-as-current thunk.

(get-current-gl-context) → gl-context<%>

If within the dynamic extent of a `call-as-current` method call, returns the current context; otherwise returns `#f`. This is possibly most useful for caching context-dependent state or data, such as extension strings. Create such caches using `make-weak-hasheq`.

Added in version 1.3 of package `draw-lib`.

15 linear-gradient%

```
linear-gradient% : class?
  superclass: object%
```

A *linear gradient* is used with a brush% to fill areas with smooth color transitions. Color transitions are based on a line, where colors are assigned to stop points along the line, and colors for in-between points are interpolated from the stop-point colors. The color of a point on the gradient's line is propagated to all points in the drawing context that are touched by a line through the point and perpendicular to the gradient's line.

```
(new linear-gradient%
   [x0 x0]
   [y0 y0]
   [x1 x1]
   [y1 y1]
   [stops stops])
→ (is-a?/c linear-gradient%)
 x0 : real?
 y0 : real?
 x1 : real?
 y1 : real?
 stops : (listof (list/c (real-in 0 1) (is-a?/c color%)))
```

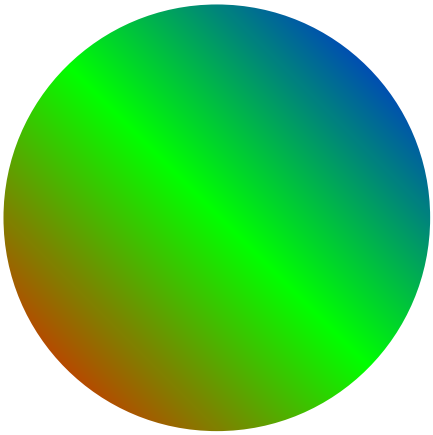
Creates a linear gradient with a line from (*x0*, *y0*) to end point (*x1*, *y1*). The *stops* list assigns colors to stop points along the line, where 0.0 corresponds to (*x0*, *y0*), 1.0 corresponds to (*x1*, y2), and numbers in between correspond to points in between.

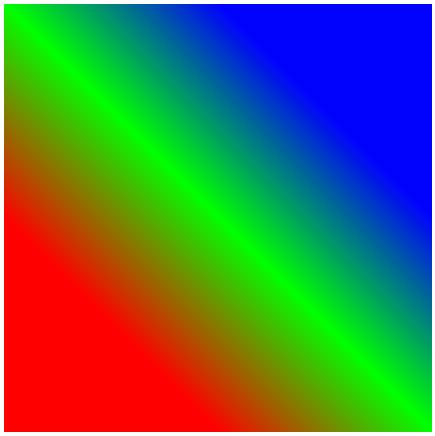
Elements in *stops* are implicitly sorted by point (i.e., by the number between 0.0 and 1.0). Order is preserved for multiple elements for the same point, in which case the first element for a given point is treated infinitesimally before the point, and additional elements between the first and last for a stop point are effectively ignored.

Examples:

```
> (define ellipse-brush
    (new brush%
         [gradient
          (new linear-gradient%
               [x0 0]
               [y0 200]
               [x1 200]
               [y1 0]
               [stops
                (list (list 0   (make-object color% 255 0 0))
                      (list 0.5 (make-object color% 0 255 0))
```

```
                    (list 1   (make-object color% 0 0 255))])]))

> (define rectangle-brush
    (new brush%
         [gradient
          (new linear-gradient%
               [x0 0]
               [y0 100]
               [x1 100]
               [y1 0]
               [stops
                (list (list 0   (make-object color% 255 0 0))
                      (list 0.5 (make-object color% 0 255 0))
                      (list 1   (make-object color% 0 0 255)))])]))

> (dc
   (λ (dc dx dy)
     (define old-pen (send dc get-pen))
     (define old-brush (send dc get-brush))
     (define-values (ox oy) (send dc get-origin))
     (send dc set-pen "black" 1 'transparent)
     (send dc set-brush ellipse-brush)

     (send dc set-origin (+ ox dx 50) (+ oy dy 50))
     (send dc draw-ellipse 0 0 200 200)

     (send dc set-brush rectangle-brush)
     (send dc set-origin (+ ox dx 300) (+ oy dy 50))
     (send dc draw-rectangle 0 0 200 200)

     (send dc set-pen old-pen)
     (send dc set-brush old-brush)
     (send dc set-origin ox oy))
   550 300)
```





```
(send a-linear-gradient get-line) → real? real? real? real?
```

Returns the gradient's control line as *x0*, *y0*, *x1*, and *y1*.

```
(send a-linear-gradient get-stops)
 → (listof (list/c (real-in/c 0 1) (is-a?/c color%)))
```

Returns the gradient's list of color stops.

16 pdf-dc%

```
pdf-dc% : class?
  superclass: object%
  extends: dc<%>
```

Like post-script-dc%, but generates a PDF file instead of a PostScript file.

```
(new pdf-dc%
   [[interactive interactive]
    [parent parent]
    [use-paper-bbox use-paper-bbox]
    [as-eps as-eps]
    [width width]
    [height height]
    [output output]])
→ (is-a?/c pdf-dc%)
 interactive : any/c = #t
 parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) #f) = #f
 use-paper-bbox : any/c = #f
 as-eps : any/c = #t
 width : (or/c (and/c real? (not/c negative?)) #f) = #f
 height : (or/c (and/c real? (not/c negative?)) #f) = #f
 output : (or/c path-string? output-port? #f) = #f
```

See post-script-dc% for information on the arguments. The *as-eps* argument is allowed for consistency with post-script-dc%, but its value is ignored.

17 pen%

```
pen% : class?
  superclass: object%
```

A pen is a drawing tool with a color, width, and style. A pen draws lines and outlines, such as the outline of a rectangle. In a monochrome destination, all non-white pens are drawn as black.

In addition to its color, width, and style, a pen can have a *pen stipple* bitmap. Drawing with a stipple pen is similar to calling `draw-bitmap` with the stipple bitmap in region painted by the pen.

A *pen style* is one of the following:

- `'transparent` — Draws with no effect (on the outline of the drawn shape).
- `'solid` — Draws using the pen's color. If a (monochrome) pen stipple is installed into the pen, black pixels from the stipple are transferred to the destination using the brush's color, and white pixels from the stipple are not transferred.
- `'xor` — The same as `'solid`, accepted only for partial backward compatibility.
- `'hilite` — Draws with black and a `0.3` alpha.
- The following special pen modes use the pen's color, and they only apply when a pen stipple is not used:
  - `'dot`
  - `'long-dash`
  - `'short-dash`
  - `'dot-dash`
  - `'xor-dot`
  - `'xor-long-dash`
  - `'xor-short-dash`
  - `'xor-dot-dash`

To avoid creating multiple pens with the same characteristics, use the global `pen-list%` object `the-pen-list`, or provide a color, width, and style to `set-pen` in `dc<%>`.

When drawing in `'unsmoothed` or `'aligned` mode, a pen's size is truncated after scaling to size that is integral after multiplication by the drawing context's alignment scale. A pen of size `0` (after truncation, if applicable) uses a non-zero, scale-insensitive line size for the destination drawing context: `1/4` unit (after scaling) for `post-script-dc%` or `pdf-dc%`

contexts in 'smoothed mode, or 1 unit (after scaling) divided by the alignment scale for any other context. For example, in unscaled canvas and bitmap contexts with an alignment scale of 1.0, a zero-width pen behaves the same as a pen of size 1.

See also make-pen.

```
(new pen%
   [[color color]
    [width width]
    [style style]
    [cap cap]
    [join join]
    [stipple stipple]]) → (is-a?/c pen%)
 color : (or/c string? (is-a?/c color%)) = "black"
 width : (real-in 0 255) = 0
 style : pen-style/c = 'solid
 cap : pen-cap-style/c = 'round
 join : pen-join-style/c = 'round
 stipple : (or/c #f (is-a?/c bitmap%)) = #f
```

Creates a pen with the given color, width, pen style, cap style, join style, and pen stipple bitmap. For the case that the color is specified using a name, see color-database<%> for information about color names; if the name is not known, the pen's color is black.

```
(send a-pen get-cap) → pen-cap-style/c
```

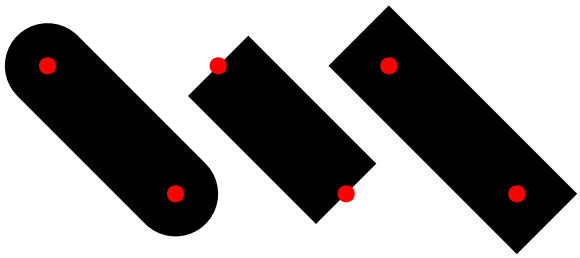
Returns the pen *cap style*, which determines the shape of a line at each of its ending points when drawn by draw-line or at the non-connecting ends of lines when drawn by draw-lines or draw-path. The default is 'round, which draws the end of a line as a semi-circle. The 'projecting style draws a square in place of the semi-circle (i.e., past the point at which the line stops). The 'butt style ends the line with a straight edge, instead of projecting past the ending point of the line.

This code draws three diagonal lines, one with each of the possible caps ('round, 'butt, and then 'projecting) and puts a little red dot on the end points of the line.

Examples:

```
> (define (plot-line dc x1 y1 x2 y2 cap)
    (send dc set-pen
          (send the-pen-list find-or-create-pen
                "black" 40 'solid cap))
    (send dc draw-line x1 y1 x2 y2)
    (send dc set-brush "red" 'solid)
    (send dc set-pen "black" 1 'transparent)
    (send dc draw-ellipse (- x1 4) (- y1 4) 8 8)
    (send dc draw-ellipse (- x2 4) (- y2 4) 8 8))
```

```
> (dc
   (λ (dc dx dy)
     (define old-pen (send dc get-pen))
     (define old-brush (send dc get-brush))

     (plot-line dc 20 30 80 90 'round)
     (plot-line dc 100 30 160 90 'butt)
     (plot-line dc 180 30 240 90 'projecting)

     (send dc set-pen old-pen)
     (send dc set-brush old-brush))
   270 120)
```



(send *a-pen* get-color) → (is-a?/c color%)

Returns the pen's color object.

(send *a-pen* get-join) → pen-join-style/c

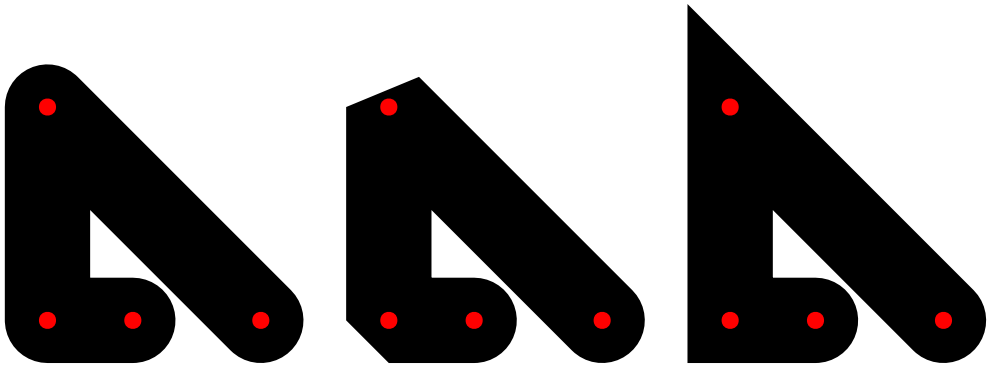
Returns the pen *join style* that is used between multiple lines connected through draw-lines, draw-rectangle, draw-polygon, or draw-path. The join style fills the space that would be left at the outside corner of two lines if they were draw separately with 'butt line endings. The default join style is 'round, which fills under an arc that lines up with the outside of each of the two lines. The 'bevel style fills in the gap without adding extra pixels (i.e., it makes a blunt corner). The 'miter style fills the gap by adding pixels that would be covered by both lines if they were extended past the corner (i.e., it makes a sharp corner).

This code shows the three join styles ('round, 'bevel and then 'miter) by drawing a sequence of lines, first with a sharp corner and then with a right-angle. Each of the end points of the lines i with a red dot.

Examples:

```
> (define points '((100 . 100)
                   (0 . 0)
                   (0 . 100)
                   (40 . 100)))
```

```
> (define (connect-points dc dx dy join)
    (send dc set-pen
          (send the-pen-list find-or-create-pen
                "black" 40 'solid 'round join))
    (send dc draw-lines points dx dy)
    (send dc set-brush "red" 'solid)
    (send dc set-pen "black" 1 'transparent)
    (for ([pt (in-list points)])
      (send dc draw-ellipse
            (+ dx (car pt) -4) (+ dy (cdr pt) -4)
            8 8)))

> (dc
   (λ (dc dx dy)
     (define old-pen (send dc get-pen))
     (define old-brush (send dc get-brush))

     (connect-points dc 20 50 'round)
     (connect-points dc 180 50 'bevel)
     (connect-points dc 340 50 'miter)

     (send dc set-pen old-pen)
     (send dc set-brush old-brush))
   460 170)
```



(send *a-pen* get-stipple) → (or/c (is-a?/c bitmap%) #f)

Gets the current pen stipple bitmap, or returns #f if no stipple bitmap is installed.

(send *a-pen* get-style) → pen-style/c

Returns the pen style. See pen% for information about possible styles.

(send *a-pen* get-width) → (real-in 0 255)

Returns the pen width.

```
(send a-pen is-immutable?) → boolean?
```

Returns `#t` if the pen object is immutable.

```
(send a-pen set-cap cap-style) → void?
  cap-style : pen-cap-style/c
```

Sets the pen cap style. See `get-cap` for information about cap styles.

A pen cannot be modified if it was obtained from a `pen-list%` or while it is selected into a drawing context.

```
(send a-pen set-color color) → void?
  color : (is-a?/c color%)
(send a-pen set-color color-name) → void?
  color-name : string?
(send a-pen set-color red green blue) → void?
  red : byte?
  green : byte?
  blue : byte?
```

Sets the pen color.

A pen cannot be modified if it was obtained from a `pen-list%` or while it is selected into a drawing context.

```
(send a-pen set-join join-style) → void?
  join-style : pen-join-style/c
```

Sets the pen join style. See `get-join` for information about join styles.

A pen cannot be modified if it was obtained from a `pen-list%` or while it is selected into a drawing context.

```
(send a-pen set-stipple bitmap) → void?
  bitmap : (or/c (is-a?/c bitmap%) #f)
```

Sets the pen pen stipple bitmap, where `#f` turns off the stipple bitmap.

If *bitmap* is modified while is associated with a pen, the effect on the pen is unspecified. A pen cannot be modified if it was obtained from a `pen-list%` or while it is selected into a drawing context.

```
(send a-pen set-style style) → void?
  style : pen-style/c
```

Sets the pen style. See pen% for information about the possible styles.

A pen cannot be modified if it was obtained from a pen-list% or while it is selected into a drawing context.

```
(send a-pen set-width width) → void?
  width : (real-in 0 255)
```

Sets the pen width.

A pen cannot be modified if it was obtained from a pen-list% or while it is selected into a drawing context.

18 pen-list%

```
pen-list% : class?
  superclass: object%
```

A pen-list% object maintains a list of pen% objects to avoid repeatedly creating pen objects. A pen% object in a pen list cannot be mutated.

A global pen list the-pen-list is created automatically.

```
(new pen-list%) → (is-a?/c pen-list%)
```

Creates an empty pen list.

```
(send a-pen-list find-or-create-pen  color
                                     width
                                     style
                                    [cap
                                     join]) → (is-a?/c pen%)
  color : (is-a?/c color%)
  width : (real-in 0 255)
  style : (or/c 'transparent 'solid 'xor 'hilite
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash)
  cap : (or/c 'round 'projecting 'butt) = 'round
  join : (or/c 'round 'bevel 'miter) = 'round
(send a-pen-list find-or-create-pen  color-name
                                     width
                                     style
                                    [cap
                                     join])
 → (or/c (is-a?/c pen%) #f)
  color-name : string?
  width : (real-in 0 255)
  style : (or/c 'transparent 'solid 'xor 'hilite
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash)
  cap : (or/c 'round 'projecting 'butt) = 'round
  join : (or/c 'round 'bevel 'miter) = 'round
```

Finds a pen of the given specification, or creates one and adds it to the list. The arguments are the same as for creating a pen% instance plus a cap and join style as for set-cap and set-join. When *color-name* is provided, however, the return value is #f when no color matching *color-name* can be found in the-color-database.

19 point%

```
point% : class?
  superclass: object%
```

A point% is used for certain drawing commands. It encapsulates two real numbers.

```
(make-object point%) → (is-a?/c point%)
(make-object point% x y) → (is-a?/c point%)
  x : real?
  y : real?
```

Creates a point. If *x* and *y* are not supplied, they are set to 0.

```
(send a-point get-x) → real?
```

Gets the point x-value.

```
(send a-point get-y) → real?
```

Gets the point y-value.

```
(send a-point set-x x) → void?
  x : real?
```

Sets the point x-value.

```
(send a-point set-y y) → void?
  y : real?
```

Sets the point y-value.

20 post-script-dc%

```
post-script-dc% : class?
  superclass: object%
  extends: dc<%>
```

A post-script-dc% object is a PostScript device context, that can write PostScript files on any platform. See also ps-setup% and pdf-dc%.

Be sure to use the following methods to start/end drawing:

- start-doc
- start-page
- end-page
- end-doc

Attempts to use a drawing method outside of an active page raises an exception.

See also printer-dc%.

```
(new post-script-dc%
   [[interactive interactive]
    [parent parent]
    [use-paper-bbox use-paper-bbox]
    [as-eps as-eps]
    [width width]
    [height height]
    [output output]])
→ (is-a?/c post-script-dc%)
 interactive : any/c = #t
 parent : (or/c (is-a?/c frame%) (is-a?/c dialog%) #f) = #f
 use-paper-bbox : any/c = #f
 as-eps : any/c = #t
 width : (or/c (and/c real? (not/c negative?)) #f) = #f
 height : (or/c (and/c real? (not/c negative?)) #f) = #f
 output : (or/c path-string? output-port? #f) = #f
```

If *interactive* is true, the user is given a dialog for setting printing parameters (see get-ps-setup-from-user); the resulting configuration is installed as the current configuration). If the user chooses to print to a file (the only possibility on Windows and Mac OS X), another dialog is given to select the filename. If the user hits cancel in either of these dialogs, then ok? returns #f.

If *parent* is not #f, it is used as the parent window of the configuration dialog.

If *interactive* is #f, then the settings returned by current-ps-setup are used. A file dialog is still presented to the user if the get-file method returns #f and *output* is #f, and the user may hit Cancel in that case so that ok? returns #f.

If *use-paper-bbox* is #f, then the PostScript bounding box for the output is determined by *width* and *height* (which are rounded upward using ceiling). If *use-paper-bbox* is not #f, then the bounding box is determined by the current paper size (as specified by current-ps-setup). When *width* or *height* is #f, then the corresponding dimension is determined by the paper size, even if *use-paper-bbox* is #f.

If *as-eps* is #f, then the generated PostScript does not include an Encapsulated PostScript (EPS) header, and instead includes a generic PostScript header. The margin and translation factors specified by current-ps-setup are used only when *as-eps* is #f. If *as-eps* is true, then the generated PostScript includes a header that identifiers it as EPS.

When *output* is not #f, then file-mode output is written to *output*. If *output* is #f, then the destination is determined via current-ps-setup or by prompting the user for a path-name. When *output* is a port, then data is written to port by a thread that is created with the post-script-dc% instance; in case that writing thread's custodian is shut down, calling end-doc resumes the port-writing thread with thread-resume and (current-thread) as the second argument.

See also ps-setup% and current-ps-setup. The settings for a particular post-script-dc% object are fixed to the values in the current configuration when the object is created (after the user has interactively adjusted them when *interactive* is true).

21 ps-setup%

```
ps-setup% : class?
  superclass: object%
```

A ps-setup% object contains configuration information for producing PostScript files using a post-script-dc% object. To a lesser extent, it contains information for printing with a printer-dc% object.

When a post-script-dc% object is created, its configuration is determined by the current-ps-setup parameter's ps-setup% value. After a post-script-dc% object is created, it is unaffected by changes to the current-ps-setup parameter or mutations to the ps-setup% object.

```
(new ps-setup%) → (is-a?/c ps-setup%)
```

Creates a new ps-setup% object with the (platform-specific) default configuration.

```
(send a-ps-setup copy-from  source
                            [copy-filename?]) → void?
  source : (is-a?/c ps-setup%)
  copy-filename? : any/c = #f
```

Copies the settings copy-from to *a-ps-setup*, excluding the filename unless *copy-filename?* is true.

```
(send a-ps-setup get-command) → string?
```

Historically, gets the printer command used to print a file on Unix. The default is "lpr". This value is not currently used by any platforms.

```
(send a-ps-setup get-editor-margin h-margin
                                   v-margin) → void?
  h-margin : (box/c (and/c real? (not/c negative?)))
  v-margin : (box/c (and/c real? (not/c negative?)))
```

Returns the current settings for horizontal and vertical margins when printing an editor<%>. See also set-editor-margin.

```
(send a-ps-setup get-file) → (or/c path-string? #f)
```

Gets the PostScript output filename. A #f value (the default) indicates that the user should be prompted for a filename when a post-script-dc% object is created.

```
(send a-ps-setup get-level-2) → boolean?
```

Reports whether Level 2 commands are output in PostScript files.

Currently, Level 2 commands are only needed to include color bitmap images in PostScript output (drawn with `draw-bitmap`), or bitmap pen and brush stipples. When Level 2 commands are disabled, bitmaps are converted to grayscale images and stipples are not supported.

```
(send a-ps-setup get-margin h-margin
                            v-margin) → void?
  h-margin : (box/c (and/c real? (not/c negative?)))
  v-margin : (box/c (and/c real? (not/c negative?)))
```

Returns the current settings for horizontal and vertical PostScript margins. See also `set-margin`.

```
(send a-ps-setup get-mode) → (or/c 'preview 'file 'printer)
```

Gets the printing mode that determines where output is sent: `'preview`, `'file`, or `'printer`. The default for X is `'preview`. The value in Windows and Mac OS X is always `'file`.

```
(send a-ps-setup get-orientation)
 → (or/c 'portrait 'landscape)
```

Gets the orientation: `'portrait` or `'landscape`. The default is `'portrait`. Unlike most other settings, this one affects native printing (via `printer-dc%`) as well as PostScript output.

Landscaped orientation affects the size of the drawing area as reported by `get-size`: the horizontal and vertical sizes determined by the selected paper type are transposed and then scaled.

```
(send a-ps-setup get-paper-name) → string?
```

Returns the name of the current paper type: `"A4 210 x 297 mm"`, `"A3 297 x 420 mm"`, `"Letter 8 1/2 x 11 in"`, or `"Legal 8 1/2 x 14 in"`. The default is `"Letter 8 1/2 x 11 in"`.

The paper name determines the size of the drawing area as reported by `get-size` (along with landscape transformations from `get-orientation` and/or the scaling factors of `get-scaling`). It also determines the bounding box of PostScript output when a `post-script-dc%` context is created with a true value for the `use-paper-bbox?` initialization argument.

```
(send a-ps-setup get-preview-command) → string?
```

Gets the command used to view a PostScript file for X. The default is `"gv"`. This value is not used by other platforms.

```
(send a-ps-setup get-scaling x y) → void?
  x : (box/c (and/c real? (not/c negative?)))
  y : (box/c (and/c real? (not/c negative?)))
```

Gets the scaling factor for PostScript output. The *x* box is filled with the horizontal scaling factor. The *y* box is filled with the vertical scaling factor. The default is 0.8 by 0.8.

This scale is in addition to a scale that can be set by set-scale in a post-script-dc% context. The size reported by get-size is the size of the selected paper type (transposed for landscaped mode) divided by this scale.

```
(send a-ps-setup get-translation x y) → void?
  x : (box/c (and/c real? (not/c negative?)))
  y : (box/c (and/c real? (not/c negative?)))
```

Gets the translation (from the bottom left corner) for PostScript output. The *x* box is filled with the horizontal offset. The *y* box is filled with the vertical offset. The default is 0.0 and 0.0.

The translation is not scaled by the numbers returned from get-scaling and the translation does not affect the size of the drawing area.

```
(send a-ps-setup set-command command) → void?
  command : string?
```

Historically, sets the printer command that was used to print a file on Unix. See get-command.

```
(send a-ps-setup set-editor-margin h v) → void?
  h : exact-nonnegative-integer?
  v : exact-nonnegative-integer?
```

Sets the horizontal and vertical margins used when printing an editor with the print method. These margins are always used for printing, whether the drawing destination is a post-script-dc% or printer-dc%. The margins are in the units of the destination printer-dc% or post-script-dc%. In the case of post-script-dc% printing, the editor margin is in addition to the PostScript margin that is determined by set-margin.

```
(send a-ps-setup set-file filename) → void?
  filename : (or/c path-string? #f)
```

Sets the PostScript output filename. See get-file.

```
(send a-ps-setup set-level-2 on?) → void?
  on? : any/c
```

Sets whether Level 2 commands are output in PostScript files. See get-level-2.

```
(send a-ps-setup set-margin h v) → void?
  h : (and/c real? (not/c negative?))
  v : (and/c real? (not/c negative?))
```

Sets the horizontal and vertical PostScript margins. When drawing to a post-script-dc%, the page size reported by get-size subtracts these margins from the normal page area (before taking into account scaling affects). In addition, drawing into the post-script-dc% produces PostScript output that is offset by the margins.

When using the output of a post-script-dc% as Encapsulated PostScript, the margin values are effectively irrelevant. Changing the margins moves the PostScript image in absolute coordinates, but it also moves the bounding box.

The margins are in unscaled post-script-dc% units, which are points. The default margins are 16 points.

```
(send a-ps-setup set-mode mode) → void?
  mode : (or/c 'preview 'file 'printer)
```

Sets the printing mode controlling where output is sent. See get-mode.

On Windows and Mac OS X, if 'preview or 'printer is provided, an exn:fail:contract exception is raised.

```
(send a-ps-setup set-orientation orientation) → void?
  orientation : (or/c 'portrait 'landscape)
```

Sets the orientation. See get-orientation.

```
(send a-ps-setup set-paper-name type) → void?
  type : string?
```

Sets the name of the current paper type. See get-paper-name.

```
(send a-ps-setup set-preview-command command) → void?
  command : string?
```

Sets the command used to view a PostScript file on Unix. See get-preview-command.

```
(send a-ps-setup set-scaling x y) → void?
  x : (and/c real? (not/c negative?))
  y : (and/c real? (not/c negative?))
```

Sets the scaling factor for PostScript output. See get-scaling.

```
(send a-ps-setup set-translation x y) → void?
  x : real?
  y : real?
```

Sets the translation (from the bottom left corner) for PostScript output. See get-translation.

22 radial-gradient%

```
radial-gradient% : class?
  superclass: object%
```

A *radial gradient* is used with a brush% to fill areas with smooth color transitions. Color transitions are based on two circles and the sequence of circles that "morph" from the starting circle to the ending circle. Normally, one of the two circles defining a gradient is nested within the other; in that case, points within the inner circle get the same color as the inner circle's edge, while points outside the outer circle get the same color as the outer circle's edge.

```
(new radial-gradient%
   [x0 x0]
   [y0 y0]
   [r0 r0]
   [x1 x1]
   [y1 y1]
   [r1 r1]
   [stops stops])
→ (is-a?/c radial-gradient%)
 x0 : real?
 y0 : real?
 r0 : real?
 x1 : real?
 y1 : real?
 r1 : real?
 stops : (listof (list/c (real-in 0 1) (is-a?/c color%)))
```

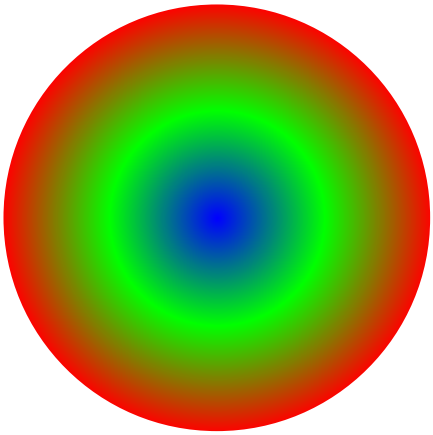
Creates a radial gradient with the starting circle as the one with radius *r0* centered at (*x0*, *y0*) and the ending circle as the one with radius *r1* centered at (*x1*, *y1*). The *stops* list assigns colors to circles, where 0.0 corresponds to the starting circle, 1.0 corresponds to the ending circle, and numbers in between correspond to circles in between.

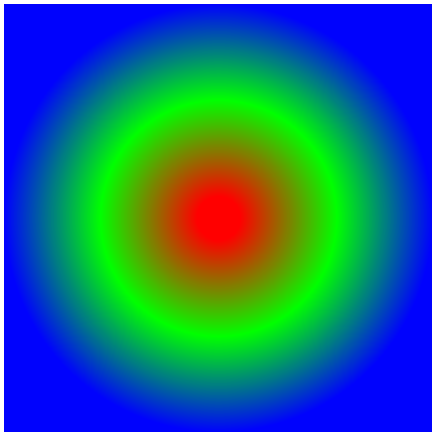
The order of elements within *stops* and duplicate points are treated in the same way for as linear-gradient%.

Examples:

```
> (define ellipse-brush
    (new brush%
         [gradient
          (new radial-gradient%
               [x0 100] [y0 100] [r0 0]
               [x1 100] [y1 100] [r1 100]
               [stops
```

```
                (list (list 0   (make-object color% 0 0 255))
                      (list 0.5 (make-object color% 0 255 0))
                      (list 1   (make-object color% 255 0 0))))]))))

> (define rectangle-brush
    (new brush%
         [gradient
          (new radial-gradient%
               [x0 100] [y0 100] [r0 10]
               [x1 100] [y1 100] [r1 100]
               [stops
                (list (list 0   (make-object color% 255 0 0))
                      (list 0.5 (make-object color% 0 255 0))
                      (list 1   (make-object color% 0 0 255)))])]))

> (dc
   (λ (dc dx dy)
     (define old-pen (send dc get-pen))
     (define old-brush (send dc get-brush))
     (define-values (ox oy) (send dc get-origin))

     (send dc set-pen "black" 1 'transparent)
     (send dc set-brush ellipse-brush)
     (send dc set-origin (+ ox dx 50) (+ oy dy 50))
     (send dc draw-ellipse 0 0 200 200)

     (send dc set-origin (+ ox dx 300) (+ oy dy 50))
     (send dc set-brush rectangle-brush)
     (send dc draw-rectangle 0 0 200 200)

     (send dc set-pen old-pen)
     (send dc set-brush old-brush)
     (send dc set-origin ox oy))
   550 300)
```





```
(send a-radial-gradient get-circles)
 → real? real? real? real? real? real?
```

Returns the gradient's boundary circles as *x0*, *y0*, *r0*, *x1*, *y1*, and *r1*.

```
(send a-radial-gradient get-stops)
 → (listof (list/c (real-in 0 1) (is-a?/c color%)))
```

Returns the gradient's list of color stops.

23 record-dc%

```
record-dc% : class?
  superclass: object%
  extends: dc<%>
```

A record-dc% object records drawing actions for replay into another drawing context. The recorded drawing operations can be extracted as a procedure via get-recorded-procedure, or the actions can be extracted as a datum (that can be printed with write and recreated with read) via get-recorded-datum.

When drawing recorded actions, the target drawing context's pen, brush, font, text, background, text background, and text foreground do not affect the recorded actions. The target drawing context's transformation, alpha, and clipping region compose with settings in the recorded actions (so that, for example, a recorded action to set the clipping region actually intersects the region with the drawing context's clipping region at the time that the recorded commands are replayed). After recoded commands are replayed, all settings in the target drawing context, such as its clipping region or current font, are as before the replay.

```
(new record-dc%
   [[width width]
    [height height]]) → (is-a?/c record-dc%)
 width : (>=/c 0) = 640
 height : (>=/c 0) = 480
```

Creates a new recording DC. The optional *width* and *height* arguments determine the result of get-size on the recording DC; the *width* and *height* arguments do not clip drawing.

```
(send a-record-dc get-recorded-datum) → any/c
```

Extracts a recorded drawing to a value that can be printed with write and re-read with read. Use recorded-datum->procedure to convert the datum to a drawing procedure.

```
(send a-record-dc get-recorded-procedure)
 → ((is-a?/c dc<%>) . -> . void?)
```

Extracts a recorded drawing to a procedure that can be applied to another DC to replay the drawing commands to the given DC.

The get-recorded-procedure method can be more efficient than composing get-recorded-datum and recorded-datum->procedure.

24 region%

```
region% : class?
  superclass: object%
```

A region% object specifies a portion of a drawing area (possibly discontinuous). It is normally used for clipping drawing operations.

A region% object can be associated to a particular dc<%> object when the region is created. In that case, the region uses the drawing context's current transformation matrix, translation, scaling, and rotation, independent of the transformation that is in place when the region is installed. Otherwise, the region is transformed as usual when it is installed into a dc<%>. For an auto-scrolled canvas, the canvas's current scrolling always applies when the region is used (and it does not affect the region's bounding box).

Region combination with operations like region% union are approximate, and they are implemented by combining paths. Certain combinations work only if the paths have a suitable fill mode, which can be either 'winding, 'even-odd, or a *flexible fill* mode. When a region is installed as a device context's clipping region, any subpath with a flexible fill mode uses 'even-odd mode if any other path uses 'even-odd mode.

See also set-clipping-region in dc<%> and get-clipping-region in dc<%>.

```
(new region% [dc dc]) → (is-a?/c region%)
  dc : (or/c (is-a?/c dc<%>) #f)
```

Creates an empty region. If *dc* is a dc<%> object, the dc<%>'s current transformation matrix is essentially recorded in the region.

```
(send a-region get-bounding-box) → real? real? real? real?
```

Returns a rectangle that approximately encloses the region. The return values are the left, top, width, and height of the rectangle. If the region has an associated drawing context, the bounding box is in the drawing context's current logical coordinates.

```
(send a-region get-dc) → (or/c (is-a?/c dc<%>) #f)
```

Returns the region's drawing context, if it was created for one.

```
(send a-region in-region? x y) → boolean?
  x : real?
  y : real?
```

Returns #t if the given point is approximately within the region, #f otherwise. If the region has an associated drawing context, the given point is effectively transformed according to the region's dc<%>'s current transformation matrix.

```
(send a-region intersect rgn) → void?
  rgn : (is-a?/c region%)
```

Sets the region to the intersection of itself with the given region.

The drawing context of *rgn* and *a-region* must be the same, or they must both be unassociated to any drawing context.

An intersect corresponds to clipping with this region's path, and then clipping with the given region's path. Further combining sends to this region correspond to combination with the original path before initial clip, and further combination with this region as an argument correspond to a combination with the given path after the initial clip. Thus, an intersecting region is a poor input for `union`, `subtract`, or `xor`, but it intersects properly in further calls to `intersect`.

```
(send a-region is-empty?) → boolean?
```

Returns `#t` if the region is approximately empty, `#f` otherwise, but only if the region is associated with a drawing context. If the region is unassociated to any drawing context, the `exn:fail:contract` exception is raised.

```
(send a-region set-arc x
                       y
                       width
                       height
                       start-radians
                       end-radians)  → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  start-radians : real?
  end-radians : real?
```

Sets the region to the interior of the specified wedge.

See also `draw-ellipse` in `dc<%>`, since the region content is determined the same way as brush-based filling in a `dc<%>`.

The region corresponds to a clockwise path with a flexible fill. The region is also atomic for the purposes of region combination.

```
(send a-region set-ellipse x y width height) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Sets the region to the interior of the specified ellipse.

See also draw-ellipse in dc<%>, since the region content is determined the same way as brush-based filling in a dc<%>.

The region corresponds to a clockwise path with a flexible fill. The region is also atomic for the purposes of region combination.

```
(send a-region set-path  path
                        [xoffset
                         yoffset
                         fill-style]) → void?
  path : (is-a?/c dc-path%)
  xoffset : real? = 0
  yoffset : real? = 0
  fill-style : (or/c 'odd-even 'winding) = 'odd-even
```

Sets the region to the content of the given path.

See also draw-path in dc<%>, since the region content is determined the same way as brush-based filling in a dc<%>.

The fill style affects how well the region reliably combines with other regions (via union, xor, and subtract). The region is also atomic for the purposes of region combination.

```
(send a-region set-polygon  points
                           [xoffset
                            yoffset
                            fill-style]) → void?
  points : (or/c (listof (is-a?/c point%))
                 (listof (cons/c real? real?)))
  xoffset : real? = 0
  yoffset : real? = 0
  fill-style : (or/c 'odd-even 'winding) = 'odd-even
```

Sets the region to the interior of the polygon specified by *points*. A pair is treated as a point where the car of the pair is the x-value and the cdr is the y-value.

See also draw-polygon in dc<%>, since the region content is determined the same way as brush-based filling in a dc<%>.

The fill style affects how well the region reliably combines with other regions (via union, xor, and subtract). The region is also atomic for the purposes of region combination.

```
(send a-region set-rectangle x
                             y
                             width
                             height) → void?
```

```
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
```

Sets the region to the interior of the specified rectangle.

The region corresponds to a clockwise path with a flexible fill. The region is also atomic for the purposes of region combination.

```
(send a-region set-rounded-rectangle  x
                                      y
                                      width
                                      height
                                     [radius]) → void?
  x : real?
  y : real?
  width : (and/c real? (not/c negative?))
  height : (and/c real? (not/c negative?))
  radius : real? = -0.25
```

Sets the region to the interior of the specified rounded rectangle.

See also `draw-rounded-rectangle` in `dc<%>`, since the region content is determined the same way as brush-based filling in a `dc<%>`.

The region corresponds to a clockwise path with a flexible fill. The region is also atomic for the purposes of region combination.

```
(send a-region subtract rgn) → void?
  rgn : (is-a?/c region%)
```

Sets the region to the subtraction of itself minus the given region. In other words, a point is removed from the region if it is included in the given region. (The given region may contain points that are not in the current region; such points are ignored.)

This region's drawing context and given region's drawing context must be the same, or they must both be unassociated to any drawing context.

The result is consistent across platforms and devices, but it is never a true subtraction. A subtraction corresponds to combining the sub-paths of this region with the reversed sub-paths of the given region, then intersecting the result with this region. This fails as a true subtraction, because the boundary of loops (with either `'odd-even` or `'winding` filling) is ambiguous.

```
(send a-region union rgn) → void?
  rgn : (is-a?/c region%)
```

Sets the region to the union of itself with the given region.

This region's drawing context and given region's drawing context must be the same, or they must both be unassociated to any drawing context.

A union corresponds to combining the sub-paths of each region into one path, using an `'odd-even` fill if either of the region uses an `'odd-even` fill (otherwise using a `'winding` fill), a `'winding` fill in either region uses a `winding` fill, or the fill remains a flexible fill if both paths have a flexible fill. Consequently, while the result is consistent across platforms and devices, it is a true union only for certain input regions. For example, it is a true union for non-overlapping *atomic* and union regions. It is also a true union for atomic and union regions (potentially overlapping) that are all clockwise and use `'winding` fill or if the fills are all flexible fills.

```
(send a-region xor rgn) → void?
  rgn : (is-a?/c region%)
```

Sets the region to the xoring of itself with the given region (i.e., contains points that are enclosed by exactly one of the two regions).

This region's drawing context and given region's drawing context must be the same, or they must both be unassociated to any drawing context.

The result is consistent across platforms and devices, but it is not necessarily a true xoring. An xoring corresponds to combining the sub-paths of this region with the reversed sub-paths of the given region. The result uses an `'odd-even` fill if either of the region uses an `'odd-even` fill, a `'winding` fill in either region uses a `winding` fill, or the fill remains a flexible fill if both paths have a flexible fill. Consequently, the result is a reliable xoring only for certain input regions. For example, it is reliable for atomic and xoring regions that all use `'even-odd` fill.

25 svg-dc%

```
svg-dc% : class?
  superclass: object%
  extends: dc<%>
```

Similar to post-script-dc%, but generates a SVG (scalable vector graphics) file instead of a PostScript file.

Be sure to use the following methods to start/end drawing:

- start-doc
- start-page
- end-page
- end-doc

Attempts to use a drawing method outside of an active page raises an exception.

```
(new svg-dc%
   [width width]
   [height height]
   [output output]
  [[exists exists]]) → (is-a?/c svg-dc%)
 width : (and/c real? (not/c negative?))
 height : (and/c real? (not/c negative?))
 output : (or/c path-string? output-port?)
 exists : (or/c 'error 'append 'update 'can-update = 'error
                'replace 'truncate
                'must-truncate 'truncate/replace)
```

The *width* and *height* arguments determine the width and height of the generated image.

The image is written to *output*. If *output* is a path and the file exists already, then *exists* determines how the existing file is handled in the same way as for the #:exists argument to open-output-file.

26 Drawing Functions

```
(current-ps-setup) → (is-a?/c ps-setup%)
(current-ps-setup pss) → void?
  pss : (is-a?/c ps-setup%)
```

A parameter that determines the current PostScript configuration settings. See post-script-dc% and printer-dc%.

```
(get-face-list [kind
                #:all-variants? all-variants?])
 → (listof string?)
  kind : (or/c 'mono 'all) = 'all
  all-variants? : any/c = #f
```

Returns a list of font face names available on the current system. If *kind* is 'mono, then only faces that are known to correspond to monospace fonts are included in the list.

If *all-variants?* is #f (the default), then the result is in more standard terminology a list of font family names, which are combined with style and weight options to arrive at a face; if *all-variants?* is true, then the result includes a string for each available face in the family.

```
(get-family-builtin-face family) → string?
  family : (or/c 'default 'decorative 'roman 'script
                 'swiss 'modern 'symbol 'system)
```

Returns the built-in default face mapping for a particular font family.

See font% for information about *family*.

```
(make-bitmap  width
              height
             [alpha?
              #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  width : exact-positive-integer?
  height : exact-positive-integer?
  alpha? : any/c = #t
  backing-scale : (>/c 0.0) = 1.0
```

Returns (make-object bitmap% *width height* #f *alpha? backing-scale*), but this procedure is preferred because it defaults *alpha?* in a more useful way.

See also make-platform-bitmap and §1.8 “Portability and Bitmap Variants”.

Changed in version 1.1 of package `draw-lib`: Added the `#:backing-scale` optional argument.

```
(make-brush [#:color color
             #:style style
             #:stipple stipple
             #:gradient gradient
             #:transformation transformation
             #:immutable? immutable?])        → (is-a?/c brush%)
  color : (or/c string? (is-a?/c color%)) = (make-color 0 0 0)
  style : (or/c 'transparent 'solid 'opaque        = 'solid
                'xor 'hilite 'panel
                'bdiagonal-hatch 'crossdiag-hatch
                'fdiagonal-hatch 'cross-hatch
                'horizontal-hatch 'vertical-hatch)
  stipple : (or/c #f (is-a?/c bitmap%)) = #f
  gradient : (or/c #f                           = #f
                   (is-a?/c linear-gradient%)
                   (is-a?/c radial-gradient%))
  transformation : (or/c #f (vector/c (vector/c real? real? real?
                                                 real? real? real?)
                                      real? real? real? real? real?))
                   = #f
  immutable? : any/c = #t
```

Creates a `brush%` instance. This procedure provides a nearly equivalent interface compared to using `make-object` with `brush%`, but it also supports the creation of immutable brushes (and creates immutable bursh es by default).

When *stipple* is `#f`, *gradient* is `#f`, *transformation* is `#f`, *immutable?* is true, and *color* is either a `color%` object or a string in `the-color-database`, the result brush is created via `find-or-create-brush` of `the-brush-list`.

```
(make-color red green blue [alpha]) → (is-a?/c color%)
  red : byte?
  green : byte?
  blue : byte?
  alpha : (real-in 0 1) = 1.0
```

Creates a `color%` instance. This procedure provides a nearly equivalent interface compared to using `make-object` with `color%`, but it creates an immutable `color%` object.

To create an immutable color based on a color string, use `find-color` or `the-color-database`.

```
(make-font [#:size size
            #:face face
            #:family family
            #:style style
            #:weight weight
            #:underlined? underlined?
            #:smoothing smoothing
            #:size-in-pixels? size-in-pixels?
            #:hinting hinting])                → (is-a?/c font%)
  size : (real-in 0.0 1024.0) = 12
  face : (or/c string? #f) = #f
  family : (or/c 'default 'decorative 'roman 'script = 'default
                 'swiss 'modern 'symbol 'system)
  style : (or/c 'normal 'italic 'slant) = 'normal
  weight : (or/c 'normal 'bold 'light) = 'normal
  underlined? : any/c = #f
  smoothing : (or/c 'default 'partly-smoothed = 'default
                    'smoothed 'unsmoothed)
  size-in-pixels? : any/c = #f
  hinting : (or/c 'aligned 'unaligned) = 'aligned
```

Creates a font% instance. This procedure provides an equivalent but more convenient interface compared to using make-object with font%.

Changed in version 1.4 of package draw-lib: Changed *size* to allow non-integer and zero values.

```
(make-monochrome-bitmap width height [bits]) → (is-a?/c bitmap%)
  width : exact-positive-integer?
  height : exact-positive-integer?
  bits : (or/c bytes? #f) = #f
```

Returns (make-object bitmap% *width height* #t) if *bits* is #f, or (make-object bitmap% *bits width height*) otherwise. This procedure is preferred to using make-object on bitmap% because it is less overloaded.

```
(make-pen [#:color color
           #:width width
           #:style style
           #:cap cap
           #:join join
           #:stipple stipple
           #:immutable? immutable?]) → (is-a?/c pen%)
  color : (or/c string? (is-a?/c color%)) = (make-color 0 0 0)
  width : (real-in 0 255) = 0
```

```
  style : (or/c 'transparent 'solid 'xor 'hilite        = 'solid
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash)
  cap : (or/c 'round 'projecting 'butt) = 'round
  join : (or/c 'round 'bevel 'miter) = 'round
  stipple : (or/c #f (is-a?/c bitmap%)) = #f
  immutable? : any/c = #t
```

Creates a pen% instance. This procedure provides a nearly equivalent interface compared to using make-object with pen%, but it also supports the creation of immutable pens (and creates immutable pens by default).

When *stipple* is #f, *immutable?* is true, and *color* is either a color% object or a string in the-color-database, the result pen is created via find-or-create-pen of the-pen-list.

```
(make-platform-bitmap  width
                       height
                      [#:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  width : exact-positive-integer?
  height : exact-positive-integer?
  backing-scale : (>/c 0.0) = 1.0
```

Creates a bitmap that uses platform-specific drawing operations as much as possible, which is different than a make-bitmap result on Windows and Mac OS X. See §1.8 “Portability and Bitmap Variants” for more information.

Changed in version 1.1 of package draw-lib: Added the #:backing-scale optional argument.

```
(read-bitmap  in
             [kind
              bg-color
              complain-on-failure?
              #:backing-scale backing-scale
              #:try-@2x? try-@2x?])          → (is-a?/c bitmap%)
  in : (or path-string? input-port?)
  kind : (or/c 'unknown 'unknown/mask 'unknown/alpha
               'gif 'gif/mask 'gif/alpha
               'jpeg 'jpeg/alpha
               'png 'png/mask 'png/alpha
               'xbm 'xbm/alpha 'xpm 'xpm/alpha
               'bmp 'bmp/alpha)
       = 'unknown/alpha
  bg-color : (or/c (is-a?/c color%) #f) = #f
```

```
  complain-on-failure? : any/c = #t
  backing-scale : (>/c 0.0) = 1.0
  try-@2x? : any/c = #f
```

Returns `(make-object bitmap% in kind bg-color complain-on-failure? backing-scale)`, but this procedure is preferred because it defaults *kind* and *complain-on-failure?* in a more useful way.

If *try-@2x?* is true, *in* is a path, and *kind* is not one of the `/mask` symbols, then `read-bitmap` checks whether a file exists matching *in* but with `"@2x"` added to the name (before the file suffix, if any). If the `"@2x"` path exists, it is used instead of *in*, and `backing-store` is multiplied by 2.

Changed in version 1.1 of package `draw-lib`: Added the `#:backing-scale` and `#:try-@2x?` optional arguments.

```
(recorded-datum->procedure datum)
 → ((is-a?/c dc<%>) . -> . void?)
  datum : any/c
```

Converts a value from `get-recorded-datum` in `record-dc%` to a drawing procedure.

```
the-brush-list : (is-a?/c brush-list%)
```

See `brush-list%`.

```
the-color-database : (is-a?/c color-database<%>)
```

See `color-database<%>`.

```
the-font-list : (is-a?/c font-list%)
```

See `font-list%`.

```
the-font-name-directory : (is-a?/c font-name-directory<%>)
```

See `font-name-directory<%>`.

```
the-pen-list : (is-a?/c pen-list%)
```

See `pen-list%`.

27 Drawing Contracts

This page documents the contracts that are used to describe the specification of `racket/draw` objects and functions.

`font-family/c : flat-contract?`

Recognizes font designations. Corresponds to the *family* initialization argument of the `font%` class.

Equivalent to the following definition:

```
(or/c 'default 'decorative 'roman 'script 'swiss
      'modern 'symbol 'system)
```

`font-style/c : flat-contract?`

Recognizes font styles. Corresponds to the *style* initialization argument of the `font%` class.

Equivalent to the following definition:

```
(or/c 'normal 'italic 'slant)
```

`font-weight/c : flat-contract?`

Recognizes font weights. Corresponds to the *weight* initialization argument of the `font%` class.

Equivalent to the following definition:

```
(or/c 'normal 'bold 'light)
```

`font-smoothing/c : flat-contract?`

Recognizes a font smoothing amount. Corresponds to the *smoothing* initialization argument of the `font%` class.

Equivalent to the following definition:

```
(or/c 'default 'partly-smoothed
      'smoothed 'unsmoothed)
```

font-hinting/c : flat-contract?

Recognizes font hinting modes. Corresponds to the *hinting* initialization argument of the font% class.

Equivalent to the following definition:

```
(or/c 'aligned 'unaligned)
```

pen-style/c : flat-contract?

Recognizes pen styles. Corresponds to the *style* initialization argument of the pen% class.

Equivalent to the following definition:

```
(or/c 'transparent 'solid 'xor 'hilite
      'dot 'long-dash 'short-dash 'dot-dash
      'xor-dot 'xor-long-dash 'xor-short-dash
      'xor-dot-dash)
```

pen-cap-style/c : flat-contract?

Recognizes pen cap styles. Corresponds to the *cap* initialization argument of the pen% class.

Equivalent to the following definition:

```
(or/c 'round 'projecting 'butt)
```

pen-join-style/c : flat-contract?

Recognizes pen join styles. Corresponds to the *join* initialization argument of the pen% class.

Equivalent to the following definition:

```
(or/c 'round 'bevel 'miter)
```

brush-style/c : flat-contract?

Recognizes brush styles. Corresponds to the *style* initialization argument of the brush% class.

Equivalent to the following definition:

```
(or/c 'transparent 'solid 'opaque
      'xor 'hilite 'panel
      'bdiagonal-hatch 'crossdiag-hatch
      'fdiagonal-hatch 'cross-hatch
      'horizontal-hatch 'vertical-hatch)
```

28 Signature and Unit

The racket/draw/draw-sig and racket/draw/draw-unit libraries define the draw^ signature and draw@ implementation.

28.1 Draw Unit

```
(require racket/draw/draw-unit)
```
package: draw-lib

draw@ : unit?

Re-exports all of the exports of racket/draw.

28.2 Draw Signature

```
(require racket/draw/draw-sig)
```
package: draw-lib

draw^ : signature

Includes all of the identifiers exported by racket/draw.

29 Unsafe Libraries

The racket/draw library is currently implemented using Cairo and Pango. The get-handle in bitmap% method exposes the underlying Cairo surface for a bitmap% object, while make-handle-brush supports the creation of a brush from an existing Cairo surface. The representation of handles for these methods, however, is subject to change if the racket/draw library is implemented differently in the future.

29.1 Handle Brushes

```
(require racket/draw/unsafe/brush)
```
package: draw-lib

```
(make-handle-brush handle
                   width
                   height
                   transformation
                   [#:copy? copy?]) → (is-a?/c brush%)
  handle : cpointer?
  width : exact-nonnegative-integer?
  height : exact-nonnegative-integer?
  transformation : (or/c #f (vector/c (vector/c real? real? real?
                                                real? real? real?)
                                      real? real? real? real? real?))
  copy? : any/c = #t
```

Creates a brush given a *handle* that (currently) is a cairo_surface_t. If *copy?* is true, then the surface is copied, so that it can be freed or modified after the brush is created; if *copy?* is #f, the surface must remain available and unchanged as long as the brush can be used.

The *width* and *height* arguments specify the surface bounds for use when the surface must be copied—even when *copy?* is #f. The surface may need to be converted to a stipple bitmap, for example, when drawing to a monochrome target.

The given surface is treated much like a stipple bitmap: it is implicitly repeated, and the given *transformation* (if any) determines the surface's alignment relative to the target drawing context.

When the brush is used with a record-dc% object, and if that object's get-recorded-datum method is called, then the surface is effectively converted to a stipple bitmap for the result datum.

29.2 Cairo Library

```
(require racket/draw/unsafe/cairo-lib)
```
package: draw-lib

```
cairo-lib : (or/c ffi-lib? #f)
```

A reference to the Cairo library for use with functions such as `get-ffi-obj`, or `#f` if Cairo is unavailable.

30 Platform Dependencies

On Windows and Mac OS X, the Racket distribution includes all necessary libraries that are not part of a stock installation of the operating system, and the libraries are included in any distribution created with `raco distribute` (see §3 "`raco distribute`: Sharing Stand-Alone Executables").

On Unix, the following system libraries must be installed. Numbers in square brackets indicate a version that is tried first, and if that fails, the name without the version is tried.

- `"libglib-2.0[.0]"`
- `"libgmodule-2.0[.0]"`
- `"libgobject-2.0[.0]"`
- `"libpango-1.0[.0]"`
- `"libpangocairo-1.0[.0]"`
- `"libcairo[.2]"`
- `"libjpeg[.{62,8,9}]"`
- `"libpng[{16[.16],15[.15],12[.0]]}"`

Bibliography

[Adobe99] Adobe Systems Incorporated, *PostScript Language Reference, third edition*. 1999. `http://partners.adobe.com/public/developer/en/ps/PLRM.pdf`

Index