Geometry: a package for basic geometric calculation in Common Lisp

JOHN ALAN MCDONALD
MICHAEL SANNELLA
Depts. of Statistics, Computer Science and Engineering,
University of Washington

February, 1991

This document describes the Geometry module, a component of a system called Arizona, now under development at the U. of Washington.

Additional reports describe the other currently released modules in Arizona, Slate and Chart [13, 12].

Arizona is intended to be a portable, public-domain collection of tools supporting scientific computing, quantitative graphics, and data analysis, implemented in Common Lisp and CLOS (the Common Lisp Object System) [1, 6, 7, 14, 16, 17, 18, 19]. This document assumes the reader is familiar with Common Lisp and CLOS. An overview of Arizona is given in [8] and an introduction to the current release is in [11].

The code implementing the Geometry module is found somewhere like az/geometry/. All definitions are in the Geometry package, nicknamed g.

The Geometry module is intended to support common geometric calculations arising in graphics, numerical linear algebra, optimization, and scientific computing in general. The goal is to allow geometric computation using abstractions that directly represent high level mathematical concepts like affine spaces, vectors, and linear transformations, while retaining the level of performance provided by traditional scientific subroutine packages like Linpack[5]. It will be derived from Cactus [10, 9] and improved with ideas from work by DeRose [2, 3, 4] and Segal [15].

However, this release of the Geometry module does not support general geometric calculations. It consists of two submodules specialized for high performance in simple graphics calculations, needed by the Slate and Chart modules discussed below. The Screen Geometry submodule supports calculations in a discrete two-dimensional coordinate system called Screen Space, i.e. a bitmapped display. The Chart Geometry submodule supports calculations in a continuous (float) two-dimensional coordinate system, called Chart Space, a natural world space for simple scientific diagrams. In addition, the Geometry module provides affine mappings between the Chart and Screen spaces.

These two submodules make no use of CLOS, because of the relatively low performance of the currently available implementations. There is consequently a fair amount of inelegant
redundancy in the two submodules. For example, we have separate screen-point-x and chart-point-x operations, rather than a single, generic point-x. We expect that with better CLOS implementation it should be possible to reduce the size of these two submodules and also better support extension to higher level geometric calculations, for example, that it will be possible, with loss in performance, to replace (screen-point-x p) by a combination of (declare (type Screen-Point p)) and (point-x p).

1 Screen Geometry

The Screen Geometry submodule is intended to support geometric calculation using high-level mathematical abstractions, similar to those used in [2, 3, 4, 10, 9, 15], but restricted to and optimized for the common special case of a discrete two-dimensional coordinate system, i.e. a bitmapped display. In addition to performance, requiring the coordinates to be integers makes it easier to deal reliably with the off-by-one errors that are the curse of many bitmap graphics systems.

Screen Geometry provides an affine space of screen points and vectors and an algebra of operations on screen points and vectors. This affine space is the natural range of the last mapping in a viewing pipeline. It also provides objects representing simple rectangular regions and functions for standard calculations with these regions (e.g. intersection). It may be extended in the future to include objects representing other geometric shapes.

Screen Geometry was developed to support the Slate package [13].

1.1 Screen Coordinates

These definitions are found somewhere like az/geometry/screen-coordinates.lisp.

(deftype g:Screen-Coordinate ()

This is a datatype for the possible values of a point on a slate:Screen or slate:Slate (see [13]). It will be a subtype of Integer and usually a subtype of Fixnum.

(deftype g:Positive-Screen-Coordinate ()

This is a data type for the possible values of the width, height, etc., of a screen object. Positive-Screen-Coordinate really means non-negative; it will be a subtype of (Integer 0 *).

(deftype g:Screen-Coordinate-List ()
(deftype g:Positive-Screen-Coordinate-List ()

These types are supplied to make it convenient to check if all the coordinates in a list are legal.
1.2 Screen Vectors

1.2.1 The abstract type

(defun g:make-screen-vector (&key x y)
  (tools:type-check g:Screen-Coordinate x y)
  (values g:Screen-Vector))

(defun g:equal-screen-vectors? (v0 v1)
  (tools:type-check g:Screen-Vector v0 v1)
  (values (or T Nil)))

(defun g:copy-screen-vector (v &key result)
  (tools:type-check g:Screen-Vector v result)
  (values result))

1.2.2 Accessors

(defun g:screen-vector-x (v)
  (tools:type-check g:Screen-Vector v)
  (values g:Screen-Coordinate))

(defun g:screen-vector-y (v)
  (tools:type-check g:Screen-Vector v)
  (values g:Screen-Coordinate))

(defun (setf g:screen-vector-x) (x v)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Vector v)
  (values x))

(defun (setf g:screen-vector-y) (y v)
  (tools:type-check g:Screen-Coordinate y)
  (tools:type-check g:Screen-Vector v)
  (values y))

(defun g:set-screen-vector-coords (v &key x y)
  (tools:type-check g:Screen-Vector v)
  (tools:type-check g:Screen-Coordinate x y)
1.2.3 Related Types

(deftype g:Positive-Screen-Vector ()

The coordinates of a Positive-Screen-Vector are both Positive-Screen-Coordinates.

(deftype g:Screen-Vector-List ()
(deftype g:Positive-Screen-Vector-List ()

1.2.4 Screen Vector Resource

(defun g:borrow-screen-vector (&key x y)
  (tools:type-check g:Screen-Coordinate x y)
  (values g:Screen-Vector))

(defun g:return-screen-vector (v)
  (tools:type-check g:Screen-Vector v))

(defun g:with-borrowed-screen-vector ((name &key x y) &body body))

1.2.5 Screen Vector Algebra

(defun g:linear-mix-screen-vectors (a0 v0 a1 v1 &key result)
  (tools:type-check Number a0 a1)
  (tools:type-check g:Screen-Vector v0 v1 result)
  (values result))

(defun g:add-screen-vectors (v0 v1 &key result)
  (tools:type-check g:Screen-Vector v0 v1 result)
  (values result))

(defun g:subtract-screen-vectors (v0 v1 &key result)
  (tools:type-check g:Screen-Vector v0 v1 result)
  (values result))
(defun g:screen-vector-12-norm2 (v)
  (tools:type-check g:Screen-Vector v)
  (values g:Positive-Screen-Coordinate))

(defun g:screen-vector-12-norm (v)
  (tools:type-check g:Screen-Vector v)
  (values (Float 0.0 *)))

(defun g:screen-vector-11-norm (v)
  (tools:type-check g:Screen-Vector v)
  (values g:Positive-Screen-Coordinate))

1.3 Screen Points
1.3.1 The abstract type

(deftype g:Screen-Point ()

(defun g:make-screen-point (&key x y)
  (tools:type-check g:Screen-Coordinate x y)
  (values g:Screen-Point))

(defun g:copy-screen-point (p &key result)
  (tools:type-check g:Screen-Point p result)
  (values result))

(defun g:equal-screen-points? (p0 p1)
  (tools:type-check g:Screen-Point p0 p1)
  (values (or T Nil)))

1.3.2 Accessors

(defun g:screen-point-x (p)
  (tools:type-check g:Screen-Point p)
  (values g:Screen-Coordinate))

(defun g:screen-point-y (p)
  (tools:type-check g:Screen-Point p)
  (values g:Screen-Coordinate))
(defun (setf g:screen-point-x) (x p)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Point p)
  (values x))

(defun (setf g:screen-point-y) (y p)
  (tools:type-check g:Screen-Coordinate y)
  (tools:type-check g:Screen-Point p)
  (values y))

(defun g:set-screen-point-coords (p &key x y)
  (tools:type-check g:Screen-Point p)
  (tools:type-check g:Screen-Coordinate x y)
  (values p))

1.3.3 Related Types

(deftype g:Screen-Point-List ()

1.3.4 Screen Point Resource

(defun g:borrow-screen-point (&key x y)
  (tools:type-check g:Screen-Coordinate x y)
  (values g:Screen-Point))

(defun g:return-screen-point (p)
  (tools:type-check g:Screen-Point p))

(defmacro g:with-borrowed-screen-point ((name &key x y) &body body))

(defmacro g:with-borrowed-screen-points (names &body body))

(defun g:borrow-screen-point-list (npoints)
  (tools:type-check Positive-Fixnum npoints)
  (values g:Screen-Point-List))

(defun g:return-screen-point-list (p-list)
  (tools:type-check g:Screen-Point-List p-list))

(defmacro g:with-borrowed-screen-point-list ((name npoints) &body body))
1.3.5 Screen Point Algebra

(defun g:move-screen-point (p dp &key result)
  (tools:type-check g:Screen-Point p result)
  (tools:type-check g:Screen-Vector dp)
  (values result))

(defun g:subtract-screen-points (p0 p1 &key result)
  (tools:type-check g:Screen-Point p0 p1)
  (tools:type-check g:Screen-Vector result)
  (values result))

1.4 Screen Rects

A Screen-Rect is a screen rectangle whose sides are parallel to the coordinate axes. Other names are commonly used for this, including rectangle and region. We use Screen-Rect because we want to reserve rectangle for general rectangles and region for more general specifications of sets of pixels. We welcome any suggestions for names that are better than Screen-Rect. (One we have considered is 2D-Interval, but it seems too verbose.)

We use a coordinate system where x increases from left to right, and, unfortunately, like most window systems, y from top to bottom.

Screen-Rects have the following generalized accessors:
- screen-rect-xmin, screen-rect-xmax, screen-rect-ymin, screen-rect-ymax,
- screen-rect-left, screen-rect-top, screen-rect-right, screen-rect-bottom,
- screen-rect-width, and screen-rect-height.

At present, all rects are represented internally by xmin, width, ymin, and height slots. For Screen-Rects, xmin is equivalent to left, and ymin to top. The internal representation may be changed at any time.


Because of the pernicious danger of fence post errors, we give a careful description of which pixels the various coordinates refer to, and also give names to common alternative specifications:

The (screen-rect-left, screen-rect-right) and
(screen-rect-top, screen-rect-bottom) pairs are specifications of integer intervals like
the (start, end) parameters to the CL sequence functions. Screen-rect-left is the co-
ordinate of the leftmost column that intersects the Screen-Rect and screen-rect-top
the coordinate of the topmost row that intersects the screen-rect. Screen-rect-right
is the coordinate of the first column not intersecting the Screen-Rect to the right and
screen-rect-bottom is the coordinate of the first row below the Screen-Rect not inter-
secting it. (See figure refscreen-rect-coordinates.) This means that screen-rect-width
= screen-rect-right - screen-rect-left is the number of columns that intersect the
screen-rect-left
|    | screen-rect-right
|    |
\   \v   v
....... 
.xxxxx. <--screen-rect-top
.xxxxx.
.xxxxx.
....... <--screen-rect-bottom

Figure 1: Screen Rect coordinates

outer-left
|inner-left
|| inner-right
|| |outer-right
|| ||
\   \vv   v
....... <--outer-top
.xxxxx. <--inner-top
.xxxxx.
.xxxxx. <--inner-bottom
....... <--outer-bottom

Figure 2: Other Screen Rect coordinates

screen-rect, screen-rect-height = screen-rect-bottom - screen-rect-top is the number of rows that intersect the screen-rect, and
screen-rect-width * screen-rect-height is the number of pixels in the screen-rect.

Implementation of drawing operations will require interfacing with other parameterizations; to make this a little easier, we give standard names to commonly occurring coordinates, shown in figure 2.

For example, implementations of the drawing operations will often need to transform to the analogous parameterization in top-to-bottom coordinates. This means converting our (inner-left, inner-bottom, width, height) representation to the coordinates of the (inner-left, inner-top, width, height) in the top-to-bottom coordinates.

There's a natural ambiguity about what's supposed to happen when we change the coordinates of a Rect. When we change the right coordinate, does that mean translating the Rect rigidly so its width remains fixed, or does it mean leaving the left coordinate fixed and changing the width.
We have adopted the convention that setting any of the bounds (left, right, top, bottom, outer-left, etc.) of the Rect is interpreted as changing the origin, that is, a translation of the Rect without changing its extent (width and height). Setting the width and height changes the size or shape of the Rect, holding the origin fixed. This convention applies to both Screen and ChartRects.

Width and height must be non-negative. Zero width and zero height are taken to imply an empty screen-rect. A width and height of 1 means the Screen-Rect covers one pixel. Empty screen-rects still have a location.

### 1.4.1 The Abstract Type

**(deftype g:Screen-Rect ()**

**(defun g:make-screen-rect (&key vidth left height bottom top)**

(Tools:type-check g:Screen-Coordinate right left bottom top)

(Tools:type-check g:Positive-Screen-Coordinate width height)

(assert (= width (- right left)))

(assert (= height (- bottom top)))

(values g:Screen-Rect))

**(defun g:equal-screen-rects? (r result)**

(Tools:type-check g:Screen-Rect r result)

(values (or T Nil))

**(defun g:copy-screen-rect (r &key result)**

(Tools:type-check g:Screen-Rect r result)

(values result))

### 1.4.2 Accessors

Point and vector based accessors:

These functions get and set the upper left corner of the Rect.

**(defun g:screen-rect-origin (r &key result)**

(Tools:type-check g:Screen-Rect r)

(Tools:type-check g:Screen-Point result)

(values result))

**(defun (setf g:screen-rect-origin) (x r)**

(Tools:type-check g:Screen-Point x)

(Tools:type-check g:Screen-Rect r)

(values x))
These functions get and set a vector corresponding to the diagonal (width, height) of the Rect.

(defun g:screen-rect-extent (r &key result)
  (tools:type-check g:Screen-Rect r)
  (tools:type-check g:Positive-Screen-Vector result)
  (values result))

(defun (setf g:screen-rect-extent) (x r)
  (tools:type-check g:Positive-Screen-Vector x)
  (tools:type-check g:Screen-Rect r))

Basic coordinate accessors:

(defun g:screen-rect-xmin (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate))

(defun (setf g:screen-rect-xmin) (x r)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Rect r)
  (values x))

(defun g:screen-rect-xmax (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate))

(defun (setf g:screen-rect-xmax) (x r)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Rect r)
  (values x))

(defun g:screen-rect-width (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Positive-Screen-Coordinate))

(defun (setf g:screen-rect-width) (w r)
  (tools:type-check g:Positive-Screen-Coordinate w)
  (tools:type-check g:Screen-Rect r)
  (values w))

(defun g:screen-rect-ymin (r)
  (tools:type-check g:Screen-Rect r)
(values g:Positive-Screen-Coordinate)
(values g:Screen-Coordinate))

(defun (setf g:screen-rect-ymin) (y r)
  (tools:type-check g:Screen-Coordinate y)
  (tools:type-check g:Screen-Rect r)
  (values y))

(defun g:screen-rect-ymax (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate))

(defun (setf g:screen-rect-ymax) (y r)
  (tools:type-check g:Screen-Coordinate y)
  (tools:type-check g:Screen-Rect r)
  (values y))

(defun g:screen-rect-height (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Positive-Screen-Coordinate))

(defun (setf g:screen-rect-height) (h r)
  (tools:type-check g:Positive-Screen-Coordinate h)
  (tools:type-check g:Screen-Rect r)
  (values h))

Other, conveniently named accessors:

(defun g:screen-rect-left (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate))

(defun (setf g:screen-rect-left) (x r)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Rect r)
  (values x))

(defun g:screen-rect-inner-left (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate)
  (values g:Screen-Coordinate))
(defun g:screen-rect-inner-left (x r)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Rect r)
  (values x))

(defun g:screen-rect-outer-left (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate))

(defun (setf g:screen-rect-outer-left) (x r)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Rect r)
  (values x))

(defun g:screen-rect-right (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate))

(defun (setf g:screen-rect-right) (x r)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Rect r)
  (values x))

(defun g:screen-rect-inner-right (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate))

(defun (setf g:screen-rect-inner-right) (x r)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Rect r)
  (values x))

(defun g:screen-rect-outer-right (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate))

(defun (setf g:screen-rect-outer-right) (x r)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Rect r)
  (values x))

(defun g:screen-rect-top (r)
(tools:type-check g:Screen-Rect r)
(values g:Screen-Coordinate))

(defun (setf g:screen-rect-top) (x r)
(tools:type-check g:Screen-Coordinate x)
(tools:type-check g:Screen-Rect r)
(values x))

(defun g:screen-rect-inner-top (r)
(tools:type-check g:Screen-Rect r)
(values g:Screen-Coordinate))

(defun (setf g:screen-rect-inner-top) (x r)
(tools:type-check g:Screen-Coordinate x)
(tools:type-check g:Screen-Rect r)
(values x))

(defun g:screen-rect-outer-top (r)
(tools:type-check g:Screen-Rect r)
(values g:Screen-Coordinate))

(defun (setf g:screen-rect-outer-top) (x r)
(tools:type-check g:Screen-Coordinate x)
(tools:type-check g:Screen-Rect r)
(values x))

(defun g:screen-rect-bottom (r)
(tools:type-check g:Screen-Rect r)
(values g:Screen-Coordinate))

(defun (setf g:screen-rect-bottom) (x r)
(tools:type-check g:Screen-Coordinate x)
(tools:type-check g:Screen-Rect r)
(values x))

(defun g:screen-rect-inner-bottom (r)
(tools:type-check g:Screen-Rect r)
(values g:Screen-Coordinate))

(defun (setf g:screen-rect-inner-bottom) (x r)
(tools:type-check g:Screen-Coordinate x)
(tools:type-check g:Screen-Rect r)
(values x))

(defun g:screen-rect-outer-bottom (r)
  (tools:type-check g:Screen-Rect r)
  (values g:Screen-Coordinate))

(defun (setf g:screen-rect-outer-bottom) (x r)
  (tools:type-check g:Screen-Coordinate x)
  (tools:type-check g:Screen-Rect r)
  (values x))

(defun g:set-screen-rect-coords (r
  &key
  width right left
  height bottom top)
  (tools:type-check g:Screen-Rect r)
  (tools:type-check g:Screen-Coordinate right left bottom top)
  (tools:type-check g:Positive-Screen-Coordinate width height)
  (assert (= width (- right left)))
  (assert (= height (- bottom top)))
  (values r))

1.4.3 A Resource for g:Screen-Rects

(defun g:borrow-screen-rect (&key width right left height bottom top)
  (tools:type-check g:Screen-Coordinate right left bottom top)
  (tools:type-check g:Positive-Screen-Coordinate width height)
  (assert (= width (- right left)))
  (assert (= height (- bottom top)))
  (values g:Screen-Rect))

(defun g:return-screen-rect (r) (tools:type-check g:Screen-Rect r))

(defmacro g:with-borrowed-screen-rect ((name
  &key
  width right left
  height bottom top)
  &body body))

(defmacro g:with-borrowed-screen-rects ((names

14
1.5 Geometric Calculations

(deftype g:Screen-Object ()
  '(or g:Screen-Vector g:Screen-Point g:Screen-Rect))

(defun g:sub-screen-rect? (smaller larger)
  (tools:type-check g:Screen-Rect smaller larger)
  (values (or T Nil)))

(defun g:screen-point-in-rect? (point rect)
  (tools:type-check g:Screen-Point point)
  (tools:type-check g:Screen-Rect rect)
  (values (or T Nil)))

(defun g:screen-points-in-rect? (points r)
  (tools:type-check g:Screen-Point-List points)
  (tools:type-check g:Screen-Rect r)
  (values (or T Nil)))

(defun g:screen-rect-intersect? (r0 rl)
  (tools:type-check g:Screen-Rect r0 rl)
  (values (or T Nil)))

(defun g:intersect-screen-rects (rects &key result)
  (tools:type-check g:Screen-Rect-List rects)
  (tools:type-check g:Screen-Rect result)
  (values result))

(defun g:screen-rect-covering-rects (rects &key result)
  (tools:type-check g:Screen-Rect-List rects)
  (tools:type-check g:Screen-Rect result)
  (values result))

(defun g:screen-rect-covering-points (points &key result)
  (tools:type-check g:Screen-Point-List points)
  (tools:type-check g:Screen-Rect result)
(values result))

(defun g:extend-screen-rect-to-point (r p &key result)
  (tools:type-check g:Screen-Point p)
  (tools:type-check g:Screen-Rect r result)
  (values result))

(defun g:extend-screen-rect-to-rect (rect-to-extend r &key result)
  (tools:type-check g:Screen-Rect rect-to-extend r result)
  (values result))

(defun g:screen-rect-min-12-dist2 (r0 r1)
  (tools:type-check g:Screen-Rect r0 r1)
  (values g:Positive-Screen-Coordinate))

(defun g:screen-rect-min-12-dist (r0 r1)
  (tools:type-check g:Screen-Rect r0 r1)
  (values (Float 0.0 *))

(defun g:screen-distance-to-line (p lp0 lp1)
  (tools:type-check g:Screen-Point p lp0 lp1)
  (values (Float 0.0 *)))

2 Chart Geometry

The Chart Geometry submodule is intended to support geometric calculation using high-level mathematical abstractions, similar to those used in [2, 3, 4, 10, 9, 15], but restricted to and optimized for the common special case of a two-dimensional coordinate system. Specializing for the two-dimensional case gives us some small performance benefits, but the main reason for implementing Chart Geometry was as a quick and dirty world space for the scientific diagrams provided by the Chart module [12].

Like the Screen Geometry module, Chart Geometry provides an affine space of chart points and vectors and an algebra of operations on those points and vectors. It also provides objects representing simple rectangular regions and functions for standard calculations with these regions (e.g. intersection). It may be extended in the future to include objects representing other geometric shapes.

2.1 Chart Coordinates

These definitions are found somewhere like az/geometry/chart-coordinates.lisp.

(deftype g:Chart-Coordinate ())
This is a datatype for the possible values of a coordinate on a `slate:Chart` (see [12]). It’s safe to assume it is a subtype of `Float`.

```lisp
(deftype g:Positive-Chart-Coordinate ()

This is a data type for the possible values of the width, height, etc., of a chart object. “Positive” really means non-negative; it’s safe to assume it is a subtype of `(Float 0.0 *)`.

```lisp
(deftype g:Chart-Coordinate-List ()
(deftype g:Positive-Chart-Coordinate-List ()

These types are supplied to make it convenient to check if all the coordinates in a list are legal.

### 2.2 Chart Vectors

**2.2.1 The abstract type**

```lisp
(deftype g:Chart-Vector ()

(defun g:make-chart-vector (&key x y)
  (tools:type-check g:Chart-Coordinate x y)
  (values g:Chart-Vector))

(defun g:equal-chart-vectors? (v0 v1)
  (tools:type-check g:Chart-Vector v0 v1)
  (values (or T Nil)))

(defun g:copy-chart-vector (v &key result)
  (tools:type-check g:Chart-Vector v result)
  (values result))

**2.2.2 Accessors**

```lisp
(defun g:chart-vector-x (v)
  (tools:type-check g:Chart-Vector v)
  (values g:Chart-Coordinate))

(defun g:chart-vector-y (v)
  (tools:type-check g:Chart-Vector v)
  (values g:Chart-Coordinate))

17
(defun (setf g:chart-vector-x) (x v)  
   (tools:type-check g:Chart-Coordinate x)  
   (tools:type-check g:Chart-Vector v)  
   (values x))

(defun (setf g:chart-vector-y) (y v)  
   (tools:type-check g:Chart-Coordinate y)  
   (tools:type-check g:Chart-Vector v)  
   (values y))

(defun g:set-chart-vector-coords (v &key x y)  
   (tools:type-check g:Chart-Vector v)  
   (tools:type-check g:Chart-Coordinate x y)  
   (values v))

2.2.3 Related Types

(deftype g:Positive-Chart-Vector ())

The coordinates of a Positive-Chart-Vector are both Positive-Chart-Coordinates.

(deftype g:Chart-Vector-List ())
(deftype g:Positive-Chart-Vector-List ())

2.2.4 Chart Vector Resource

(defun g:borrow-chart-vector (&key x y)  
   (tools:type-check g:Chart-Coordinate x y)  
   (values g:Chart-Vector))

(defun g:return-chart-vector (v)  
   (tools:type-check g:Chart-Vector v))

(defmacro g:with-borrowed-chart-vector ((name &key x y) &body body))
2.2.5 Chart Vector Algebra

(defun g:linear-mix-chart-vectors (a0 v0 a1 v1 &key result)
  (tools:type-check Number a0 a1)
  (tools:type-check g:Chart-Vector v0 v1 result)
  (values result))

(defun g:add-chart-vectors (v0 v1 &key result)
  (tools:type-check g:Chart-Vector v0 v1 result)
  (values result))

(defun g:subtract-chart-vectors (v0 v1 &key result)
  (tools:type-check g:Chart-Vector v0 v1 result)
  (values result))

(defun g:chart-vector-12-norm2 (v)
  (tools:type-check g:Chart-Vector v)
  (values g:Positive-Chart-Coordinate))

(defun g:chart-vector-12-norm (v)
  (tools:type-check g:Chart-Vector v)
  (values g:Positive-Chart-Coordinate))

(defun g:chart-vector-11-norm (v)
  (tools:type-check g:Chart-Vector v)
  (values g:Positive-Chart-Coordinate))

2.3 Chart Points

2.3.1 The abstract type

(deftype g:Chart-Point ()

(defun g:make-chart-point (&key x y)
  (tools:type-check g:Chart-Coordinate x y)
  (values g:Chart-Point))

(defun g:copy-chart-point (p &key result)
  (tools:type-check g:Chart-Point p result)
  (values result))

(defun g:equal-chart-points? (p0 p1)
2.3.2 Accessors

(defun g:chart-point-x (p)
  (tools:type-check g:Chart-Point p)
  (values g:Chart-Coordinate))

(defun g:chart-point-y (p)
  (tools:type-check g:Chart-Point p)
  (values g:Chart-Coordinate))

(defun (setf g:chart-point-x) (x p)
  (tools:type-check g:Chart-Coordinate x)
  (tools:type-check g:Chart-Point p)
  (values x))

(defun (setf g:chart-point-y) (y p)
  (tools:type-check g:Chart-Coordinate y)
  (tools:type-check g:Chart-Point p)
  (values y))

(defun g:set-chart-point-coords (p &key x y)
  (tools:type-check g:Chart-Point p)
  (tools:type-check g:Chart-Coordinate x y)
  (values p))

2.3.3 Related Types

(deftype g:Chart-Point-List ())

2.3.4 Chart Point Resource

(defun g:borrow-chart-point (&key x y)
  (tools:type-check g:Chart-Coordinate x y)
  (values g:Chart-Point))

(defun g:return-chart-point (p)
(tools:type-check g:Chart-Point p))

(defmacro g:with-borrowed-chart-point ((name &key x y) &body body))

(defmacro g:with-borrowed-chart-points (names &body body))

(defun g:borrow-chart-point-list (npoints)
  (tools:type-check Positive-Fixnum npoints)
  (values g:Chart-Point-List))

(defun g:return-chart-point-list (p-list)
  (tools:type-check g:Chart-Point-List p-list))

(defmacro g:with-borrowed-chart-point-list ((name npoints) &body body))

2.3.5 Chart Point Algebra

(defun g:move-chart-point (p dp &key result)
  (tools:type-check g:Chart-Point p result)
  (tools:type-check g:Chart-Vector dp)
  (values result))

(defun g:subtract-chart-points (p0 p1 &key result)
  (tools:type-check g:Chart-Point p0 p1)
  (tools:type-check g:Chart-Vector result)
  (values result))

2.4 Chart Rects

A Chart-Rect is a chart rectangle whose sides are parallel to the coordinate axes. Other
names are commonly used for this, including rectangle and region. We use Chart-Rect
because we want to reserve rectangle for general rectangles and region for more general
sets in the plane. We welcome any suggestions for names that are better than Chart-Rect.
(One we have considered is 2D-Interval, but it seems too verbose.)

We use a coordinate system where x increases from left to right, and, in the natural way
for quantitative graphics, but unlike most window systems, y from bottom to top.

Chart-Rects have the following generalized accessor:
chart-rect-xmin, chart-rect-xmax, chart-rect-ymin, chart-rect-ymax,
chart-rect-left, chart-rect-top, chart-rect-right, chart-rect-bottom,
chart-rect-width, and chart-rect-height. At present, all rects are represented internally by xmin, width, ymin, and height slots. For Chart-Rects, xmin is equivalent to left,
and ymin to bottom.
The external interface for making Chart-Rect's make-chart-rect takes any two of :left, :right, and :width and any two of :top, :bottom, and :height. The internal representation may be changed at any time.

There's a natural ambiguity about what's supposed to happen when we change the coordinates of a Rect. When we change the right coordinate, does that mean translating the Rect rigidly so its width remains fixed, or does it mean leaving the left coordinate fixed and changing the width?

We have adopted the convention that setting any of the bounds (left, right, top, bottom, outer-left, etc.) of the Rect is interpreted as changing the origin, that is, a translation of the Rect without changing its extent (width and height). Setting the width and height changes the size or shape of the Rect, holding the origin fixed. This convention applies to both Screen and Chart Rects.

Zero width and zero height are taken to imply an empty chart-rect. Empty chart-rects still have a location.

### 2.4.1 The Abstract Type

(deftype g:Chart-Rect ())

(defun g:make-chart-rect (&key width right left height bottom top)
  (tools:type-check g:Chart-Coordinate right left bottom top)
  (tools:type-check g:Positive-Chart-Coordinate width height)
  (assert (= width (- right left)))
  (assert (= height (- top bottom)))
  (values g:Chart-Rect))

(defun g:equal-chart-rects? (r result)
  (tools:type-check g:Chart-Rect r result)
  (values (or T Nil)))

(defun g:copy-chart-rect (r &key result)
  (tools:type-check g:Chart-Rect r result)
  (values result))

### 2.4.2 Accessors

Point and vector based accessors:
These functions get and set the lower left corner of the Rect.

(defun g:chart-rect-origin (r &key result)
(tools:type-check g:Chart-Rect r)
(tools:type-check g:Chart-Point result)
(values result))

(defun (setf g:chart-rect-origin) (x r)
  (tools:type-check g:Chart-Point x)
  (tools:type-check g:Chart-Rect r)
  (values x))

These functions get and set a vector corresponding to the diagonal (width, height) of the Rect.

(defun g:chart-rect-extent (r &key result)
  (tools:type-check g:Chart-Rect r)
  (tools:type-check g:Positive-Chart-Vector result)
  (values result))

(defun (setf g:chart-rect-extent) (x r)
  (tools:type-check g:Positive-Chart-Vector x)
  (tools:type-check g:Chart-Rect r)
  (values x))

Basic coordinate accessors:

(defun g:chart-rect-xmin (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Chart-Coordinate))

(defun (setf g:chart-rect-xmin) (x r)
  (tools:type-check g:Chart-Coordinate x)
  (tools:type-check g:Chart-Rect r)
  (values x))

(defun g:chart-rect-xmax (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Chart-Coordinate))

(defun (setf g:chart-rect-xmax) (x r)
  (tools:type-check g:Chart-Coordinate x)
  (tools:type-check g:Chart-Rect r)
  (values x))
(defun g:chart-rect-width (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Positive-Chart-Coordinate))

(defun (setf g:chart-rect-width) (w r)
  (tools:type-check g:Positive-Chart-Coordinate w)
  (tools:type-check g:Chart-Rect r)
  (values w))

(defun g:chart-rect-ymin (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Positive-Chart-Coordinate)
  (values g:Chart-Coordinate))

(defun (setf g:chart-rect-ymin) (y r)
  (tools:type-check g:Chart-Coordinate y)
  (tools:type-check g:Chart-Rect r)
  (values y))

(defun g:chart-rect-ymax (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Chart-Coordinate))

(defun (setf g:chart-rect-ymax) (y r)
  (tools:type-check g:Chart-Coordinate y)
  (tools:type-check g:Chart-Rect r)
  (values y))

(defun g:chart-rect-height (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Positive-Chart-Coordinate))

(defun (setf g:chart-rect-height) (h r)
  (tools:type-check g:Positive-Chart-Coordinate h)
  (tools:type-check g:Chart-Rect r)
  (values h))

Other, conveniently named accessors:

(defun g:chart-rect-left (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Chart-Coordinate))
(defun (setf g:chart-rect-left) (x r)
  (tools:type-check g:Chart-Coordinate x)
  (tools:type-check g:Chart-Rect r)
  (values x))

(defun g:chart-rect-right (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Chart-Coordinate))

(defun (setf g:chart-rect-right) (x r)
  (tools:type-check g:Chart-Coordinate x)
  (tools:type-check g:Chart-Rect r)
  (values x))

(defun g:chart-rect-top (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Chart-Coordinate))

(defun (setf g:chart-rect-top) (x r)
  (tools:type-check g:Chart-Coordinate x)
  (tools:type-check g:Chart-Rect r)
  (values x))

(defun g:chart-rect-bottom (r)
  (tools:type-check g:Chart-Rect r)
  (values g:Chart-Coordinate))

(defun (setf g:chart-rect-bottom) (x r)
  (tools:type-check g:Chart-Coordinate x)
  (tools:type-check g:Chart-Rect r)
  (values x))

(defun g:set-chart-rect-coords (r
  &key
    width right left
    height bottom top)
  (tools:type-check g:Chart-Rect r)
  (tools:type-check g:Chart-Coordinate right left bottom top)
  (tools:type-check g:Positive-Chart-Coordinate width height)
  (assert (= width (- right left))))
(assert (= height (- bottom top)))
(values r))

2.4.3 A Resource for g:Chart-Rects

(defun g:borrow-chart-rect (&key width right left height bottom top)
  (tools:type-check g:Chart-Coordinate right left bottom top)
  (tools:type-check g:Positive-Chart-Coordinate width height)
  (assert (= width (- right left)))
  (assert (= height (- bottom top)))
  (values g:Chart-Rect))

(defun g:return-chart-rect (r) (tools:type-check g:Chart-Rect r))

(defmacro g:with-borrowed-chart-rect ((name
  &key
  width right left
  height bottom top)
  &body body))

(defmacro g:with-borrowed-chart-rects ((names
  &key
  width right left
  height bottom top)
  &body body))

2.5 Geometric Calculations

(deftype g:Chart-Object ()
  '(or g:Chart-Vector g:Chart-Point g:Chart-Rect))

(defun g:sub-chart-rect? (smaller larger)
  (tools:type-check g:Chart-Rect smaller larger)
  (values (or T Nil)))

(defun g:chart-point-in-rect? (point rect)
  (tools:type-check g:Chart-Point point)
  (tools:type-check g:Chart-Rect rect)
  (values (or T Nil)))
(defun g:chart-points-in-rect? (points r)
  (tools:type-check g:Chart-Point-List points)
  (tools:type-check g:Chart-Rect r)
  (values (or T Nil)))

(defun g:chart-rect-intersect? (r0 r1)
  (tools:type-check g:Chart-Rect r0 r1)
  (values (or T Nil)))

(defun g:intersect-chart-rects (rects &key result)
  (tools:type-check g:Chart-Rect-List rects)
  (tools:type-check g:Chart-Rect result)
  (values result))

(defun g:chart-rect-covering-rects (rects &key result)
  (tools:type-check g:Chart-Rect-List rects)
  (tools:type-check g:Chart-Rect result)
  (values result))

(defun g:chart-rect-covering-points (points &key result)
  (tools:type-check g:Chart-Point-List points)
  (tools:type-check g:Chart-Rect result)
  (values result))

(defun g:extend-chart-rect-to-point (r p &key result)
  (tools:type-check g:Chart-Point p)
  (tools:type-check g:Chart-Rect r result)
  (values result))

(defun g:extend-chart-rect-to-rect (rect-to-extend r &key result)
  (tools:type-check g:Chart-Rect rect-to-extend r result)
  (values result))

(defun g:chart-rect-min-12-dist2 (r0 r1)
  (tools:type-check g:Chart-Rect r0 r1)
  (values g:Positive-Chart-Coordinate))

(defun g:chart-rect-min-12-dist (r0 r1)
  (tools:type-check g:Chart-Rect r0 r1)
  (values g:Positive-Chart-Coordinate))

(defun g:chart-distance-to-line (p lp0 lp1)
(tools:type-check g:Chart-Point p lp0 lp1)
(values g:Positive-Chart-Coordinate))

3 Transforming between Chart and Screen Coordinates

At present, only a limited range of transforms (diagonal ones from Chart space to Screen space and from Screen space to Chart space) are available and there are only limited facilities for constructing them. We've provided only what's essential for the Chart module [12]. This should change soon.

This is a type for maps between Chart and Screen space, in either direction:

(deftype g:Diagonal-C<->S-Map ()))

To get a map that takes a given chart rect to a given screen rect, use:

(defun g:make-affine-map-between (c s)
  (tools:type-check g:Chart-Rect c)
  (tools:type-check g:Screen-Rect s)
  (values g:Diagonal-C<->S-Map))

To get the inverse map (from Screen space to Chart space) use g:inverse, which is actually a pseudo-inverse:

(defun g:inverse (map &key result)
  (tools:type-check g:Diagonal-C<->S-Map map result)
  (assert (c<->s-check-type-match map result)))

To apply either direction of map to any g:Screen-Object or any g:Chart-Object, as appropriate, use:

(defun g:transform (map x &key result)
  (tools:type-check g:Diagonal-C<->S-Map map)
  (tools:type-check (or g:Screen-Object g:Chart-Object) x)
  (assert (g:c<->s-check-type-match x result)))

The following function tests to see its arguments are of appropriately matching types, for example, one g:Screen-Rect and one g:Chart-Rect.

(defun g:c<->s-check-type-match (a b)
  (tools:type-check (or g:Screen-Object g:Chart-Object) a b)
  (values (or T Nil)))
4 Calculating Tic Positions

This sections contains some utilities for choosing nice tic mark sets for scientific diagrams. It is derived from code by Jan Pedersen and Robert Gentleman.

Descriptions of where (in chart coordinates) tic marks should be placed are encapsulated in instances of the abstract Tics type, which are intended to be immutable.

```lisp
(deftype g:Tics ())

(defun tics? (x) (values (or T Nil)))

(defun equal-tics? (t0 t1)
  (tools:type-check g:Tics t0 t1)
  (values (or T Nil)))

(defun g:tic-min (tics)
  (tools:type-check g:Tics tics)
  (values g:Chart-Coordinate))

(defun g:tic-max (tics)
  (tools:type-check g:Tics tics)
  (values g:Chart-Coordinate))

The Tics cover an interval from tic-min to tic-max.

(defun g:tic-inc (tics)
  (tools:type-check g:Tics tics)
  (values g:Positive-Chart-Coordinate))

These utilities only support equally sized tics, which are placed every tic-inc from tic-min to tic-max.

(defun g:tic-n (tics)
  (tools:type-check g:Tics tics)
  (values (Integer 0 *))

The number of tics should be (+ 1 (/ (- tic-max tic-min) tic-inc)) (fenceposts!).

(defun g:tic-values (tics)
  (tools:type-check g:Tics tics)
  (values (List g:Chart-Coordinate)))
```
tic-values is a list of the positions at which the tics are placed. Strictly speaking, it shouldn’t be needed, but it might be useful in the future if we want to generalize to non-equally spaced tics (eg. for plotting on a log scale).

(defun g:get-nice-tics (min max &key (tics '(3 4 5 6 7 8)))
  (tools:type-check Number min max)
  (tools:type-check (List Integer) tics)
  (assert (= min max))
  (values g:Tics))

get-nice-tics is the basic function for computing tic marks. The tics argument is a list of numbers of tic marks to try, that is, with the default value, the heuristic first tries fitting 3 tic marks, then 4, and so on up to 8, picking the resulting tic interval that is closest to the original interval specified by (min, max).

(defun g:nice-chart-rect (r &key (result (g:make-chart-rect)))
  (tools:type-check g:Chart-Rect r result)
  (values result xtics ytics))
nice-chart-rect is a minor convenience for a common case; it saves us having to call g:get-nice-tics twice and then modify a rect to be consistent with the resulting intervals.

References


31