# Images

Version 9.2.0.5

Neil Toronto <neil.toronto@gmail.com>

June 8, 2026

This library contains convenient functions for constructing icons and logos, and will eventually offer the same for other `bitmap%`s. The idea is to make it easy to include such things in your own programs.

Generally, the images in this library are computed when requested, not loaded from disk. Most of them are drawn on a `dc<%>` and then ray traced. Ray tracing images can become computationally expensive, so this library also includes `images/compile-time`, which makes it easy to compute images at compile time and access them at run time.

The ray tracing API will eventually be finalized and made public. This Racket release begins doing so by finalizing and making public the basic image API used by the ray tracer. It is provided by the `images/flomap` module.

# Contents


**1 Icons** **4**

1.1 What is an icon? . . . . . 4

1.2 About These Icons . . . . . 5

1.3 Icon Style . . . . . 6

1.4 Arrow Icons . . . . . 10

1.5 Control Icons . . . . . 14

1.6 File Icons . . . . . 21

1.7 Symbol and Text Icons . . . . . 24

1.8 Miscellaneous Icons . . . . . 27

1.9 Stickman Icons . . . . . 37

1.10 Tool Icons . . . . . 38

**2 Logos** **42**

**3 Embedding Bitmaps in Compiled Files** **46**

**4 Floating-Point Bitmaps** **49**

4.1 Overview . . . . . 50

4.1.1 Motivation . . . . . 50

4.1.2 Conceptual Model . . . . . 51

4.1.3 Opacity (Alpha Components) . . . . . 53

4.1.4 Data Layout . . . . . 55

4.2 Struct Type and Accessors . . . . . 56

4.3 Conversion and Construction . . . . . 59

4.4 Component Operations . . . . . 64

4.5 Pointwise Operations . . . . . 68
4.6 Gradients and Normals . . . . . 74
4.7 Blur . . . . . 76
4.8 Resizing . . . . . 82
4.9 Compositing . . . . . 88
4.10 Spatial Transformations . . . . . 94
4.10.1 Provided Transformations . . . . . 95
4.10.2 General Transformations . . . . . 100
4.10.3 Lens Projection and Correction . . . . . 107
4.11 Effects . . . . . 112

# 1 Icons

## 1.1 What is an icon?

This section describes an ideal that DrRacket and its tools are steadily approaching.

As a first approximation, an icon is just a small `bitmap%`, usually with an alpha channel.

But an icon also communicates. Its shape and color are a visual metaphor for an action or a message. Icons should be **easily recognizable**, **distinguishable**, **visually consistent**, and **metaphorically appropriate** for the actions and messages they are used with. It can be difficult to meet all four requirements at once ("distinguishable" and "visually consistent" are often at odds), but good examples, good abstractions, and an existing icon library help considerably.

Example: The Macro Stepper icon is composed by appending a text icon and a step icon to get . The syntax quote icon is the color that DrRacket colors syntax quotes by default. The step icon is colored like DrRacket colors identifier syntax by default, and is shaped using metaphors used in debugger toolbars, TV remotes, and music players around the world. It is composed of to connote starting and to connote immediately stopping.

It would not do to have just as the Macro Stepper icon: it would be too easily confused with the Debugger icon , especially for new users and people with certain forms of color-blindness, and thus fail to be distinguishable enough.

As another example, the Check Syntax icon connotes inspecting and passing. Notice that the check mark is also the color of syntax.

## 1.2 About These Icons

The icons in this collection are designed to be composed to create new ones: they are simple, thematically consistent, and can be constructed in any size and color. Further, slideshow's `pict` combiners offer a way to compose them almost arbitrarily. For example, a media player application might create a large "step" button by superimposing a `record-icon` and a `step-icon`:

```
> (require pict images/icons/control images/icons/style)
> (pict->bitmap
   (cc-superimpose
    (bitmap (record-icon #:color "forestgreen" #:height 96
                         #:material glass-icon-material))
    (bitmap (step-icon #:color light-metal-icon-color #:height 48
                       #:material metal-icon-material))))
```

All the icons in this collection are first drawn using standard `dc<%>` drawing commands. Then, to get lighting effects, they are turned into 3D objects and ray traced. Many are afterward composed to create new icons; for example, the `stop-signs-icon` superimposes three `stop-sign-icon`s, and the `magnifying-glass-icon` is composed of three others (frame, glass and handle).

The ray tracer helps keep icons visually consistent with each other and with physical objects in day-to-day life. As an example of the latter, the `record-icon`, when rendered in clear glass, looks like the clear, round button on a Wii Remote. See the `plt-logo` and `planet-logo` functions for more striking examples.

When the rendering API is stable enough to publish, it will allow anyone who can draw a shape to turn that shape into a visually consistent icon.

As with any sort of rendering (such as SVG rendering), ray tracing takes time. For icons, this usually happens during tool or application start up. You can reduce the portion of start-up time taken by rendering to almost nothing by using the `images/compile-time` library to embed bitmaps directly into compiled modules.

## 1.3 Icon Style

```
(require images/icons/style)
```
package: images-lib

Use these constants and parameters to help keep icon sets visually consistent.

```
light-metal-icon-color : (or/c string? (is-a?/c color%))
 = "azure"
```

```
metal-icon-color : (or/c string? (is-a?/c color%))
 = "lightsteelblue"
```

```
dark-metal-icon-color : (or/c string? (is-a?/c color%))
 = "steelblue"
```

Good colors to use with metal-icon-material. See bomb-icon and magnifying-glass-icon for examples.

```
syntax-icon-color : (or/c string? (is-a?/c color%))
 = (make-object color% 76 76 255)
```

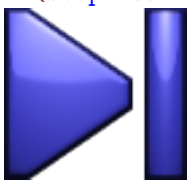
```
halt-icon-color : (or/c string? (is-a?/c color%))
 = (make-object color% 255 32 24)
```

```
run-icon-color : (or/c string? (is-a?/c color%)) = "lawngreen"
```

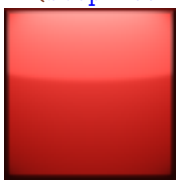
Standard toolbar icon colors.

Use syntax-icon-color in icons that connote macro expansion or syntax. Example:

```
> (step-icon #:color syntax-icon-color #:height 32)
```



Use halt-icon-color in icons that connote stopping or errors. Example:

```
> (stop-icon #:color halt-icon-color #:height 32)
```


Use `run-icon-color` in icons that connote executing programs or evaluation. Examples:

```
> (play-icon #:color run-icon-color #:height 32)
```

```
> (require images/icons/stickman)
> (running-stickman-icon 0.9 #:height 32
                         #:body-color run-icon-color
                         #:arm-color "white"
                         #:head-color run-icon-color)
```

For new users and for accessibility reasons, do not try to differentiate icons for similar functions only by color.

```
(default-icon-height) → (and/c rational? (>=/c 0))
(default-icon-height height) → void?
  height : (and/c rational? (>=/c 0))
 = 24
```

The height of DrRacket's standard icons.

```
(toolbar-icon-height) → (and/c rational? (>=/c 0))
(toolbar-icon-height height) → void?
  height : (and/c rational? (>=/c 0))
 = 16
```

The height of DrRacket toolbar icons.

Use (toolbar-icon-height) as the *height* argument for common icons that will be used in toolbars, status bars, and buttons.

(When making an icon for DrRacket's main toolbar, try to keep it nearly square so that it will not take up too much horizontal space when the toolbar is docked vertically. If you cannot, as with the Macro Stepper, send a thinner icon as the alternate-bitmap argument to a switchable-button%.)

```
(default-icon-backing-scale) → (and/c rational? (>/c 0))
(default-icon-backing-scale scale) → void?
  scale : (and/c rational? (>/c 0))
 = 2
```

The backing scale of DrRacket icons.

A backing scale of 2 means that the icon bitmap internally has two pixels per drawing unit, so it renders well at double resolution, such as Retina display mode for Mac OS.

Added in version 1.1 of package images-lib.

```
plastic-icon-material : deep-flomap-material-value?
```

```
rubber-icon-material : deep-flomap-material-value?
```

```
glass-icon-material : deep-flomap-material-value?
```

```
metal-icon-material : deep-flomap-material-value?
```

Materials for icons.

Plastic is opaque and reflects a little more than glass.

Rubber is also opaque, reflects more light than plastic, but diffuses less.

Glass is transparent but frosted, so it scatters refracted light. It has the high refractive index of cubic zirconia, or fake diamond. The "glassy look" cannot actually be achieved using glass.

Metal reflects the most, its specular highlight is nearly the same color as the material (in the others, the highlight is white), and it diffuses much more ambient light than directional. This is because while plastic and glass mostly reflect light directly, metal mostly absorbs light and re-emits it.

Examples:

```
> (require images/icons/misc)
> (for/list ([material  (list plastic-icon-material
                              rubber-icon-material
                              glass-icon-material
                              metal-icon-material)])
    (bomb-icon #:height 32 #:material material))
```

(list )

```
(default-icon-material) → deep-flomap-material-value?
(default-icon-material material) → void?
  material : deep-flomap-material-value?
 = plastic-icon-material
```

The material used for rendering most icons and icon parts. There are exceptions; for example, the `floppy-disk-icon` always renders the sliding cover in metal.

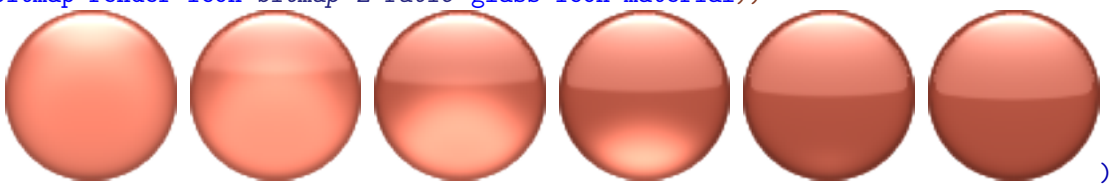
```
(bitmap-render-icon bitmap [z-ratio material]) → (is-a?/c bitmap%)
  bitmap : (is-a?/c bitmap%)
  z-ratio : (and rational? (>=/c 0)) = 5/8
  material : deep-flomap-material-value?
           = (default-icon-material)
```

Makes a 3D object out of *bitmap* and renders it as an icon.

The *z-ratio* argument only makes a difference when *material* is transparent, such as `glass-icon-material`. It controls what fraction of *bitmap*'s height the icon is raised, which in turn affects the refracted shadow under the icon: the higher the *z-ratio*, the lower the shadow.

Examples:

```
> (define bitmap
    (pict->bitmap (colorize (filled-ellipse 64 64) "tomato")))
> (for/list ([z-ratio  (in-range 0 2 1/3)])
    (bitmap-render-icon bitmap z-ratio glass-icon-material))
```

(list  )

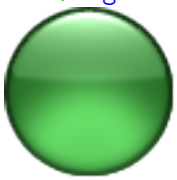
More complex shapes than “embossed and rounded” are possible with the full rendering API, which will be made public in a later release. Still, most of the simple icons (such as those in images/icons/arrow and images/icons/control) can be rendered using only bitmap-render-icon.

```
(icon-color->outline-color color) → (is-a?/c color%)
  color : (or/c string? (is-a?/c color%))
```

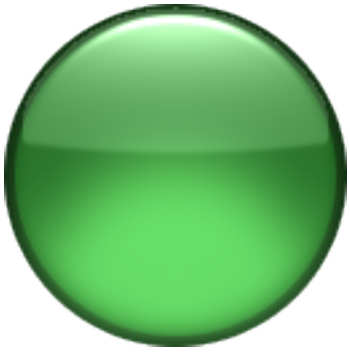
For a given icon color, returns the proper outline color%.

As an example, here is how to duplicate the record-icon using pict:

```
> (define outline-color (icon-color->outline-color "forestgreen"))
> (define brush-pict (colorize (filled-ellipse 62 62) "forestgreen"))
> (define pen-pict (linewidth 2 (colorize (ellipse 62 62) outline-
color)))
> (bitmap-render-icon
   (pict->bitmap (inset (cc-superimpose brush-pict pen-pict) 1))
   5/8 glass-icon-material)
```



```
> (record-icon #:color "forestgreen" #:height 64
               #:material glass-icon-material)
```



The outline width is usually (/ height 32) (in this case, 2), but not always. (For example, recycle-icon is an exception, as are parts of floppy-disk-icon.)

## 1.4 Arrow Icons

(require images/icons/arrow)  package: images-lib

Changed in version 1.1 of package `images-lib`: Added optional `#:backing-scale` arguments.

```
(right-arrow-icon  #:color color
                   [#:height height
                    #:material material
                    #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

```
(left-arrow-icon  #:color color
                  [#:height height
                   #:material material
                   #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

```
(up-arrow-icon  #:color color
                [#:height height
                 #:material material
                 #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

```
(down-arrow-icon  #:color color
                  [#:height height
                   #:material material
                   #:backing-scale backing-scale])
```

```
→ (is-a?/c bitmap%)
 color : (or/c string? (is-a?/c color%))
 height : (and/c rational? (>=/c 0)) = (default-icon-height)
 material : deep-flomap-material-value?
          = (default-icon-material)
 backing-scale : (and/c rational? (>/c 0.0))
               = (default-icon-backing-scale)
```

Standard directional arrows.

Example:

```
> (list (right-arrow-icon #:color syntax-icon-color
                          #:height (toolbar-icon-height))
        (left-arrow-icon #:color run-icon-color)
        (up-arrow-icon #:color halt-icon-color #:height 37)
        (down-arrow-icon #:color "lightblue" #:height 44
                         #:material glass-icon-material))
```

(list )

```
(right-over-arrow-icon  #:color color
                       [#:height height
                        #:material material
                        #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
 color : (or/c string? (is-a?/c color%))
 height : (and/c rational? (>=/c 0)) = (default-icon-height)
 material : deep-flomap-material-value?
          = (default-icon-material)
 backing-scale : (and/c rational? (>/c 0.0))
               = (default-icon-backing-scale)
```

```
(left-over-arrow-icon  #:color color
                      [#:height height
                       #:material material
                       #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
```

```
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

```
(right-under-arrow-icon  #:color color
                         [#:height height
                          #:material material
                          #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

```
(left-under-arrow-icon  #:color color
                        [#:height height
                         #:material material
                         #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Standard bent arrows.

Example:

```
> (list (right-over-arrow-icon #:color metal-icon-color
                               #:height (toolbar-icon-height))
        (left-over-arrow-icon #:color dark-metal-icon-color)
        (right-under-arrow-icon #:color run-icon-
color #:height 37)
        (left-under-arrow-icon #:color "lightgreen" #:height 44
                               #:material glass-icon-material))
```

(list )

## 1.5 Control Icons

```
(require images/icons/control)
```
package: `images-lib`

Changed in version 1.1 of package `images-lib`: Added optional `#:backing-scale` arguments.

```
(bar-icon  #:color color
           [#:height height
            #:material material
            #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (bar-icon #:color run-icon-color #:height 32)
```

This is not a “control” icon *per se*, but is used to make many others.

```
(play-icon  #:color color
            [#:height height
             #:material material
             #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
```

```
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (play-icon #:color run-icon-color #:height 32)
```

```
(back-icon  #:color color
            [#:height height
             #:material material
             #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

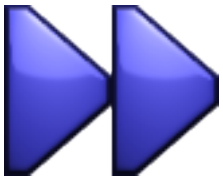
```
> (back-icon #:color run-icon-color #:height 32)
```

```
(fast-forward-icon  #:color color
                    [#:height height
                     #:material material
                     #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
```

```
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

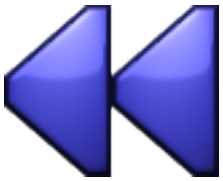
```
> (fast-forward-icon #:color syntax-icon-color #:height 32)
```



```
(rewind-icon  #:color color
              [#:height height
               #:material material
               #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

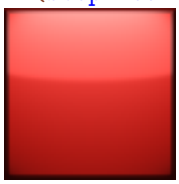
```
> (rewind-icon #:color syntax-icon-color #:height 32)
```



```
(stop-icon  #:color color
            [#:height height
             #:material material
             #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (stop-icon #:color halt-icon-color #:height 32)
```



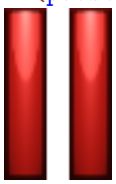
```
(record-icon  #:color color
              [#:height height
               #:material material
               #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (record-icon #:color "red" #:height 32)
```

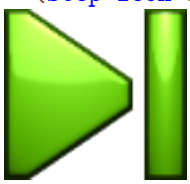
```
(pause-icon  #:color color
             [#:height height
              #:material material
              #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (pause-icon #:color halt-icon-color #:height 32)
```


```
(step-icon  #:color color
            [#:height height
             #:material material
             #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (step-icon #:color run-icon-color #:height 32)
```


```
(step-back-icon  #:color color
                 [#:height height
                  #:material material
                  #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (step-back-icon #:color run-icon-color #:height 32)
```

```
(continue-forward-icon  #:color color
                        [#:height height
                         #:material material
                         #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (continue-forward-icon #:color run-icon-color #:height 32)
```

```
(continue-backward-icon  #:color color
                         [#:height height
                          #:material material
                          #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

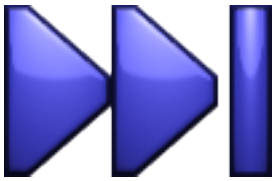
Example:

```
> (continue-backward-icon #:color run-icon-color #:height 32)
```

```
(search-forward-icon  #:color color
                      [#:height height
                       #:material material
                       #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

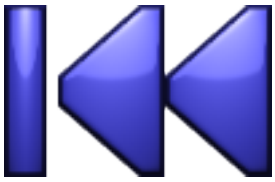
```
> (search-forward-icon #:color syntax-icon-color #:height 32)
```



```
(search-backward-icon  #:color color
                       [#:height height
                        #:material material
                        #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (search-backward-icon #:color syntax-icon-color #:height 32)
```



## 1.6 File Icons

```
(require images/icons/file)
```
package: images-lib

Changed in version 1.1 of package images-lib: Added optional #:backing-scale arguments.

```
(floppy-disk-icon [#:color color
                   #:height height
                   #:material material
                   #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = "slategray"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (floppy-disk-icon #:height 32 #:material glass-icon-material)
```

```
(save-icon [#:disk-color disk-color
            #:arrow-color arrow-color
            #:height height
            #:material material
            #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  disk-color : (or/c string? (is-a?/c color%)) = "gold"
  arrow-color : (or/c string? (is-a?/c color%))
              = syntax-icon-color
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
```

```
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (save-icon #:height 32)
```

```
(load-icon [#:disk-color disk-color
            #:arrow-color arrow-color
            #:height height
            #:material material
            #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  disk-color : (or/c string? (is-a?/c color%)) = "gold"
  arrow-color : (or/c string? (is-a?/c color%))
              = syntax-icon-color
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (load-icon #:height 32)
```

```
(small-save-icon [#:disk-color disk-color
                  #:arrow-color arrow-color
                  #:height height
                  #:material material
                  #:backing-scale backing-scale])
```

```
→ (is-a?/c bitmap%)
 disk-color : (or/c string? (is-a?/c color%)) = "gold"
 arrow-color : (or/c string? (is-a?/c color%))
             = syntax-icon-color
 height : (and/c rational? (>=/c 0)) = (default-icon-height)
 material : deep-flomap-material-value?
          = (default-icon-material)
 backing-scale : (and/c rational? (>/c 0.0))
               = (default-icon-backing-scale)
```

Example:

```
> (small-save-icon #:height 32)
```

```
(small-load-icon [#:disk-color disk-color
                  #:arrow-color arrow-color
                  #:height height
                  #:material material
                  #:backing-scale backing-scale])
→ (is-a?/c bitmap%)
 disk-color : (or/c string? (is-a?/c color%)) = "gold"
 arrow-color : (or/c string? (is-a?/c color%))
             = syntax-icon-color
 height : (and/c rational? (>=/c 0)) = (default-icon-height)
 material : deep-flomap-material-value?
          = (default-icon-material)
 backing-scale : (and/c rational? (>/c 0.0))
               = (default-icon-backing-scale)
```

Example:

```
> (small-load-icon #:height 32)
```

## 1.7 Symbol and Text Icons

```
(require images/icons/symbol)
```
package: images-lib

Changed in version 1.1 of package images-lib: Added optional #:backing-scale arguments.

```
(text-icon  str
            [font
             #:trim? trim?
             #:color color
             #:height height
             #:material material
             #:outline outline
             #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  str : string?
  font : (is-a?/c font%) = (make-font)
  trim? : boolean? = #t
  color : (or/c string? (is-a?/c color%)) = "white"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  outline : (and/c rational? (>=/c 0)) = (/ height 32)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Renders a text string as an icon. For example,

```
> (text-icon "An Important Point!"
             (make-font #:weight 'bold #:underlined? #t)
             #:color "lightskyblue" #:height 44)
```

The size and face of *font* are ignored. If *trim?* is #f, the drawn text is not cropped before rendering. Otherwise, it is cropped to the smallest rectangle containing all the non-zero-alpha pixels. Rendering very small glyphs shows the difference dramatically:

```
> (list (text-icon "." #:trim? #t)
        (text-icon "." #:trim? #f))
```

```
(list  )
```

Notice that both icons are `(default-icon-height)` pixels tall.

Because different platforms have different fonts, `text-icon` cannot guarantee the icons it returns have a consistent look or width across all platforms, or that any unicode characters in *str* will exist.

```
(recycle-icon [#:color color
               #:height height
               #:material material
               #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = "forestgreen"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Returns the universal recycling symbol, rendered as an icon.

Example:

```
> (recycle-icon #:height 48)
```

```
(x-icon [#:color color
         #:height height
         #:material material
         #:thickness thickness
         #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = halt-icon-color
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
```

```
  thickness : (and/c rational? (>=/c 0)) = 10
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Returns an “x” icon that is guaranteed to look the same on all platforms. (Anything similar that would be constructed by `text-icon` would differ at least slightly across platforms.)

Example:

```
> (x-icon #:height 32)
```

Changed in version 1.1 of package `images-lib`: Added optional `#:thickness` argument.

```
(check-icon [#:color color
             #:height height
             #:material material
             #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = run-icon-color
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (check-icon #:height 32)
```

```
(lambda-icon [#:color color
              #:height height
              #:material material
              #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
        = light-metal-icon-color
```

```
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (lambda-icon #:height 32 #:material metal-icon-material)
```

```
(hash-quote-icon [#:color color
                  #:height height
                  #:material material
                  #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = "mediumseagreen"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Examples:

```
> (require (only-in images/icons/tool macro-stepper-hash-color))
> (hash-quote-icon #:color macro-stepper-hash-color #:height 32)
```

## 1.8 Miscellaneous Icons

`(require images/icons/misc)` package: `images-lib`

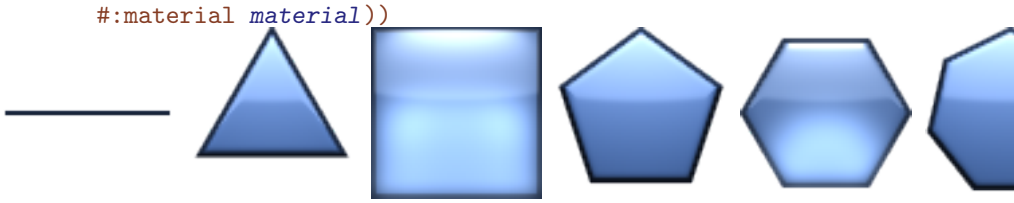
Changed in version 1.1 of package `images-lib`: Added optional `#:backing-scale` arguments.

```
(regular-polygon-icon  sides
                       [start]
                        #:color color
                       [#:height height
                        #:material material
                        #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  sides : exact-positive-integer?
  start : real? = (- (/ pi sides) (* 1/2 pi))
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Renders the largest regular polygon with *sides* sides, with the first vertex at angle *start*, that can be centered in a *height* × *height* box. The default *start* angle is chosen so that the polygon has a horizontal bottom edge.

Example:

```
> (for/list ([sides  (in-range 1 9)]
             [material  (in-cycle (list plastic-icon-material
                                        glass-icon-material))])
    (regular-polygon-icon sides #:color "cornflowerblue" #:height 32
                          #:material material))
```



```
(list
```

```
(stop-sign-icon [#:color color
                 #:height height
                 #:material material
                 #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = halt-icon-color
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (stop-sign-icon #:height 32 #:material glass-icon-material)
```

```
(stop-signs-icon [#:color color
                  #:height height
                  #:material material
                  #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = halt-icon-color
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (stop-signs-icon #:height 32 #:material plastic-icon-material)
```

```
(foot-icon  #:color color
            [#:height height
             #:material material
             #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%))
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (foot-icon #:color "chocolate" #:height 32
             #:material glass-icon-material)
```

```
(magnifying-glass-icon [#:frame-color frame-color
                        #:handle-color handle-color
                        #:height height
                        #:material material
                        #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  frame-color : (or/c string? (is-a?/c color%))
              = light-metal-icon-color
  handle-color : (or/c string? (is-a?/c color%)) = "brown"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (magnifying-glass-icon #:height 32)
```

```
(left-magnifying-glass-icon [#:frame-color frame-color
                             #:handle-color handle-color
                             #:height height
                             #:material material
                             #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  frame-color : (or/c string? (is-a?/c color%))
              = light-metal-icon-color
  handle-color : (or/c string? (is-a?/c color%)) = "brown"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
```

```
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (left-magnifying-glass-icon #:height 32)
```

```
(bomb-icon [#:cap-color cap-color
            #:bomb-color bomb-color
            #:height height
            #:material material
            #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  cap-color : (or/c string? (is-a?/c color%))
            = light-metal-icon-color
  bomb-color : (or/c string? (is-a?/c color%))
             = dark-metal-icon-color
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (bomb-icon #:height 48 #:material glass-icon-material)
```

```
(left-bomb-icon [#:cap-color cap-color
                 #:bomb-color bomb-color
                 #:height height
                 #:material material
                 #:backing-scale backing-scale])
```

```
→ (is-a?/c bitmap%)
 cap-color : (or/c string? (is-a?/c color%))
           = light-metal-icon-color
 bomb-color : (or/c string? (is-a?/c color%))
            = dark-metal-icon-color
 height : (and/c rational? (>=/c 0)) = (default-icon-height)
 material : deep-flomap-material-value?
          = (default-icon-material)
 backing-scale : (and/c rational? (>/c 0.0))
               = (default-icon-backing-scale)
```

Example:

```
> (left-bomb-icon #:height 48)
```

```
(clock-icon [hours
             minutes
             #:face-color face-color
             #:hand-color hand-color
             #:height height
             #:backing-scale backing-scale]) → (is-a?/c bitmap%)
 hours : (integer-in 0 11) = 0
 minutes : (real-in 0 60) = 47
 face-color : (or/c string? (is-a?/c color%))
            = light-metal-icon-color
 hand-color : (or/c string? (is-a?/c color%)) = "firebrick"
 height : (and/c rational? (>=/c 0)) = (default-icon-height)
 backing-scale : (and/c rational? (>/c 0.0))
               = (default-icon-backing-scale)
```

Examples:

```
> (clock-icon #:height 96)
```

```
> (clock-icon 3 21 #:height 48
              #:face-color "lightblue"
              #:hand-color "darkblue")
```

```
(stopwatch-icon [hours
                 minutes
                 #:face-color face-color
                 #:hand-color hand-color
                 #:height height
                 #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  hours : (integer-in 0 11) = 0
  minutes : (real-in 0 60) = 47
  face-color : (or/c string? (is-a?/c color%))
             = light-metal-icon-color
  hand-color : (or/c string? (is-a?/c color%)) = "firebrick"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (stopwatch-icon #:height 96)
```

```
(stethoscope-icon [#:color color
                   #:height height
                   #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = "black"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

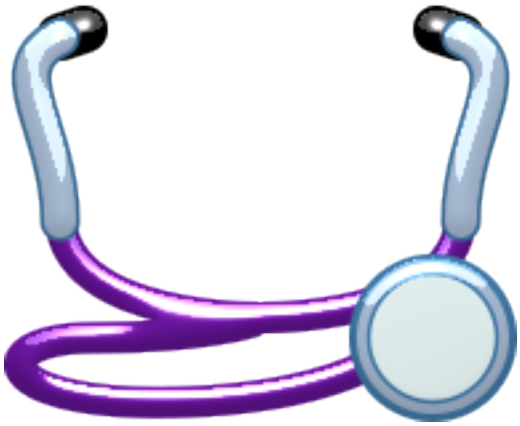
Example:

```
> (stethoscope-icon #:height 96)
```

```
(short-stethoscope-icon [#:color color
                         #:height height
                         #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = "black"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (short-stethoscope-icon #:color "purple" #:height 96)
```



```
(lock-icon [open?
            #:body-color body-color
            #:shackle-color shackle-color
            #:height height
            #:material material
            #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  open? : boolean? = #f
  body-color : (or/c string? (is-a?/c color%)) = "orange"
  shackle-color : (or/c string? (is-a?/c color%))
                = light-metal-icon-color
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Examples:

```
> (lock-icon #:height 32)
```

```
> (lock-icon #t #:height 48
             #:body-color "navajowhite"
             #:shackle-color "lemonchiffon"
             #:material glass-icon-material)
```

```
(close-icon [#:color color
             #:height height
             #:material material
             #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  color : (or/c string? (is-a?/c color%)) = "black"
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Example:

```
> (close-icon #:height 32 #:material glass-icon-material)
```

Added in version 1.1 of package `images-lib`.

## 1.9 Stickman Icons

```
(require images/icons/stickman)
```
package: images-lib

Changed in version 1.1 of package images-lib: Added optional #:backing-scale arguments.

```
(standing-stickman-icon [#:body-color body-color
                         #:arm-color arm-color
                         #:head-color head-color
                         #:height height
                         #:material material
                         #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  body-color : (or/c string? (is-a?/c color%)) = run-icon-color
  arm-color : (or/c string? (is-a?/c color%)) = "white"
  head-color : (or/c string? (is-a?/c color%)) = run-icon-color
  height : (and/c rational? (>=/c 0)) = (default-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Returns the icon displayed in DrRacket's lower-right corner when no program is running.

Example:

```
> (standing-stickman-icon #:height 64)
```

```
(running-stickman-icon  t
                        [#:body-color body-color
                         #:arm-color arm-color
                         #:head-color head-color
                         #:height height
                         #:material material
                         #:backing-scale backing-scale])
```

```
→ (is-a?/c bitmap%)
 t : rational?
 body-color : (or/c string? (is-a?/c color%)) = run-icon-color
 arm-color : (or/c string? (is-a?/c color%)) = "white"
 head-color : (or/c string? (is-a?/c color%)) = run-icon-color
 height : (and/c rational? (>=/c 0)) = (default-icon-height)
 material : deep-flomap-material-value?
          = (default-icon-material)
 backing-scale : (and/c rational? (>/c 0.0))
               = (default-icon-backing-scale)
```

Returns a frame of the icon animated in DrRacket's lower-right corner when a program is running. The frame returned is for time `t` of a run cycle with a one-second period.

The following example samples the run cycle at 12 Hz, or every `1/12` second:

```
> (for/list ([t  (in-range 0 1 1/12)])
    (running-stickman-icon t #:height 32))
(list
```

The stickman's joint angles are defined by continuous periodic functions, so the run cycle can be sampled at any resolution, or at any real-valued time `t`. The cycle is modeled after the run cycle of the player's avatar in the Commodore 64 game Impossible Mission.

## 1.10 Tool Icons

```
(require images/icons/tool)
```
package: images-lib

Changed in version 1.1 of package `images-lib`: Added optional `#:backing-scale` arguments.

```
(check-syntax-icon [#:height height
                    #:material material
                    #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = (toolbar-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

```
(small-check-syntax-icon [#:height height
                          #:material material
                          #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = (toolbar-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Icons for Check Syntax. The `small-check-syntax-icon` is used when the toolbar is on the side.

Example:

```
> (list (check-syntax-icon #:height 32)
        (small-check-syntax-icon #:height 32))
(list )
```

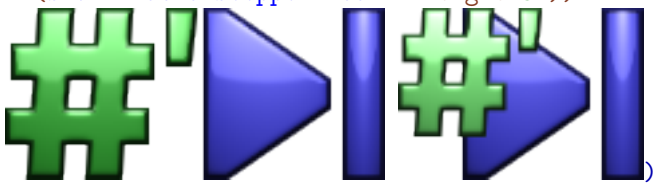
```
(macro-stepper-icon [#:height height
                     #:material material
                     #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = (toolbar-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

```
(small-macro-stepper-icon [#:height height
                           #:material material
                           #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = (toolbar-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Icons for the Macro Stepper. The `small-macro-stepper-icon` is used when the toolbar is on the side.

Example:

```
> (list (macro-stepper-icon #:height 32)
        (small-macro-stepper-icon #:height 32))
```



```
(list )
```

```
(debugger-icon [#:height height
                #:material material
                #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = (toolbar-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

```
(small-debugger-icon [#:height height
                      #:material material
                      #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = (toolbar-icon-height)
  material : deep-flomap-material-value?
           = (default-icon-material)
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Icons for the Debugger. The `small-debugger-icon` is used when the toolbar is on the side.

Example:

```
> (list (debugger-icon #:height 32)
        (small-debugger-icon #:height 32))
```

```
(list )
```

```
debugger-bomb-color : (or/c string? (is-a?/c color%))
 = (make-object color% 128 32 32)
```

```
macro-stepper-hash-color : (or/c string? (is-a?/c color%))
 = (make-object color% 60 192 60)
```

```
small-macro-stepper-hash-color : (or/c string? (is-a?/c color%))
 = (make-object color% 128 255 128)
```

Constants used within `images/icons/tool`.

# 2 Logos

```
(require images/logos)
```
package: images-lib

```
(plt-logo [#:height height
           #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = 256
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Returns the PLT logo, rendered in tinted glass and azure metal by the ray tracer that renders icons.

Example:

```
> (plt-logo)
```

The default height is the size used for DrRacket splash screen.

```
(planet-logo [#:height height
              #:backing-scale backing-scale]) → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = 96
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Returns an unofficial PLaneT logo. This is used as the PLaneT icon when DrRacket downloads PLaneT packages.

Examples:

```
> (planet-logo)
```

```
> (planet-logo #:height (default-icon-height))
```

```
(stepper-logo [#:height height
               #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = 96
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Returns the algebraic stepper logo.

Example:

```
> (stepper-logo)
```

```
(macro-stepper-logo [#:height height
                     #:backing-scale backing-scale])
 → (is-a?/c bitmap%)
  height : (and/c rational? (>=/c 0)) = 96
  backing-scale : (and/c rational? (>/c 0.0))
                = (default-icon-backing-scale)
```

Returns the macro stepper logo.

Example:

```
> (macro-stepper-logo)
```

# 3 Embedding Bitmaps in Compiled Files

```
(require images/compile-time)
```
package: images-lib

Producing computed bitmaps can take time. To reduce the startup time of programs that use computed bitmaps, use the macros exported by `images/compile-time` to *compile* them: to embed the computed bitmaps in fully expanded, compiled modules.

This is a form of constant folding, or equivalently a form of *safe* "3D" values.

The macros defined here compute bitmaps at expansion time, and expand to the bitmap's `bytes` and a simple wrapper that converts `bytes` to a `bitmap%`. Thus, fully expanded, compiled modules contain (more or less) literal bitmap values, which do not need to be computed again when the module is `required` by another.

The literal bitmap values are encoded in PNG or JPEG format, so they are compressed in the compiled module.

To get the most from compiled bitmaps during development, it is best to put them in files that are changed infrequently. For example, for games, we suggest having a separate module called something like `images.rkt` or `resources.rkt` that `provides` all the game's images.

```
(compiled-bitmap expr [quality])

  expr : (is-a?/c bitmap%)
  quality : (integer-in 0 100)
```

Evaluates *expr* at expansion time, which must return a `bitmap%`, and returns to the bitmap at run time. Keep in mind that *expr* has access only to expansion-time values, not run-time values.

If *quality* is `100`, the bitmap is stored as a PNG. If *quality* is between `0` and `99` inclusive, it is stored as a JPEG with quality *quality*. (See `save-file`.) If the bitmap has an alpha channel, its alpha channel is stored as a separate JPEG. The default value is `100`.

Generally, to use this macro, wrap a `bitmap%`-producing expression with it and move any identifiers it depends on into the expansion phase. For example, suppose we are computing a large PLT logo at run time:

```
#lang racket

(require images/logos)

(define the-logo (plt-logo #:height 384))
```

Running this takes several seconds. To move the cost to expansion time, we change the program to

```
#lang racket

(require images/compile-time
         (for-syntax images/logos))

(define the-logo (compiled-bitmap (plt-logo #:height 384)))
```

The logo will not change, but now *expanding* the program takes several seconds, and *running* it takes a few milliseconds. Note that `images/logos` is now required `for-syntax`, so that the expansion-phase expression `(plt-logo #:height 384)` has access to the identifier `plt-logo`.

```
(compiled-bitmap-list expr [quality])

  expr : (listof (is-a?/c bitmap%))
  quality : (integer-in 0 100)
```

Like `compiled-bitmap`, but it expects *expr* to return a `list` of `bitmap%`s, and it returns the list at run time. The *quality* argument works as in `compiled-bitmap`, but is applied to all the images in the list.

Use this for animations. For example,

```
#lang racket


(require images/compile-time
         (for-syntax images/icons/stickman))

(begin-for-syntax
  (define num-stickman-frames 12))

(define running-stickman-frames
  (compiled-bitmap-list
   (for/list ([t  (in-range 0 1 (/ 1 num-stickman-frames))])
     (running-stickman-icon t #:height 32
                            #:body-color "red"
                            #:arm-color "white"
                            #:head-color "red"))
   50))
```

This computes

```
> running-stickman-frames
```

```
(list
```

at expansion time.

# 4 Floating-Point Bitmaps

```
(require images/flomap)
```
package: images-lib

The images/flomap module provides the struct type flomap, whose instances represent floating-point bitmaps with any number of color components. It also provides purely functional operations on flomaps for compositing, pointwise floating-point math, blur, gradient calculation, arbitrary spatial transforms (such as rotation), and conversion to and from bitmap% instances.

**This is a Typed Racket module.** Its exports can generally be used from untyped code with negligible performance loss over typed code. Exceptions are documented **in bold text**. Most exceptions are macros used to inline floating-point operations.

The following flomap fm is used in various examples:

```
> (define fm
    (draw-flomap
     (λ (fm-dc)
       (send fm-dc set-alpha 0)
       (send fm-dc set-background "black")
       (send fm-dc clear)
       (send fm-dc set-alpha 1/3)
       (send fm-dc translate 2 2)
       (send fm-dc set-pen "black" 4 'long-dash)
       (send fm-dc set-brush "red" 'solid)
       (send fm-dc draw-ellipse 0 0 192 192)
       (send fm-dc set-brush "green" 'solid)
       (send fm-dc draw-ellipse 64 0 192 192)
       (send fm-dc set-brush "blue" 'solid)
       (send fm-dc draw-ellipse 32 44 192 192))
     260 240))
> (flomap->bitmap fm)
```

It is typical to use `flomap->bitmap` to visualize a flomap at the REPL.

Contents:

## 4.1 Overview

Contents:

### 4.1.1 Motivation

There are three main reasons to use flomaps:

- **Precision.** A point in a typical bitmap is represented by a few bytes, each of which can have one of 256 distinct values. In contrast, a point in a flomap is represented by double-precision floating-point numbers, typically between `0.0` and `1.0` inclusive. This range contains about 4.6 *quintillion* (or $4.6 \times 10^{18}$) distinct values. While bytes are fine for many applications, their low precision becomes a problem when images are repeatedly operated on, or when their values are built by adding many small amounts—which are often rounded to zero.

- **Range.** A floating-point value can also represent about 4.6 quintillion distinct intensities above saturation (`1.0`). If distinguishing oversaturated values is important, flomaps have the range for it. Further, floating-point images are closed under pointwise arithmetic (up to floating-point error).
- **Speed.** The `images/flomap` module benefits greatly from Typed Racket's type-directed optimizations. Even getting individual color values—interpolated between points, if desired—is fast.

For these reasons, other parts of the `images` library use flomaps internally, to represent and operate on RGB and ARGB images, light maps, shadow maps, height maps, and normal maps.

### 4.1.2 Conceptual Model

A flomap is conceptually infinite in its width and height, but has nonzero values in a finite rectangle starting at coordinate `0 0` and extending to its width and height (exclusive). A flomap is **not** conceptually infinite in its components because there is no natural linear order on component coordinates, as the meaning of components depends on programmer intent.

The following example creates a $10 \times 10$ bitmap with RGB components, and indexes its top-left red value and two values outside the finite, nonzero rectangle. It also attempts to index component `3`, which doesn't exist. Note that `flomap-ref` accepts its coordinate arguments in a standard order: `k x y` (with `k` for **k**omponent).

```
> (define magenta-fm (make-flomap* 10 10 #(0.5 0.0 1.0)))
> (flomap->bitmap magenta-fm)
```

```
> (flomap-ref* magenta-fm 0 0)
(flvector 0.5 0.0 1.0)
> (flomap-ref* magenta-fm -1 0)
(flvector 0.0 0.0 0.0)
> (flomap-ref* magenta-fm 0 1000)
(flvector 0.0 0.0 0.0)
> (flomap-ref magenta-fm 3 0 0)
flomap-ref: expected argument of type <nonnegative fixnum <
3>; given: 3
```

Many flomap functions, such as `flomap-bilinear-ref` and `flomap-rotate`, treat their arguments as if every *real* `x y` coordinate has values. In all such cases, known values are at half-integer coordinates and others are interpolated.

Examples:

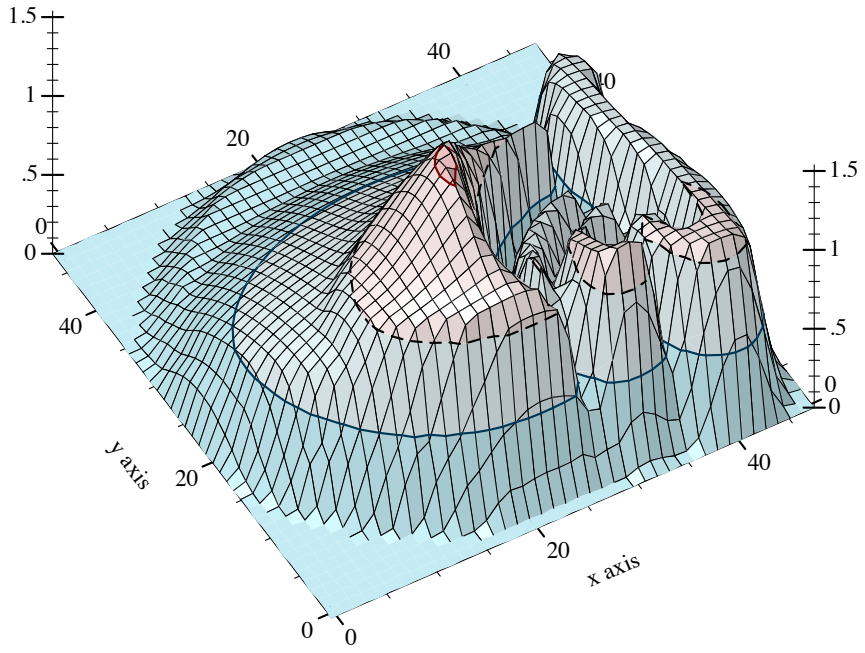
```
> (flomap-bilinear-ref* magenta-fm 0.5 0.5)
```

```
(flvector 0.5 0.0 1.0)
> (flomap-bilinear-ref* magenta-fm 0.25 0.25)
(flvector 0.28125 0.0 0.5625)
> (flomap-bilinear-ref* magenta-fm 0.0 0.0)
(flvector 0.125 0.0 0.25)
> (flomap-bilinear-ref* magenta-fm -0.25 -0.25)
(flvector 0.03125 0.0 0.0625)
```

This conceptual model allows us to treat flomaps as if they were multi-valued functions on `Real`×`Real`. For example, we might plot the red component of an ARGB icon:

```
> (require images/icons/misc plot)
> (define icon-fm (bomb-flomap #:bomb-color "orange" #:height 48))
> (flomap->bitmap icon-fm)
```

```
> (define-values (icon-width icon-height) (flomap-size icon-fm))
> (plot3d (contour-intervals3d
           (λ (x y) (flomap-bilinear-ref icon-fm 1 x y))
           -0.5 (+ 0.5 icon-width) -0.5 (+ 0.5 icon-height)))
```




Notice that the plot's maximum height is above saturation (`1.0`). The tallest peak corresponds to the specular highlight (the shiny part) on the bomb. Specular highlights are one case where it is important to operate on oversaturated values without truncating them—until it is time to display the image.

If we have a `w`×`h` flomap and consider its known values as being at half-integer coordinates, the exact center of the flomap is at `(* 1/2 w) (* 1/2 h)`. When unknown values are estimated using bilinear interpolation, the finite rectangle containing all the known *and estimated* nonzero values is from `-1/2 -1/2` to `(+ w 1/2) (+ h 1/2)`.

### 4.1.3 Opacity (Alpha Components)

A partially transparent flomap is simply a flomap in which component `0` is assumed to be an alpha (opacity) component. The other components should be multiplied by their corresponding alpha value; i.e. an RGB triple `1.0 0.5 0.25` with opacity `0.5` should be represented by `0.5 0.5 0.25 0.125`.

This representation generally goes by the unfortunate misnomer "premultiplied alpha," which makes it seem as if the *alpha* component is multiplied by something.

We will refer to this representation as *alpha-multiplied* because the color components are multiplied by the alpha component. All alpha-aware functions consume alpha-multiplied flomaps and produce alpha-multiplied flomaps.

There are many good reasons to use alpha-multiplied flomaps instead of non-alpha-multiplied flomaps. Some are:

- Compositing requires fewer operations per point.
- Compositing is associative; i.e. `(flomap-lt-superimpose fm1 (flomap-lt-superimpose fm2 fm3))` is the same as `(flomap-lt-superimpose (flomap-lt-superimpose fm1 fm2) fm3)`, up to floating-point error.
- There is only one transparent point: all zeros. We could not conceptualize partially transparent flomaps as being infinite in size without a unique transparent point.
- Many functions can operate on flomaps without treating the alpha component specially and still be correct.

As an example of the last point, consider blur. The following example creates an alpha-multiplied flomap using `draw-flomap`. It blurs the flomap using a general-purpose (i.e. non-alpha-aware) blur function, then converts the flomap to non-alpha-multiplied and does the same.

```
> (define circle-fm (draw-flomap (λ (fm-dc)
                                   (send fm-dc set-
pen "black" 1 'transparent)
                                   (send fm-dc set-
brush "green" 'solid)
                                   (send fm-dc draw-
ellipse 10 10 30 30))
                                 50 50))
> (flomap->bitmap circle-fm)
```

```
> (flomap->bitmap (flomap-blur circle-fm 4 4))
```

```
> (let* ([fm  (flomap-divide-alpha circle-fm)]
         [fm  (flomap-blur fm 4 4)]
         [fm  (flomap-multiply-alpha fm)])
```

```
(flomap->bitmap fm))
```

Notice the dark band around the second blurred circle.

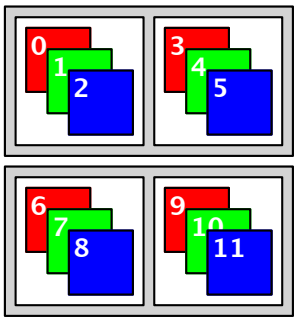
Of course, this could be fixed by making `flomap-blur` operate differently on flomaps with an alpha component. But the implementation would amount to converting them to alpha-multiplied flomaps anyway.

The only valid reason to not multiply color components by alpha is loss of precision, which is not an issue with flomaps.

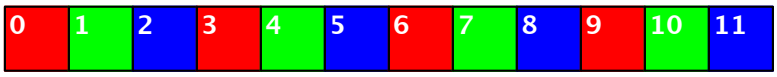
### 4.1.4 Data Layout

For most applications, there should be enough flomap functions available that you should not need to access their fields directly. However, there will always be use cases for direct manipulation, so the fields are public.

The color values in a flomap are stored flattened in a single `FlVector`, in row-major order with adjacent color components. For example, a $2\times2$ RGB flomap can be visualized as




In a flomap, it would be stored as




Mathematically, for a `c`-component, `w`-width flomap, the `k`th color component at position `x y` is at index

```
(+ k (* c (+ x (* y w))))
```

The `coords->index` function carries out this calculation quickly using only fixnum arithmetic.

If `i` is a calculated index for the value at `k x y`, then the `(+ k 1)`th value is at index `(+ i 1)`, the `(+ x 1)`th value is at index `(+ i c)`, and the `(+ y 1)`th value is at index `(+ i (* c w))`.

## 4.2 Struct Type and Accessors

```
(struct flomap (values components width height))
  values : FlVector
  components : Integer
  width : Integer
  height : Integer
```

Represents a `width`×`height` floating-point bitmap with `components` color components. The `values` vector contains the flattened image data (see §4.1.4 “Data Layout”).

A guard ensures that the `values` field has length `(* components width height)`, and that each size field is a nonnegative fixnum.

Examples:

```
> (require racket/flonum)
> (flomap (flvector 0.0 0.0 0.0 0.0) 4 1 1)
(flomap (flvector 0.0 0.0 0.0 0.0) 4 1 1)
> (flomap (flvector) 0 0 0)
(flomap (flvector) 0 0 0)
> (flomap (flvector 0.0) 2 1 1)
flomap: expected flvector of length 2; given one of length 1
```

The default `flomap` constructor is perhaps the hardest to use. Instead, to construct a `flomap` from scratch, you should generally use `make-flomap`, `make-flomap*`, `build-flomap` or `draw-flomap`.

```
(flomap-size fm) → Nonnegative-Fixnum Nonnegative-Fixnum
  fm : flomap
```

Returns the width and height of *fm* as nonnegative fixnums.

```
(flomap-ref fm k x y) → Float
  fm : flomap
  k : Integer
  x : Integer
  y : Integer
```

Returns *fm*'s value at *k x y*.

If *x* or *y* is out of bounds, this function returns 0.0. If *k* is out of bounds, it raises an error. The §4.1.2 "Conceptual Model" section explains why *k* is treated differently.

```
(flomap-ref* fm x y) → FlVector
  fm : flomap
  x : Integer
  y : Integer
```

Returns *fm*'s component values at *x y* as an flvector.

If *x* or *y* is out of bounds, this function returns an flvector filled with 0.0. It always returns an flvector of length (flomap-components *fm*).

```
(flomap-bilinear-ref fm k x y) → Float
  fm : flomap
  k : Integer
  x : Real
  y : Real
```

Returns an estimated value at any given *k x y* coordinate, calculated from known values in *fm*.

Like all other flomap functions that operate on real-valued coordinates, flomap-bilinear-ref regards known values as being at half-integer coordinates. Mathematically, if *x* = (+ i 0.5) and *y* = (+ j 0.5) for any integers i and j, then (flomap-bilinear-ref *fm k x y*) = (flomap-ref *fm k* i j).

Suppose *fm* is size w×h. If *x* ⩽ -0.5 or *x* ⩾ (+ w 0.5), this function returns 0.0; similarly for *y* and h. If *k* is out of bounds, it raises an error. The §4.1.2 "Conceptual Model" section explains why *k* is treated differently.

```
(flomap-bilinear-ref* fm x y) → FlVector
  fm : flomap
  x : Real
  y : Real
```

Like flomap-bilinear-ref, but returns an flvector containing estimates of all the components at *x y*.

```
(flomap-min-value fm) → Float
  fm : flomap
```

```
(flomap-max-value fm) → Float
  fm : flomap
```

These return the minimum and maximum values in *fm*.

```
(flomap-extreme-values fm) → Float Float
  fm : flomap
```

Equivalent to `(values (flomap-min-value fm) (flomap-max-value fm))`, but faster.

```
(flomap-nonzero-rect fm) → Nonnegative-Fixnum
                           Nonnegative-Fixnum
                           Nonnegative-Fixnum
                           Nonnegative-Fixnum
  fm : flomap
```

Returns the smallest rectangle containing every nonzero value (in any component) in *fm*. The values returned are *x* minimum, *y* minimum, *x* maximum + 1, and *y* maximum + 1.

The values returned by `flomap-nonzero-rect` can be sent to `subflomap` to trim away zero values. But see `flomap-trim`, which is faster for alpha-multiplied flomaps.

```
(coords->index c w k x y) → Fixnum
  c : Integer
  w : Integer
  k : Integer
  x : Integer
  y : Integer
```

Returns the index of the value at coordinates *k x y* of a flomap with *c* color components and width *w*. This function does not check any coordinates against their bounds.

```
(unsafe-flomap-ref vs c w h k x y) → Float
  vs : FlVector
  c : Integer
  w : Integer
  h : Integer
  k : Integer
  x : Integer
  y : Integer
```

If `fm` = `(flomap vs c w h)`, returns `fm`'s value at *k x y*. If *x* or *y* is out of bounds, this returns `0.0`. It is unsafe because *k* is unchecked, as well as indexing into *vs*.

This function is used by some library functions, such as `flomap-bilinear-ref`, to index into already-destructured flomaps. From untyped code, applying this function is likely no faster than applying `flomap-ref`, because of extra contract checks.

```
(unsafe-flomap-ref* vs c w h x y) → FlVector
  vs : FlVector
  c : Integer
  w : Integer
  h : Integer
  x : Integer
  y : Integer
```

Like `unsafe-flomap-ref`, but returns an flvector containing all the component values at *x* *y*.

## 4.3 Conversion and Construction

```
(flomap->bitmap fm
                #:backing-scale backing-scale) → Any
  fm : flomap
  backing-scale : Positive-Real
```

Converts a flomap to a `bitmap%`.

The return type is imprecise because Typed Racket does not support the object system well yet. As a typed function, this is most useful in DrRacket's REPL to visualize flomaps; any other typed use is difficult.

Flomaps are interpreted differently depending on the number of components:

- **Zero components.** Raises an error.
- **One component.** Interpreted as intensity (grayscale).
- **Two components.** Interpreted as AL, or alpha+intensity, with intensity multiplied by alpha.
- **Three components.** Interpreted as RGB.
- **Four components.** Interpreted as ARGB with color components multiplied by alpha.
- **More components.** Raises an error.

See §4.1.3 "Opacity (Alpha Components)" for a discussion of opacity (alpha) representation.

A zero-size *fm* is padded by one point in any zero direction before conversion. For example, if *fm* is size $0 \times 1$, the result of `(flomap->bitmap fm)` is size $1 \times 1$.

Values are clamped to between `0.0` and `1.0` before conversion.

```
(bitmap->flomap bm #:unscaled? unscaled?) → flomap
  bm : Any
  unscaled? : Any
```

Given a bitmap% instance *bm*, returns an ARGB flomap with alpha-multiplied color components. See §4.1.3 "Opacity (Alpha Components)" for a discussion of opacity (alpha) representation.

If *unscaled?* is true, the flomap is converted from the actual bitmap backing *bm* rather than a scaled version. See the #:unscaled? keyword parameter of get-argb-pixels for more information.

The argument type is imprecise because Typed Racket does not support the object system well yet.

```
(make-flomap c w h [v]) → flomap
  c : Integer
  w : Integer
  h : Integer
  v : Real = 0.0
```

Returns a $w \times h$ flomap with *c* components, with every value initialized to *v*. Analogous to make-vector.

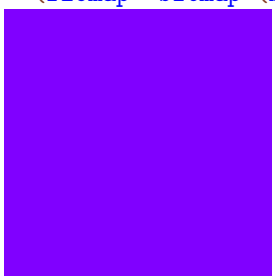
To create flomaps filled with a solid color, use make-flomap*.

```
(make-flomap* w h vs) → flomap
  w : Integer
  h : Integer
  vs : (U (Vectorof Real) FlVector)
```

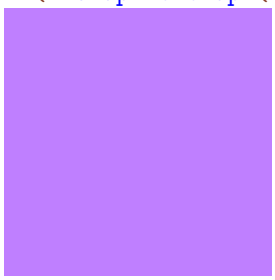
Returns a $w \times h$ flomap with each point's components initialized using the values in *vs*. Analogous to make-vector.

The following two examples create an RGB and an ARGB flomap:

```
> (flomap->bitmap (make-flomap* 100 100 #(0.5 0.0 1.0)))
```



```
> (flomap->bitmap (make-flomap* 100 100 #(0.5 0.25 0.0 0.5)))
```



See §4.1.3 “Opacity (Alpha Components)” for a discussion of opacity (alpha) representation.

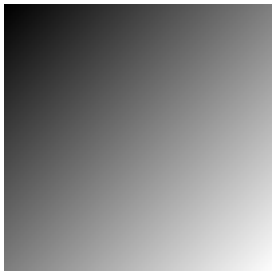
```
(build-flomap c w h f) → flomap
  c : Integer
  w : Integer
  h : Integer
  f : (Nonnegative-Fixnum Nonnegative-Fixnum Nonnegative-Fixnum -> Real)
```

Returns a $w \times h$ flomap with $c$ color components, with values defined by $f$. Analogous to `build-vector`.

The function $f$ receives three arguments `k x y`: the color component and two positional coordinates.

Examples:

```
> (flomap->bitmap
   (build-flomap 1 100 100
                 (λ (k x y) (/ (+ x y) 200))))
```



```
> (define sine-fm
    (build-flomap
     1 100 100
     (λ (k x y)
       (* 1/2 (+ 1 (sin (sqrt (+ (sqr (- x 50))
                                 (sqr (- y 50)))))))))))
> (flomap->bitmap sine-fm)
```

To build a flomap using a function that returns vectors, see `build-flomap*`.

```
(build-flomap* c w h f) → flomap
  c : Integer
  w : Integer
  h : Integer
  f : (Nonnegative-Fixnum Nonnegative-Fixnum
       -> (U (Vectorof Real) FlVector))
```

Returns a $w \times h$ flomap with $c$ color components. Its values are defined by $f$, which returns vectors of point components. The vectors returned by $f$ must be length $c$.

Analogous to `build-vector`.

Examples:

```
> (flomap->bitmap
   (build-flomap* 4 100 100
                  (λ (x y)
                    (vector (/ (+ x y) 200)
                            (/ (+ (- 100 x) y) 200)
                            (/ (+ (- 100 x) (- 100 y)) 200)
                            (/ (+ x (- 100 y)) 200)))))
```

```
> (build-flomap* 4 100 100
                 (λ (x y) (vector (/ (+ x y) 200))))
build-flomap*: expected argument of type <length-4 Vector or
FlVector>; given: '#(0)
```

```
(draw-flomap draw w h) → flomap
  draw : (Any -> Any)
  w : Integer
  h : Integer
```

Returns a $w \times h$ bitmap drawn by *draw*. Analogous to slideshow's dc.

The *draw* function should accept a dc<%> instance and use its drawing methods to draw on an underlying bitmap. The bitmap is converted to a flomap using bitmap->flomap and returned. See §4 "Floating-Point Bitmaps" for an example.

This function is very difficult to use in Typed Racket, requiring occurrence checks for, and use of, experimental types. However, as Typed Racket grows to handle Racket's object system, the types will be made more precise.

```
(flomap-multiply-alpha fm) → flomap
  fm : flomap
```

```
(flomap-divide-alpha fm) → flomap
  fm : flomap
```

Multiplies/divides each nonzero-component value with the corresponding zero-component value. Dividing by 0.0 produces 0.0.

In other words, flomap-multiply-alpha converts non-alpha-multiplied flomaps into alpha-multiplied flomaps, and flomap-divide-alpha converts them back.

You should not generally have to use these functions, because bitmap->flomap returns an alpha-multiplied flomap and every alpha-aware flomap function assumes its arguments are alpha-multiplied and produces alpha-multiplied flomaps.

See §4.1.3 "Opacity (Alpha Components)" for a discussion of opacity (alpha) representation.

```
(inline-build-flomap c w h f)

  c : Integer
  w : Integer
  h : Integer
  f : (Nonnegative-Fixnum Nonnegative-Fixnum Nonnegative-Fixnum
       Nonnegative-Fixnum -> Float)
```

A macro version of build-flomap.

There are three differences between the function `f` passed to `build-flomap` and the `f` passed to `inline-build-flomap`. First, the `f` passed to `inline-build-flomap` can be a macro. Second, it receives arguments `k x y i`, where `i` is a precalculated index into the result's `values`. Third, it must return a `Float`.

Using `inline-build-flomap` instead of `build-flomap` may ensure that `f` is inlined, and therefore floats remain unboxed. Many library functions use `inline-build-flomap` internally for speed, notably `fm+` and the other pointwise arithmetic operators.

**This is not available in untyped Racket.**

```
(inline-build-flomap* c w h f)

  c : Integer
  w : Integer
  h : Integer
  f : (Nonnegative-Fixnum Nonnegative-Fixnum
       Nonnegative-Fixnum -> FlVector)
```

A macro version of `build-flomap*`.

There are three differences between the function `f` passed to `build-flomap*` and the `f` passed to `inline-build-flomap*`. First, the `f` passed to `inline-build-flomap*` can be a macro. Second, it receives arguments `x y i`, where `i` is a precalculated index into the result's `values`. Third, it must return a `FlVector`.

**This is not available in untyped Racket.**

## 4.4 Component Operations

```
(flomap-ref-component fm k) → flomap
  fm : flomap
  k : Integer
```

Extracts one component of a flomap and returns it as a new flomap. Raises an error if `k` is out of bounds.

Use this, for example, to extract the A and R components from an ARGB flomap:

```
> (flomap->bitmap (flomap-ref-component fm 0))
```

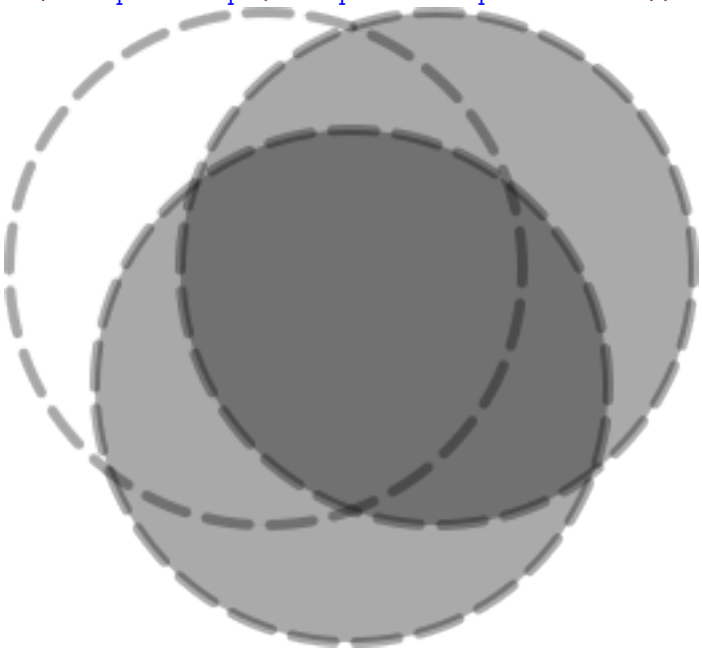
```
> (flomap->bitmap (flomap-ref-component fm 1))
```

```
(flomap-take-components fm k) → flomap
  fm : flomap
  k : Integer
```

Extracts the first *k* components and returns them as a new flomap. Raises an error if *k* is out of bounds.

Example:

```
> (flomap->bitmap (flomap-take-components fm 2))
```



```
(flomap-drop-components fm k) → flomap
  fm : flomap
  k : Integer
```

Extracts all but the first *k* components and returns them as a new flomap. Raises an error if *k* is out of bounds.

Use this, for example, to operate on only the RGB channels of an ARGB flomap:

```
> (flomap->bitmap
   (flomap-append-components (flomap-take-components fm 1)
                             (fm* 0.25 (flomap-drop-
components fm 1))))
```

```
(flomap-append-components fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

Appends the components of the given flomaps pointwise. Raises an error if not all flomaps are the same width and height.

Examples:

```
> (equal? fm (flomap-append-components (flomap-take-
components fm 2)
                                       (flomap-drop-
components fm 2)))
#t
> (flomap-append-components (make-flomap 1 10 10)
                            (make-flomap 3 20 20))
flomap-append-components: expected same-size flomaps; given
sizes 10×10 and 20×20
```

This function could behave according to the §4.1.2 “Conceptual Model”—that is, expand the smaller ones to the largest size before appending. However, appending the components of two different-size flomaps almost always indicates a logic or design error. If it really is intended, use `flomap-inset` or `subflomap` to expand the smaller flomaps manually, with more control over the expansion.

## 4.5 Pointwise Operations

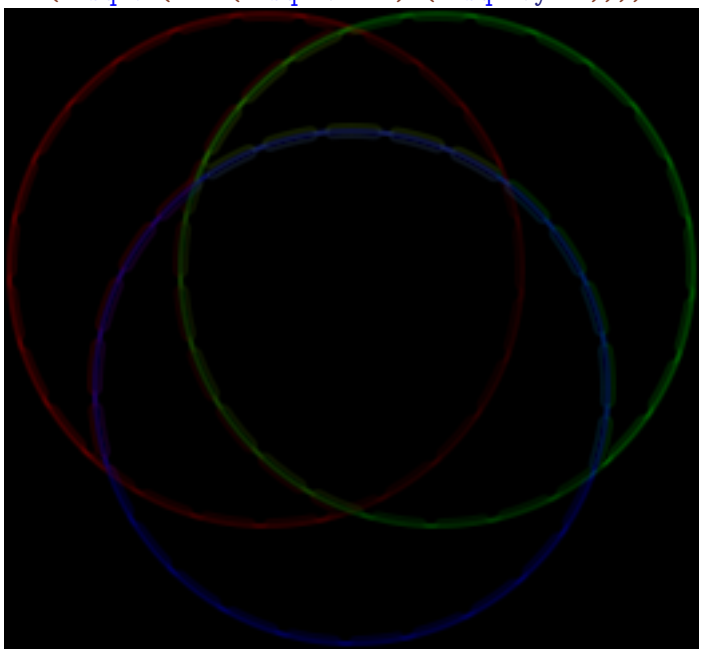
```
(fmsqrt fm) → flomap
  fm : flomap
```

```
(fmsqr fm) → flomap
  fm : flomap
```

Unary functions, lifted pointwise to operate on flomaps. Defined as `(inline-flomap-lift flsqrt)` and so on.

For example, to estimate the local gradient magnitude at each point in a flomap:

```
> (define-values (dx-fm dy-fm)
    (flomap-gradient (flomap-drop-components fm 1)))
> (flomap->bitmap
   (fmsqrt (fm+ (fmsqr dx-fm) (fmsqr dy-fm))))
```



```
(flomap-lift f) → (flomap -> flomap)
  f : (Float -> Real)
```

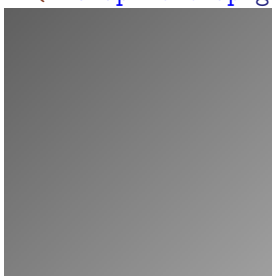
Lifts a unary floating-point function to operate pointwise on flomaps.

```
(flomap-normalize fm) → flomap
  fm : flomap
```

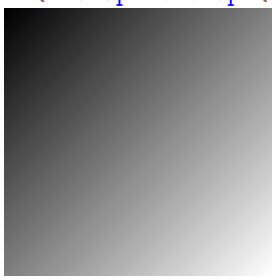
Returns a flomap like *fm*, but with values linearly rescaled to be between `0.0` and `1.0` inclusive.

Examples:

```
> (define gray-fm
    (build-flomap 1 100 100 (λ (k x y) (+ 0.375 (/ (+ x y) 800)))))
> (flomap->bitmap gray-fm)
```



```
> (flomap->bitmap (flomap-normalize gray-fm))
```



Besides increasing contrast, you could use this function to visualize oversaturated flomaps, or visualize flomaps that don't correspond directly to displayed images, such as height maps and normal maps.

```
(fm+ fm1 fm2) → flomap
  fm1 : (U Real flomap)
  fm2 : (U Real flomap)
```

```
(fm- fm1 fm2) → flomap
  fm1 : (U Real flomap)
  fm2 : (U Real flomap)
```

```
(fm* fm1 fm2) → flomap
  fm1 : (U Real flomap)
  fm2 : (U Real flomap)
```

```
(fm/ fm1 fm2) → flomap
  fm1 : (U Real flomap)
  fm2 : (U Real flomap)
```

```
(fmmin fm1 fm2) → flomap
  fm1 : (U Real flomap)
  fm2 : (U Real flomap)
```

```
(fmmax fm1 fm2) → flomap
  fm1 : (U Real flomap)
  fm2 : (U Real flomap)
```

Arithmetic, `flmin` and `flmax` lifted to operate pointwise on flomaps. Defined as `(inline-flomap-lift2 +)` and so on.

Binary operations accept the following argument combinations, in either order:

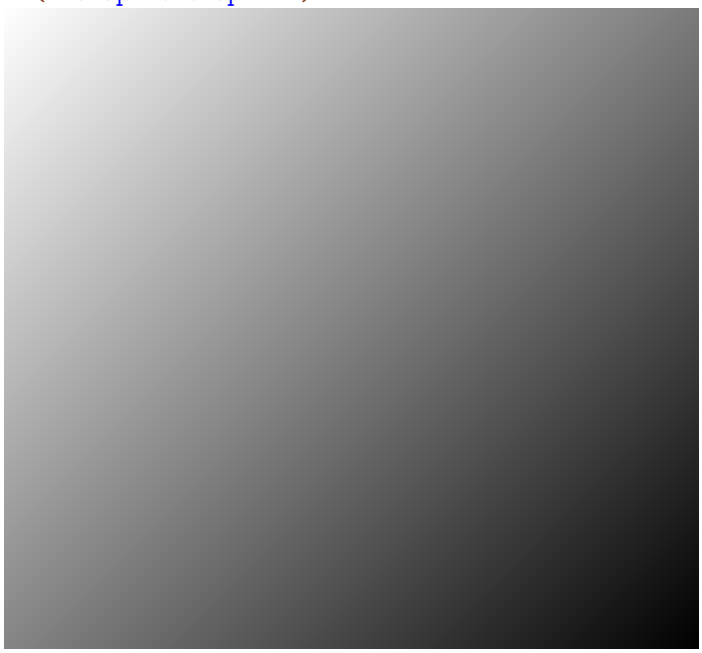
- **Two `flomap`s.** Both flomaps must have the same number of components, or one of them must have one component. If one flomap has one component, it is (conceptually) self-appended (see `flomap-append-components`) as much as needed before the operation. In either case, both flomaps must have the same width and height.
- **One `flomap`, one `Real`.** In this case, the real value is (conceptually) made into a uniform flomap (see `make-flomap`) before applying the operation.

Any other argument combination will raise a type error.

Examples:

```
> (define fm1 (build-flomap 1 260 240 (λ (k x y) (/ (+ x y) 500))))
> (define fm2 (fm- 1.0 fm1))
> (flomap->bitmap fm1)
```

```
> (flomap->bitmap fm2)
```



```
> (flomap->bitmap (fmmax fm1 fm2))
```

```
> (flomap->bitmap (fm* fm1 fm))
```

```
> (fm/ (make-flomap 1 10 10 0.5)
       (make-flomap 1 30 30 0.25))
```

*fm/: expected same-size flomaps; given sizes 10×10 and 30×30*

Binary pointwise operators could behave according to the §4.1.2 “Conceptual Model”—that is, expand the smaller one to the larger size by filling it with `0.0`. However, operating on the components of two different-size flomaps almost always indicates a logic or design error. If it really is intended, use `flomap-inset` or `subflomap` to expand the smaller flomap manually, with more control over the expansion.

Because `fm` is an alpha-multiplied flomap (see §4.1.3 “Opacity (Alpha Components)”), multiplying each component by a scalar less than `1.0` results in a more transparent flomap:

```
> (flomap->bitmap (fm* fm 0.2))
```



```
(flomap-lift2 f) → ((U Real flomap) (U Real flomap) -> flomap)
  f : (Float Float -> Real)
```

Lifts a binary floating-point function to operate pointwise on flomaps, allowing the same argument combinations as `fm+` and others.

```
(inline-flomap-lift f)

  f : (Float -> Float)
```

A macro version of `flomap-lift`. The function or macro *f* must return a `Float`, not a `Real` as the *f* argument to `flomap-lift` can.

Using `inline-flomap-lift` instead of `flomap-lift` may ensure that *f* is inlined, and therefore floats remain unboxed.

**This is not available in untyped Racket.**

```
(inline-flomap-lift2 f)

  f : (Float Float -> Float)
```

A macro version of `flomap-lift2`. The function or macro `f` must return a `Float`, not a `Real` as the `f` argument to `flomap-lift2` can.

Using `inline-flomap-lift2` instead of `flomap-lift2` may ensure that `f` is inlined, and therefore floats remain unboxed.

**This is not available in untyped Racket.**

## 4.6 Gradients and Normals

```
(flomap-gradient-x fm) → flomap
  fm : flomap
```

```
(flomap-gradient-y fm) → flomap
  fm : flomap
```

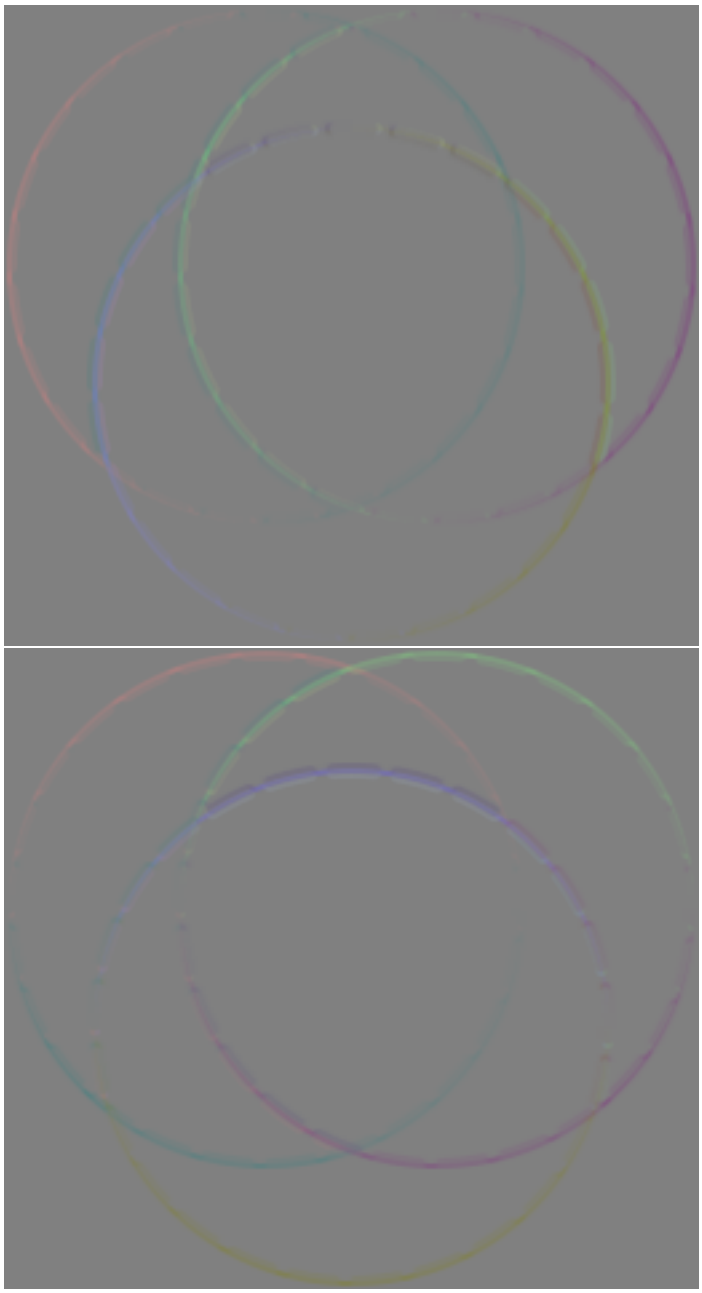
These return, per-component, estimates of the local $x$- and $y$-directional derivatives using a $3\times3$ Scharr operator.

```
(flomap-gradient fm) → flomap flomap
  fm : flomap
```

Equivalent to `(values (flomap-gradient-x fm) (flomap-gradient-y fm))`.

Examples:

```
> (define-values (dx-fm dy-fm)
    (flomap-gradient (flomap-drop-components fm 1)))
> (values (flomap->bitmap (fm* 0.5 (fm+ 1.0 dx-fm)))
          (flomap->bitmap (fm* 0.5 (fm+ 1.0 dy-fm))))
```



```
(flomap-gradient-normal fm) → flomap
  fm : flomap
```

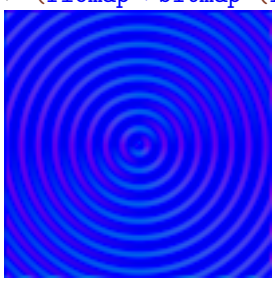
Given a one-component flomap, returns a 3-component flomap containing estimated nor-

mals. In other words, `flomap-normal` converts height maps to normal maps.

Examples:

```
> (flomap->bitmap sine-fm)
```

```
> (flomap->bitmap (flomap-gradient-normal sine-fm))
```



```
> (flomap-gradient-normal fm)
```

*flomap-gradient-normal: expected argument of type <flomap with 1 component>; given: (flomap (flvector 0.0 0.0 0.0

```
> (flomap->bitmap (flomap-gaussian-blur (flomap-inset fm 12) 4))
```

```
> (flomap->bitmap (flomap-gaussian-blur (flomap-
inset fm 12 3) 4 1))
```

```
(flomap-gaussian-blur-x fm σ) → flomap
  fm : flomap
  σ : Real
```

Returns *fm* convolved, per-component and per-row, with a Gaussian kernel with standard deviation $\sigma$.

If perfect Gaussian blur is not important, use `flomap-blur-x` instead, which approximates Gaussian blur closely and is usually much faster.

Example:

```
> (flomap->bitmap (flomap-gaussian-blur-x (flomap-
inset fm 12 0) 4))
```

```
(flomap-gaussian-blur-y fm σ) → flomap
  fm : flomap
  σ : Real
```

Like `flomap-gaussian-blur-x`, but per-column instead of per-row.

```
(flomap-box-blur fm x-radius [y-radius]) → flomap
  fm : flomap
  x-radius : Real
  y-radius : Real = x-radius
```

Returns *fm* convolved, per-component, with a box kernel with radii *x-radius* and *y-radius*. The radii are of the largest axis-aligned ellipse that would fit in the box.

Examples:

```
> (flomap->bitmap (flomap-box-blur (flomap-inset fm 4) 4))
```

```
> (flomap->bitmap (flomap-box-blur (flomap-inset fm 4 1) 4 1))
```

```
(flomap-box-blur-x fm radius) → flomap
  fm : flomap
```

```
  radius : Real
```

Returns *fm* convolved, per-component and per-row, with a box kernel with radius *radius*.

Example:

```
> (flomap->bitmap (flomap-box-blur-x (flomap-inset fm 4 0) 4))
```

```
(flomap-box-blur-y fm radius) → flomap
  fm : flomap
  radius : Real
```

Like `flomap-box-blur-x`, but per-column instead of per-row.

```
(flomap-blur fm xσ [yσ]) → flomap
  fm : flomap
  xσ : Real
  yσ : Real = xσ
```

Returns approximately the result of `(flomap-gaussian-blur fm xσ yσ)`.

Gaussian blur, as it is implemented by `flomap-gaussian-blur`, is O($x\sigma$ + $y\sigma$) for any fixed flomap size. On the other hand, `flomap-blur` is O(1) for the same size.

Examples:

```
> (define gauss-blur-fm (time (flomap-gaussian-blur fm 12)))
cpu time: 537 real time: 89 gc time: 47
> (define blur-fm (time (flomap-blur fm 12)))
cpu time: 171 real time: 29 gc time: 16
> (flomap-extreme-values
   (fmsqr (fm- gauss-blur-fm blur-fm)))
0.0
0.0031721674640532663
```

```
(flomap-blur-x fm xσ) → flomap
  fm : flomap
  xσ : Real
```

Like `flomap-blur`, but blurs per-row only.

```
(flomap-blur-y fm yσ) → flomap
  fm : flomap
  yσ : Real
```

Like `flomap-blur`, but blurs per-column only.

## 4.8 Resizing

```
(flomap-copy fm x-start y-start x-end y-end) → flomap
  fm : flomap
  x-start : Integer
  y-start : Integer
  x-end : Integer
  y-end : Integer
```

Returns the part of *fm* for which the x coordinate is $x\text{-}start \leqslant x < x\text{-}end$ and the y coordinate is $y\text{-}start \leqslant y < y\text{-}end$. If $x\text{-}start \geqslant x\text{-}end$, the result is width 0, and if $y\text{-}start \geqslant y\text{-}end$, the result is height 0.

The interval arguments may identify a rectangle with points outside the bounds of *fm*. In this case, the points' values in the returned flomap are 0.0, as per the §4.1.2 "Conceptual Model".

This function is guaranteed to return a copy.

```
(subflomap fm x-start y-start x-end y-end) → flomap
  fm : flomap
  x-start : Integer
```

```
  y-start : Integer
  x-end : Integer
  y-end : Integer
```

Like flomap-copy, but returns *fm* when *x-start* and *y-start* are 0, and *x-end* and *y-end* are respectively the width and height of *fm*.

Use subflomap instead of flomap-copy when programming functionally. Every library function that returns parts of a flomap (such as flomap-trim and flomap-inset) is defined using subflomap.

```
(flomap-trim fm [alpha?]) → flomap
  fm : flomap
  alpha? : Boolean = #t
```

Shrinks *fm* to its largest nonzero rectangle. If *alpha?* is #t, it uses only component 0 to determine the largest nonzero rectangle; otherwise, it uses every component.

This function cannot return a larger flomap.

Examples:

```
> (define small-circle-fm
    (draw-flomap (λ (fm-dc)
                   (send fm-dc draw-ellipse 20 20 10 10))
                 100 100))
> (flomap->bitmap small-circle-fm)
```

```
> (flomap->bitmap (flomap-trim small-circle-fm))
```



See flomap-nonzero-rect.

```
(flomap-inset fm amt) → flomap
  fm : flomap
  amt : Integer
(flomap-inset fm h-amt v-amt) → flomap
  fm : flomap
```

```
  h-amt : Integer
  v-amt : Integer
(flomap-inset fm l-amt t-amt r-amt b-amt) → flomap
  fm : flomap
  l-amt : Integer
  t-amt : Integer
  r-amt : Integer
  b-amt : Integer
```

Extends *fm* by some amount on each side, filling any new values with `0.0`. Positive inset amounts grow the flomap; negative insets shrink it. Large negative insets may shrink it to $0 \times 0$, which is a valid flomap size.

Example:

```
> (flomap->bitmap (flomap-inset fm -10 20 -30 -40))
```

```
(flomap-crop fm w h left-frac top-frac) → flomap
  fm : flomap
  w : Integer
  h : Integer
  left-frac : Real
  top-frac : Real
```

Shrinks or grows *fm* to be size $w \times h$. The proportion of points removed/added to the left and top are given by *left-frac* and *top-frac*; e.g. *left-frac* = `1/2` causes the same number to be removed/added to the left and right sides.

You will most likely want to use one of the following cropping functions instead, which are defined using `flomap-crop`.

```
(flomap-lt-crop fm w h) → flomap
  fm : flomap
  w : Integer
  h : Integer
```

```
(flomap-lc-crop fm w h) → flomap
  fm : flomap
  w : Integer
  h : Integer
```

```
(flomap-lb-crop fm w h) → flomap
  fm : flomap
  w : Integer
  h : Integer
```

```
(flomap-ct-crop fm w h) → flomap
  fm : flomap
  w : Integer
  h : Integer
```

```
(flomap-cc-crop fm w h) → flomap
  fm : flomap
  w : Integer
  h : Integer
```

```
(flomap-cb-crop fm w h) → flomap
  fm : flomap
  w : Integer
  h : Integer
```

```
(flomap-rt-crop fm w h) → flomap
  fm : flomap
  w : Integer
  h : Integer
```

```
(flomap-rc-crop fm w h) → flomap
  fm : flomap
  w : Integer
  h : Integer
```

```
(flomap-rb-crop fm w h) → flomap
  fm : flomap
  w : Integer
  h : Integer
```

These shrink or grow *fm* to be size $w \times h$. The two-letter abbreviation determines which area of the flomap is preserved. For example, `flomap-lt-crop` ("flomap left-top crop") preserves the left-top corner:

```
> (flomap->bitmap (flomap-lt-crop fm 150 150))
```

```
(flomap-scale fm x-scale [y-scale]) → flomap
  fm : flomap
  x-scale : Real
  y-scale : Real = x-scale
```

Scales *fm* to a proportion of its size. Uses bilinear interpolation to sample between integer coordinates, and reduces resolution (blurs) correctly before downsampling so that shrunk images are still sharp but not aliased (pixelated-looking).

Examples:

```
> (flomap->bitmap (flomap-scale fm 1/8))
```

```
> (flomap->bitmap (flomap-scale sine-fm 4))
```

```
> (flomap-scale fm 0)
(flomap (flvector) 4 0 0)
```

```
(flomap-resize fm w h) → flomap
  fm : flomap
  w : (Option Integer)
  h : (Option Integer)
```

Like `flomap-scale`, but accepts a width *w* and height *h* instead of scaling proportions. If either size is `#f`, the flomap is scaled in that direction to maintain its aspect ratio.

Examples:

```
> (flomap->bitmap (flomap-resize fm 50 #f))
```

```
> (flomap->bitmap (flomap-resize fm #f 50))
```

```
> (flomap->bitmap (flomap-resize fm 20 50))
```

```
> (flomap-resize fm 0 0)
(flomap (flvector) 4 0 0)
```

## 4.9 Compositing

Unless stated otherwise, compositing functions assume every flomap argument has an alpha component.

```
(flomap-pin fm1 x1 y1 fm2) → flomap
  fm1 : flomap
  x1 : Integer
  y1 : Integer
  fm2 : flomap
(flomap-pin fm1 x1 y1 fm2 x2 y2) → flomap
  fm1 : flomap
  x1 : Integer
  y1 : Integer
  fm2 : flomap
  x2 : Integer
  y2 : Integer
```

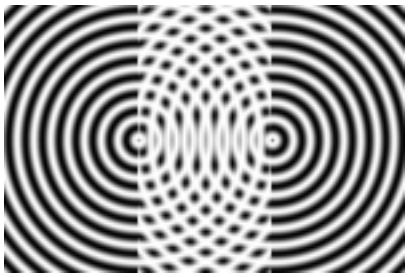
Superimposes *fm2* over *fm1* so that point *x2* *y2* on flomap `f2` is directly over point *x1* *y1* on flomap `f1`. If *x2* and *y2* are not provided, they are assumed to be `0`. The result is expanded as necessary.

*fm1* and *fm2* must have the same number of components.

Examples:

```
> (flomap-pin fm -10 -10 sine-fm)
flomap-pin: expected two flomaps with the same number of
components; given one with 4 and one with 1
> (define circle-fm
    (draw-flomap (λ (fm-dc)
                   (send fm-dc set-pen "black" 4 'short-dash)
                   (send fm-dc set-brush "yellow" 'solid)
                   (send fm-dc set-alpha 1/2)
                   (send fm-dc draw-ellipse 2 2 124 124))
                 128 128))
> (flomap->bitmap (flomap-pin fm 0 0 circle-fm 64 64))
```

```
> (flomap->bitmap (flomap-pin sine-fm 50 0 sine-fm))
```



The other compositing functions are defined in terms of `flomap-pin`.

```
(flomap-pin* x1-frac
             y1-frac
             x2-frac
             y2-frac
             fm0
             fm ...) → flomap
  x1-frac : Real
  y1-frac : Real
  x2-frac : Real
  y2-frac : Real
  fm0 : flomap
  fm : flomap
```

For each adjacent pair `fm1` `fm2` in the argument list, pins `fm2` over `fm1`.

The pin-over points are calculated from the four real arguments as follows. If `fm1` is size `w1`×`h1`, then `x1` = `(* w1 x1-frac)` and `y1` = `(* h1 y1-frac)`, and similarly for `x2` and `y2`.

The following example pins the upper-left corner of each `fm2` over a point near the upper-left corner of each `fm1`:

```
> (flomap->bitmap (flomap-pin* 1/8 1/8 0 0
                               circle-fm circle-fm circle-fm))
```

All the flomap superimpose and append functions are defined using `flomap-pin*` with different pin-over point fractions. For example, `(flomap-lt-superimpose fm0 fm ...)` = `(flomap-pin* 0 0 0 0 fm0 fm ...)`, and `(flomap-vc-append fm0 fm ...)` = `(flomap-pin* 1/2 1 1/2 0 fm0 fm ...)`.

```
(flomap-lt-superimpose fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-lc-superimpose fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-lb-superimpose fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-ct-superimpose fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-cc-superimpose fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-cb-superimpose fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-rt-superimpose fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-rc-superimpose fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-rb-superimpose fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

These create a new flomap by superimposing the flomaps in the argument list. The two-letter abbreviation determines the pin-over points. For example, `flomap-lt-superimpose` ("flomap left-top superimpose") pins points `0 0` together on each adjacent pair of flomaps:

```
> (flomap->bitmap (flomap-lt-superimpose fm circle-fm))
```

See flomap-pin and flomap-pin* for implementation details.

```
(flomap-vl-append fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-vc-append fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-vr-append fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-ht-append fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-hc-append fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

```
(flomap-hb-append fm0 fm ...) → flomap
  fm0 : flomap
  fm : flomap
```

These create a new flomap by spatially appending the flomaps in the argument list. The two-letter abbreviation determines direction (v or h) and alignment (l, c, r, or t, c, b).

Example:

```
> (flomap->bitmap (flomap-ht-append circle-fm fm
                                    (flomap-scale circle-fm 1/2)))
```

See `flomap-pin` and `flomap-pin*` for implementation details.

## 4.10 Spatial Transformations

This section gives the API for applying spatial transforms to a flomap, such as rotations, warps, morphs, and lens distortion effects.

To use the provided transforms, apply a function like `flomap-flip-horizontal` directly, or apply something like a `flomap-rotate-transform` to a flomap using `flomap-transform`.

To make your own transforms, compose existing ones with `flomap-transform-compose`, or construct a value of type `Flomap-Transform` directly:

```
(: my-awesome-transform Flomap-Transform)
(define (my-awesome-transform w h)
  (make-flomap-2d-mapping fun inv))
```

Here, `fun` is a mapping from input coordinates to output coordinates and `inv` is its inverse.

Contents:

### 4.10.1 Provided Transformations

(flomap-flip-horizontal *fm*) → flomap
  *fm* : flomap

(flomap-flip-vertical *fm*) → flomap
  *fm* : flomap

(flomap-transpose *fm*) → flomap
  *fm* : flomap

(flomap-cw-rotate *fm*) → flomap
  *fm* : flomap

(flomap-ccw-rotate *fm*) → flomap
  *fm* : flomap

Some standard image transforms. These are lossless, in that repeated applications do not degrade (blur or alias) the image.

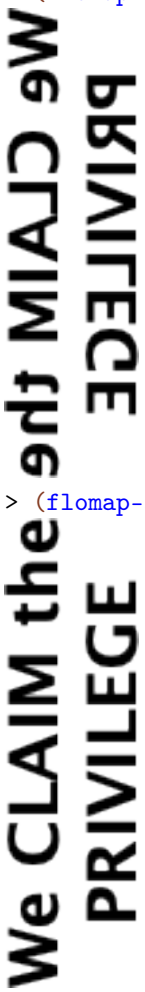
Examples:

```
> (require pict)
> (define text-fm
    (flomap-trim
     (bitmap->flomap
      (pict->bitmap (vc-append (text "We CLAIM the" '(bold) 25)
                               (text "PRIVILEGE" '(bold) 25))))))
> (flomap->bitmap text-fm)
```

```
> (flomap->bitmap (flomap-flip-horizontal text-fm))
```

```
> (flomap->bitmap (flomap-flip-vertical text-fm))
```

```
> (flomap->bitmap (flomap-transpose text-fm))
```



```
> (flomap->bitmap (flomap-cw-rotate text-fm))
```



```
> (flomap->bitmap (flomap-ccw-rotate text-fm))
```

```
> (equal? (flomap-cw-rotate fm)
          (flomap-flip-vertical (flomap-transpose fm)))
#t
> (equal? (flomap-ccw-rotate fm)
          (flomap-flip-horizontal (flomap-transpose fm)))
#t
```

```
(flomap-rotate fm θ) → flomap
  fm : flomap
  θ : Real
```

Returns a flomap rotated by $\theta$ radians counterclockwise. Equivalent to `(flomap-transform fm (flomap-rotate-transform θ))`.

Example:

```
> (flomap->bitmap (flomap-rotate text-fm (* 1/4 pi)))
```

```
(flomap-rotate-transform θ) → Flomap-Transform
  θ : Real
```

Returns a flomap transform that rotates a flomap $\theta$ radians counterclockwise around its (`Real`-valued) center.

Use `flomap-rotate-transform` if you need to know the bounds of the rotated flomap or need to compose a rotation with another transform using `flomap-transform-compose`.

Examples:

```
> (flomap-transform-bounds (flomap-rotate-transform (* 1/4 pi))
                           100 100)
-21
-21
121
121
> (flomap->bitmap
   (flomap-transform text-fm (flomap-rotate-
transform (* 1/4 pi))))
```

```
(flomap-whirl-transform θ) → Flomap-Transform
  θ : Real
```

Returns a flomap transform that "whirls" a flomap: rotates it counterclockwise $\theta$ radians in the center, and rotates less with more distance from the center.

This transform does not alter the size of its input.

Example:

```
> (flomap->bitmap
```

```
   (flomap-transform text-fm (flomap-whirl-transform pi)))
```

```
(flomap-fisheye-transform α) → Flomap-Transform
  α : Real
```

Returns a flomap transform that simulates "fisheye" lens distortion with an $\alpha$ diagonal angle of view. Equivalent to

```
(flomap-projection-transform (equal-area-projection α)
                             (perspective-projection α)
                             #f)
```

Example:

```
> (flomap->bitmap
   (flomap-transform text-fm (flomap-fisheye-
transform (* 2/3 pi))))
```

```
(flomap-scale-transform x-scale [y-scale]) → Flomap-Transform
  x-scale : Real
  y-scale : Real = x-scale
```

Returns a flomap transform that scales flomaps by *x-scale* horizontally and *y-scale* vertically.

You should generally prefer to use `flomap-scale`, which is faster and correctly reduces resolution before downsampling to avoid aliasing. This is provided for composition with other transforms using `flomap-transform-compose`.

```
flomap-id-transform : Flomap-Transform
```

A flomap transform that does nothing. See `flomap-transform-compose` for an example of using `flomap-id-transform` as the initial value for a fold.

### 4.10.2 General Transformations

```
(flomap-transform fm t) → flomap
  fm : flomap
  t : Flomap-Transform
(flomap-transform fm
                  t
                  x-start
                  y-start
                  x-end
                  y-end)  → flomap
  fm : flomap
  t : Flomap-Transform
  x-start : Integer
  y-start : Integer
  x-end : Integer
  y-end : Integer
```

Applies spatial transform `t` to `fm`.

The rectangle `x-start y-start x-end y-end` is with respect to the `fm`'s *transformed* coordinates. If given, points in `fm` are transformed only if their transformed coordinates are within that rectangle. If not given, `flomap-transform` uses the rectangle returned by `(flomap-transform-bounds t w h)`, where `w` and `h` are the size of `fm`.

This transform doubles a flomap's size:

```
> (define (double-transform w h)
    (make-flomap-2d-mapping (λ (x y) (values (* x 2) (* y 2)))
                            (λ (x y) (values (/ x 2) (/ y 2)))))
> (flomap->bitmap
   (flomap-transform text-fm double-transform))
```

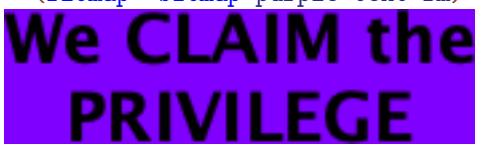
Transforms can use the width and height arguments `w h` however they wish; for example, `double-transform` ignores them, and `flomap-rotate-transform` uses them to calculate the center coordinate.

The `flomap-rotate` function usually increases the size of a flomap to fit its corners in the result. To rotate in a way that does not change the size—i.e. to do an *in-place* rotation—use `0 0 w h` as the transformed rectangle:

```
> (define (flomap-in-place-rotate fm θ)
    (define-values (w h) (flomap-size fm))
    (flomap-transform fm (flomap-rotate-transform θ) 0 0 w h))
```

Using it on `text-fm` with a purple background:

```
> (define-values (text-fm-w text-fm-h) (flomap-size text-fm))
> (define purple-text-fm
    (flomap-lt-superimpose (make-flomap* text-fm-w text-fm-
h #(1 1/2 0 1))
                           text-fm))
> (flomap->bitmap purple-text-fm)
```




```
> (flomap->bitmap (flomap-in-place-rotate purple-text-
fm (* 1/8 pi)))
```

See `flomap-projection-transform` for another example of using `flomap-transform`'s rectangle arguments, to manually crop a lens projection.

Alternatively, we could define a new transform-producing function `flomap-in-place-rotate-transform` that never transforms points outside of the orginal flomap:

```
> (define ((flomap-in-place-rotate-transform θ) w h)
    (match-define (flomap-2d-mapping fun inv _)
      ((flomap-rotate-transform θ) w h))
    (make-flomap-2d-mapping (λ (x y)
                              (let-values ([(x y)  (fun x y)])
                                (values (if (<= 0 x w) x +nan.0)
                                        (if (<= 0 y h) y +nan.0))))
                            inv))
> (flomap->bitmap
   (flomap-transform purple-text-fm
                     (flomap-in-place-rotate-
transform (* 1/8 pi))))
```

To transform *fm*, `flomap-transform` uses only the `inv` field of `(t w h)`. Every point `new-x new-y` in the transformed bounds is given the components returned by

```
(let-values ([(old-x old-y)  (inv new-x new-y)])
  (flomap-bilinear-ref* fm old-x old-y))
```

The forward mapping `fun` is used by `flomap-transform-bounds`.

`Flomap-Transform`

Defined as `(Integer Integer -> flomap-2d-mapping)`.

A value of type `Flomap-Transform` receives the width and height of a flomap to operate on, and returns a `flomap-2d-mapping` on the coordinates of flomaps of that size.

```
(struct flomap-2d-mapping (fun inv bounded-by))
  fun : (Float Float -> (values Float Float))
  inv : (Float Float -> (values Float Float))
  bounded-by : (U 'id 'corners 'edges 'all)
```

Represents an invertible mapping from `Real`×`Real` to `Real`×`Real`, or from real-valued flomap coordinates to real-valued flomap coordinates. See `flomap-transform` for examples. See §4.1.2 “Conceptual Model” for the meaning of real-valued flomap coordinates.

The forward mapping `fun` is used to determine the bounds of a transformed flomap. (See `flomap-transform-bounds` for details.) The inverse mapping `inv` is used to actually transform the flomap. (See `flomap-transform` for details.)

The symbol `bounded-by` tells `flomap-transform-bounds` how to transform bounds. In order of efficiency:

- `'id`: Do not transform bounds. Use this for in-place transforms such as `flomap-whirl-transform`.
- `'corners`: Return the smallest rectangle containing only the transformed corners. Use this for linear and affine transforms (such as `flomap-rotate-transform` or a skew transform), transforms that do not produce extreme points, and others for which it can be proved (or at least empirically demonstrated) that the rectangle containing the transformed corners contains all the transformed points.

- `'edges`: Return the smallest rectangle containing only the transformed left, top, right, and bottom edges. Use this for transforms that are almost-everywhere continuous and invertible—which describes most interesting transforms.
- `'all`: Return the smallest rectangle containing all the transformed points. Use this for transforms that produce overlaps and other non-invertible results.

For good performance, define instances of `flomap-2d-mapping` and functions that return them (e.g. instances of `Flomap-Transform`), in Typed Racket. Defining them in untyped Racket makes every application of `fun` and `inv` contract-checked when used in typed code, such as the implementation of `flomap-transform`. (In the worst case, `flomap-transform` applies `fun` to every pair of coordinates in the input flomap. It always applies `inv` to every pair of coordinates in the output flomap.)

```
(make-flomap-2d-mapping fun inv [bounded-by]) → flomap-2d-mapping
  fun : (Float Float -> (values Real Real))
  inv : (Float Float -> (values Real Real))
  bounded-by : (U 'id 'corners 'edges 'all) = 'edges
```

A more permissive, more convenient constructor for `flomap-2d-mapping`.

```
(flomap-transform-compose t2 t1) → Flomap-Transform
  t2 : Flomap-Transform
  t1 : Flomap-Transform
```

Composes two flomap transforms. Applying the result of `(flomap-transform-compose t2 t1)` is the same as applying `t1` and then `t2`, **except**:

- The points are transformed only once, meaning their component values are estimated only once, so the result is less degraded (blurry or aliased).
- The bounds are generally tighter.

The following example "whirls" `text-fm` clockwise 360 degrees and back. This is first done by applying the two transforms separately, and secondly by applying a composition of them.

```
> (let* ([text-fm  (flomap-transform
                    text-fm (flomap-whirl-transform (* 2 pi)))]
         [text-fm  (flomap-transform
                    text-fm (flomap-whirl-transform (* -2 pi)))])
    (flomap->bitmap text-fm))
```

```
> (flomap->bitmap
   (flomap-transform text-fm (flomap-transform-compose
                              (flomap-whirl-transform (* -2 pi))
                              (flomap-whirl-transform (* 2 pi)))))
```

Notice the heavy aliasing (a “Moiré pattern”) in the first result is not in the second.

In the next example, notice that rotating multiple times blurs the result and pads it with transparent points, but that applying composed rotation transforms doesn’t:

```
> (let* ([text-fm  (flomap-rotate text-fm (* 1/8 pi))]
         [text-fm  (flomap-rotate text-fm (* 1/8 pi))]
         [text-fm  (flomap-rotate text-fm (* 1/8 pi))]
         [text-fm  (flomap-rotate text-fm (* 1/8 pi))])
    (flomap->bitmap text-fm))
```

```
> (define rotate-pi/2
    (for/fold ([t flomap-id-transform]) ([_  (in-range 4)])
      (flomap-transform-compose (flomap-rotate-
transform (* 1/8 pi)) t)))
> (flomap->bitmap (flomap-transform text-fm rotate-pi/2))
```

How the bounds for the composed transform are calculated depends on how they would have been calculated for *t1* and *t2*. Suppose b1 is the bounds rule for (*t1 w h*) and b2 is the bounds rule for (*t2 w h*). Then the bounds rule b for (flomap-transform-compose *t2 t1*) is determined by the following rules, applied in order:

- If either b1 = 'all or b2 = 'all, then b = 'all.
- If either b1 = 'edges or b2 = 'edges, then b = 'edges.
- If either b1 = 'corners or b2 = 'corners, then b = 'corners.
- Otherwise, b1 = b2 = 'id, so b = 'id.

See flomap-2d-mapping for details on how b affects bounds calculation.

```
(flomap-transform-bounds t w h) → Integer Integer Integer Integer
  t : Flomap-Transform
  w : Integer
  h : Integer
```

Returns the rectangle that would contain a $w \times h$ flomap after transform by *t*.

How the rectangle is determined depends on the bounded-by field of (*t w h*). See flomap-2d-mapping for details.

See flomap-rotate-transform and flomap-projection-transform for examples.

### 4.10.3 Lens Projection and Correction

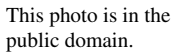
The following API demonstrates a parameterized family of spatial transforms. It also provides a physically grounded generalization of the flomap transforms returned by `flomap-fisheye-transform`.

```
(flomap-projection-transform to-proj
                             from-proj
                             crop?)     → Flomap-Transform
  to-proj : Projection
  from-proj : Projection
  crop? : Boolean
```

Returns a flomap transform that corrects for or simulates lens distortion.

To correct for lens distortion in a flomap:

- Find a projection *from-proj* that models the actual lens.
- Find a projection *to-proj* that models the desired (but fictional) lens.
- Apply `(flomap-projection-transform to-proj from-proj)` to the flomap.



This photo is in the public domain.

In the following example, a photo of the State of the Union address was taken using an "equal area" (or "equisolid angle") fisheye lens with a 180-degree diagonal angle of view:

```
> (flomap->bitmap state-of-the-union-fm)
```

We would like it to have been taken with a perfect "rectilinear" (or "perspective projection") lens with a 120-degree diagonal angle of view. Following the steps above, we apply a projection transform using `(equal-area-projection (degrees->radians 180))` for *from-proj* and `(perspective-projection (degrees->radians 120))` for *to-proj*:

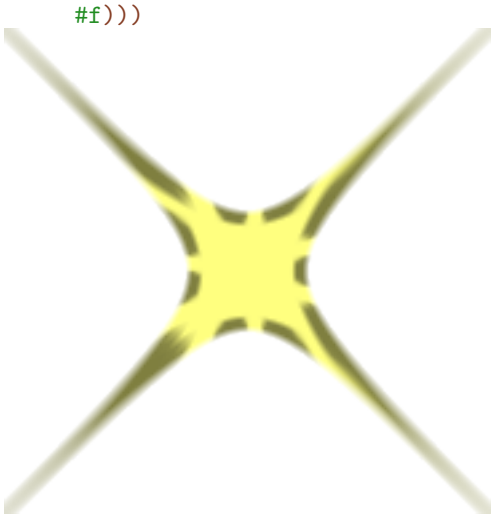
```
> (flomap->bitmap
   (flomap-transform
    state-of-the-union-fm
    (flomap-projection-transform
     (perspective-projection (degrees->radians 120))
     (equal-area-projection (degrees->radians 180)))))
```

Notice that the straight geometry in the House chamber (especially the trim around the ceiling) is represented by straight edges in the corrected photo.

When *crop?* is #t, the output flomap is no larger than the input flomap. When *crop?* is #f, the output flomap is large enough to contain the entire transformed flomap. An uncropped result can be quite large, especially with angles of view at or near 180 degrees.

```
> (define rectangle-fm
    (draw-flomap (λ (fm-dc)
                   (send fm-dc set-pen "black" 4 'dot)
                   (send fm-dc set-brush "yellow" 'solid)
                   (send fm-dc set-alpha 1/2)
                   (send fm-dc draw-rectangle 0 0 32 32))
                 32 32))
> (flomap->bitmap rectangle-fm)
```

```
> (flomap-transform-bounds
   (flomap-projection-transform
    (perspective-projection (degrees->radians 90))
    (equal-area-projection (degrees->radians 180))
    #f)
   32 32)
-56481829139474512
-56481829139474520
56481829139474552
56481829139474552
> (flomap->bitmap
   (flomap-transform
    rectangle-fm
    (flomap-projection-transform
     (perspective-projection (degrees->radians 90))
     (orthographic-projection (degrees->radians 160))
     #f)))
```



To crop manually, apply `flomap-transform` to explicit rectangle arguments:

```
> (flomap->bitmap
   (flomap-transform
    rectangle-fm
    (flomap-projection-transform
     (perspective-projection (degrees->radians 90))
     (orthographic-projection (degrees->radians 160))
     #f)
    -10 -10 42 42))
```

```
(perspective-projection α) → Projection
  α : Real
```

```
(linear-projection α) → Projection
  α : Real
```

```
(orthographic-projection α) → Projection
  α : Real
```

```
(equal-area-projection α) → Projection
  α : Real
```

```
(stereographic-projection α) → Projection
  α : Real
```

Given a diagonal angle of view $\alpha$, these all return a projection modeling some kind of camera lens. See Fisheye Lens for the defining formulas.

```
Projection
```

Equivalent to `(Float -> projection-mapping)`.

A value of type `Projection` receives the diagonal size of a flomap to operate on, and returns a `projection-mapping` instance. The provided projections (such as `perspective-projection`) use a closed-over diagonal angle of view $\alpha$ and the diagonal size to calculate the focal length.

```
(struct projection-mapping (fun inv))
  fun : (Float -> Float)
  inv : (Float -> Float)
```

Represents an invertible function from a point's angle $\rho$ from the optical axis, to the distance `r` to the center of a photo, in flomap coordinates.

For example, given a diagonal angle of view $\alpha$ and the diagonal size `d` of a flomap, the `perspective-projection` function calculates the focal length `f`:

```
(define f (/ d 2.0 (tan (* 0.5 α))))
```

It then constructs the projection mapping as

```
(projection-mapping (λ (ρ) (* (tan ρ) f))
                    (λ (r) (atan (/ r f))))
```

See Fisheye Lens for details.

## 4.11 Effects

```
(flomap-shadow fm σ [color]) → flomap
  fm : flomap
  σ : Real
  color : (Option (U (Vectorof Real) FlVector)) = #f
```

Returns the alpha (zeroth) component of *fm*, blurred with standard deviation $\sigma$ and colorized by *color*. Assumes *fm* and *color* are alpha-multiplied; see §4.1.3 “Opacity (Alpha Components)”.

If *color* = `#f`, it is interpreted as `(flvector 1.0 0.0 ...)`, or opaque black.

Examples:

```
> (flomap->bitmap
   (flomap-shadow (flomap-inset text-fm 12) 4 #(1/2 1/8 0 1/4)))
```

```
> (flomap->bitmap
   (flomap-cc-superimpose
    (flomap-shadow (flomap-inset text-fm 12) 4 #(1/2 1/8 0 1/4))
    text-fm))
```

```
(flomap-outline fm radius [color]) → flomap
  fm : flomap
  radius : Real
  color : (Option (U (Vectorof Real) FlVector)) = #f
```

Returns a flomap that outlines *fm* with a *radius*-thick line when *fm* is superimposed over it. Assumes *fm* and *color* are alpha-multiplied; see §4.1.3 “Opacity (Alpha Components)”.

If *color* = `#f`, it is interpreted as `(flvector 1.0 0.0 ...)`, or opaque black.

Examples:

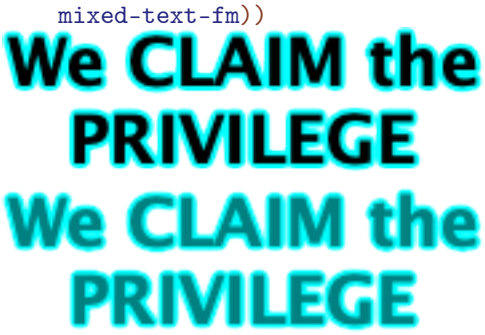
```
> (flomap->bitmap
   (flomap-outline (flomap-inset text-fm 2) 2 #(1 0 1 1)))
```

```
> (flomap->bitmap
   (flomap-cc-superimpose
    (flomap-outline (flomap-inset text-fm 2) 2 #(1 0 1 1))
    text-fm))
```

The greatest alpha value in the returned outline is the greatest alpha value in *fm*. Because of this, `flomap-outline` does fine with flomaps with fully opaque regions that are made semi-transparent:

```
> (define trans-text-fm (fm* 0.5 text-fm))
> (flomap->bitmap trans-text-fm)
```

```
> (flomap->bitmap
   (flomap-cc-superimpose
    (flomap-outline (flomap-inset trans-text-fm 2) 2 #(1 0 1 1))
    trans-text-fm))
```

However, it does not do so well with flomaps that are partly opaque and partly semi-transparent:

```
> (define mixed-text-fm
    (flomap-vc-append text-fm (make-flomap 4 0 10) trans-text-fm))
> (flomap->bitmap
   (flomap-cc-superimpose
    (flomap-outline (flomap-inset mixed-text-fm 2) 2 #(1 0 1 1))
    mixed-text-fm))
```