SLATE: A PORTABLE BITMAP GRAPHICS PACKAGE FOR COMMON LISP

by

John Alan McDonald
Michael Sannella

TECHNICAL REPORT No. 204
February 1991

Department of Statistics, GN-22
University of Washington
Seattle, Washington 98195 USA
Slate: a portable bitmap graphics package for Common Lisp

JOHN ALAN MCDONALD
MICHAEL SANNELLA
Depts. of Statistics, Computer Science and Engineering,
University of Washington
February 1991

1 Overview

This document describes the Slate 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, Geometry 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, 4, 9, 15, 19, 20, 21, 23]. This document assumes the reader is familiar with Common Lisp and CLOS. An overview of Arizona is given in [10] and an introduction to the current release is in [11].

Slate is a low level "device-independent" graphics package that has been ported to a number of Common Lisp platforms, window systems, and other graphics devices. It is intended to be used by developers of higher level scientific and statistical graphics systems rather than end users. Slate is something like a simplified version of CLX, ported to run on window systems and graphics devices other than X11.

The source code implementing Slate is found in az/slate/. The definitions are in the Slate package. Slate requires the Tools module (described in [11]) and the Geometry module (described in [13]).

1.1 Motivation

A legitimate question is why we are defining a new CL graphics/window interface, rather than just using one or more of Garnet [16, 17], CLIM, Common Windows [8, 5, 6], CLX [22], etc.?

The primary reason is that, as of Summer 1990, none of the various proposed standard CL window interfaces are stable or widely available enough.

A secondary reason is that higher level interface systems currently under development seem not to be designed with interactive scientific graphics in mind. Having our own low
level system gives us the opportunity to adjust both the set of basic operations provided to better fit the higher level systems we want to build. And we can optimize the implementations to give the necessary performance in operations that are pretty basic for us but unusual for general purpose window/mouse based user interfaces (e.g., rotating point clouds, grand tour [2, 3, 7], image painting [14]).

1.2 Design Goals

Some of the design goals of Slate are:

- It should be robust against errors in the calls from a higher level graphics system.
- It should be easy to port to a new window system, hardcopy device, or Lisp compiler.
- It should permit high performance implementations.
- Unless there's a good reason to do otherwise, the design should be as close to X as possible, because we expect the X port to be the most important.

1.3 Slate Abstractions

The basic abstraction in Slate is (surprise) the slate (see section 11). Slates are surfaces that can be drawn on and can receive input of various kinds. To make porting easy, the imaging model is fairly primitive; Slates are essentially bitmaps of some finite depth.

At present the set of drawing operations allows us to outline, fill, or tile simple geometric shapes like points, line segments, or polygons, draw characters and strings in a variety of fonts and colors, and copy rectangular sets of pixels from one slate to another. (In the near future, we expect to extend the set of drawing operations to include support for multispectral images. In the far future, we hope to have support for realistically rendered 3d images.)

The behavior of all the drawing operations in Slate can be described in the following way: The drawing operation changes the values of a set of destination pixels in the slate. For each destination pixel, a source pixel value is calculated and the destination pixel is set to some simple function (at present some boolean combination of the bits) of the source and destination pixel values.

A typical drawing operation takes three types of arguments: a pen, which is an abstraction that encapsulates boolean operation, color, font, line style, etc. (see section 9), a slate to draw on, and a specification for the set of destination pixels. Sets of destination pixels are specified using the abstractions for discrete screen provided by the Geometry package (see [13]).

Slate provides separate abstractions for the more complex pen parameters, like colors (section 2), boolean operations (section 4), fonts (section 5), line styles (section 6), and tiling patterns (section 7).

There are two basic kinds of slates, standard, visible slates and invisible slates. All drawing operations work "in the same way" on both visible and invisible slates. An invisible slate serves roughly the same purpose as a pixmap in X. It gives us a surface to draw on.
that can be copied rapidly to multiple places on one or more visible slates. We could do the same thing with only visible slates, but we'd have to clutter the screen with little bitmaps that we don't really want the user to see. (Note that for slightly technical reasons, a Tile is not the same thing as an invisible slate. For details see section 7.)

Another major abstraction in Slate in the screen. The screen object captures information about the device a slate is actually displayed on that isn't specific to any slate, such as, the number of bits per pixel. (We considered not having a screen object, letting the device specific information be represented by class variables of each instantiable slate class. However, this got us into chicken and egg problems where we needed to have some slate made in order to get at the information needed to make a slate. Moreover, having a screen object makes Slate more closely match the abstractions in CLX.) Every slate is on a particular screen, even invisible slates, which is specified when the slate is made. A slate’s screen cannot be changed; in other words, slates cannot be moved between screens. Pixel regions can only be copied between slates on the same screen. (This restriction might be relaxed in various ways in the future.)

What color the user sees for a given pixel value is determined by the current colormap of the slate’s screen. For more on colormaps, see section 3.

\section{Colors}

\subsection{The Abstract Color Type}

\begin{verbatim}
(deftype slate:Color ())

make-color is not exported because to do so would require either exposing the internal implementation of colors or supporting a complicated interface. To make a Color object use one of the functions in section 2.2.

(defun slate:equal-colors? (color0 color1)
  (tools:type-check slate:Color color0 color1)
  (values (or T Nil)))

(defun slate:copy-color (color &key result)
  (tools:type-check slate:Color color result)
  (values result))
\end{verbatim}

\subsection{Alternatives for specifying Colors}

Colors can be specified in either by name (as in X) or in RGB, IHS, Hexcone-IHS, or YIQ coordinate systems. Some version of RGB is the primitive representation.

(The type specification of IHS, Hexcone-IHS, and YIQ may be wrong, and should be verified sometime.)
(defun slate:rgb->color (r g b &key result)
  (tools:type-check (Float 0.0 1.0) r g b)
  (tools:type-check slate:Color result)
  (values result))

(defun slate:color->rgb (color)
  (tools:type-check slate:Color color)
  (values r g b))

(defun slate:ihs->color (i h s &key result)
  (tools:type-check (Float 0.0 1.0) i h s)
  (tools:type-check slate:Color result)
  (values result))

(defun slate:color->ihs (color)
  (tools:type-check slate:Color color)
  (values i h s))

(defun slate:hexcone-ihs->color (i h s &key result)
  (tools:type-check (Float 0.0 1.0) i h s)
  (tools:type-check slate:Color result)
  (values result))

(defun slate:color->hexcone-ihs (color)
  (tools:type-check slate:Color color)
  (values i h s))

(defun slate:yiq->color (y i q &key result)
  (tools:type-check (Float 0.0 1.0) y i q)
  (tools:type-check slate:Color result)
  (values result))

(defun slate:color->yiq (color)
  (tools:type-check slate:Color color)
  (values y i q))

2.3 Converting between color coordinates

For convenience in color calculation, functions are provided for converting between coordinate systems. For the exact relationship between RGB and the other coordinate systems, see the source code. (In future, we should get a precise definition with references and add to this spec.)
(defun rgb->ihs (r g b)
  (tools:type-check (Float 0.0 1.0) r g b)
  (values i h s))

(defun ihs->rgb (i h s)
  (tools:type-check (Float 0.0 1.0) i h s)
  (values r g b))

(defun rgb->hexcone-ihs (r g b)
  (tools:type-check (Float 0.0 1.0) r g b)
  (values i h s))

(defun hexcone-ihs->rgb (i h s)
  (tools:type-check (Float 0.0 1.0) i h s)
  (values r g b))

(defun rgb->yiq (r g b)
  (tools:type-check (Float 0.0 1.0) r g b)
  (values y i q))

(defun yiq->rgb (y i q)
  (tools:type-check (Float 0.0 1.0) y i q)
  (values r g b))

2.4 Color Names

Colors can also be specified by name, as in X [18]. A Color-Name is a keyword in the list given below. Because Color-Names are just keywords, we haven't bothered to implement a full abstract type as described in [11]. You can test if a keyword is a Color-Name by doing (typep keyword 'Color-Name).

(deftype slate:Color-Name ()
  '(member
    :aquamarine :medium-aquamarine :mediumaquamarine
    :black :blue
    :cadet-blue :cadetblue :cornflower-blue :cornflowerblue
    :medium-slate-blue :mediumslateblue

5
:mediumgoldenrod :mediumgoldenrod :green
:dark-green :darkgreen
:dark-olive-green :darkolivegreen
:forest-green :forestgreen
:lime-green :limegreen
:medium-forest-green :mediumforestgreen
:medium-sea-green :mediumseagreen
:mediumspringgreen :pale-green :palegreen
:sea-green :seagreen
:spring-green :springgreen :yellow-green :yellowgreen
:dark-slate-grey :darkslategrey
:dim-grey :dimgrey
:light-gray :lightgray
:dark-orchid :darkorchid :medium-orchid :mediumorchid :pink
:plum :red :indian-red :indianred
:medium-violet-red :mediumvioletred
:orange-red :orangered :violet-red :violetred
:salmon :sienna :tan
:thistle :turquoise :dark-turquoise :darkturquoise
:medium-turquoise :mediumturquoise :violet
:blue-violet :blueviolet
:wheat :white :yellow :green-yellow :greenyellow
:gray78 :gray79 :gray80 :gray81 :gray82 :gray83 :gray84
:gray99 :gray100
To get a color object corresponding to a given keyword, use \texttt{color-name->color}.

\begin{verbatim}
(defun slate:color-name->color (name &key result)
  (tools:type-check slate:Color-Name name)
  (tools:type-check slate:Color result)
  (values result))
\end{verbatim}

\section*{2.5 A Resource of Color Objects}

We provide a resource of color objects, following the protocol discussed in \cite{11}. \texttt{borrow-color} and \texttt{with-borrowed-color} do not take any initialization options, because to do so would require either exposing the internal representation of colors, or implementing a complicated interface that could handle all the possible ways of specifying colors. In the future we may add functions like \texttt{borrow-color-rgb}, \texttt{borrow-color-ihs}, and so on.

\begin{verbatim}
(defun slate:borrow-color () (values slate:Color))
(defun slate:return-color (color) (tools:type-check slate:Color color))
(defmacro slate:with-borrowed-color ((vname) &body body))
\end{verbatim}

\section*{2.6 Color Metrics}

Colormaps (see section 3) use color metric functions to choose a pixel value whose entry in the colormap best approximates a given color. A color metric is a function of two arguments, which must be colors, which returns a non-negative single-float result. (This could be generalized to any non-negative non-complex number.) It must return zero if the two arguments are equal in the sense of \texttt{equal-colors?}. It may return zero for colors that are not equal, meaning that the application using the metric doesn't care about the
difference between the two colors, which might be the case, for example, if the two colors can't be distinguished on the display device. The default color metric for standard colormaps is color-l1-dist which computes the sum of the absolute differences of the rgb values.

(defun slate:color-l1-dist (c0 c1)
  (tools:type-check slate:Color c0 c1)
  (values (Single-Float 0.0 most-positive-single-flaot)))

3 Colormaps

Colormaps are used for two approximately inverse purposes: (1) to determine how a pixel value is mapped into color and (2) to find the pixel value whose entry in the colormap best approximates a given color.

3.1 Pixel Values

Pixel-Value is a subtype of Integer, eg:

(defun slate:Pixel-Value () '(Unsigned-Byte 32))

3.2 Colormap Protocol

Colormaps in Slate combine the functions of the Visual and Colormap in X.
All colormap classes inherit from the abstract class Colormap.

(defun slate:Colormap (Standard-Object))

(defun slate:pixel-depth (colormap)
  (declare (type slate:Colormap colormap)
    (values Fixnum)))

The pixel-depth is the maximum number of bits of a pixel value that a colormap will use.

(defun slate:legal-pixel-value? (colormap pixel-value)
  (declare (type slate:Colormap colormap)
    (type slate:Pixel-Value pixel-value)
    (values (or T Nil))))

legal-pixel-value? is a more precise way to test if a pixel value is within the range accepted by a colormap.
(defgeneric slate:color->pixel-value (colormap color &key metric &allow-other-keys)

(declare (type slate:Colormap colormap)
  (type slate:Color color)
  (type (or Symbol Function) metric)
  (values slate:Pixel-Value)))

color->pixel-value returns the pixel value whose entry best approximates the given color description (in the sense of metric). (corresponds roughly to XAllocColor).

(defgeneric slate:pixel-value->color (colormap pixel-value &key result)

(declare (type slate:Colormap colormap)
  (type slate:Pixel-Value pixel-value)
  (type slate:Color result)
  (values result)))

pixel-value->color returns the color at a given pixel value.

(defgeneric (setf slate:pixel-value->color) ( color colormap pixel-value &key result)

(declare (type slate:Color color)
  (type slate:Read-Write-Colormap colormap)
  (type slate:Pixel-Value pixel-value)
  (type slate:Color result)
  (ignore result)
  (values color)))

For writable colormaps (see section 3.3), setf can be used to change the color associated with a given pixel value.

(defgeneric slate:color-name->pixel-value (lut name &key metric)

(declare (type slate:Colormap lut)
  (type slate:Color-Name name)
  (type (or Symbol Function) metric)
  (values slate:Pixel-Value)))

color-name->pixel-value finds the pixel value whose color best approximates the named color.
3.3 Colormap Classes

Colormaps can be classified into a 2 dimensional table of basic types (see [18], ch. 7, tables 7-1 and 7-2).

One dimension of this table determines whether the colormap is writable. This dimension of the table is represented in Slate by the following two abstract classes:

\[
\begin{align*}
\text{(defclass slate:Read-Only-Colormap (slate:Colormap))} \\
\text{(defclass slate:Read-Write-Colormap (slate:Colormap))}
\end{align*}
\]

The other dimension of the table reflects how the colormap translates pixel values into color. Slate represents this dimension by the following 4 classes:

\[
\begin{align*}
\text{(defclass slate:Monochrome-Colormap (slate:Colormap))} \\
\text{(defclass slate:RGB-Colormap (slate:Colormap))} \\
\text{(defclass slate:Single-Index-RGB-Colormap (slate:RGB-Colormap))} \\
\text{(defclass slate:Decomposed-Index-RGB-Colormap (slate:RGB-Colormap))}
\end{align*}
\]

A monochrome colormap is used for a display device that translates a pixel value into a single intensity or gray level. The colormap should be constrained so that \( r = g = b \) for all the colors in the map. An RGB colormap is used for a display device that translates a pixel value into 3 separate RGB intensities. A single index colormap corresponds to a display device that uses all bits of the pixel value as the same index into red, green, and blue intensity tables. A decomposed index colormap corresponds to a device that splits the pixel value’s bits into 3 fields (eg. three 8 bit fields from a 24 bit pixel value) that are used as three independent indexes into the red, green, and blue intensity tables.

Considering all possible combinations of the above classes (excluding RGB-Colormap) gives us 6 basic abstract colormap classes:

\[
\begin{align*}
\text{(defclass slate:Static-Gray-Colormap} \\
\text{\quad (slate:Read-Only-Colormap} \\
\text{\quad \quad slate:Monochrome-Colormap))}
\end{align*}
\]

\[
\begin{align*}
\text{(defclass slate:Static-Colormap} \\
\text{\quad (slate:Read-Only-Colormap} \\
\text{\quad \quad slate:Single-Index-RGB-Colormap))}
\end{align*}
\]
(defclass slate:True-Colormap
  (slate:Read-Only-Colormap
   slate:Decomposed-Index-RGB-Colormap))

(defclass slate:Gray-Scale-Colormap
  (slate:Read-Write-Colormap
   slate:Monochrome-Colormap))

(defclass slate:Pseudo-Colormap
  (slate:Read-Write-Colormap
   slate:Single-Index-RGB-Colormap))

(defclass slate:Direct-Colormap
  (slate:Read-Write-Colormap
   slate:Decomposed-Index-RGB-Colormap))

The most common instantiable case is a Static-Gray-Colormap of depth 1:

(defclass slate:B&W-Colormap (slate:Static-Gray-Colormap))

The second most common case is a Pseudo-Colormap of depth 8:

(defclass slate:Byte-Pseudo-Colormap (slate:Pseudo-Colormap))

To allow the same pixel values to be used with reasonably consistent results on both Byte-Pseudo-Colormap's and B&W-Colormap's, a B&W-Colormap translates a zero pixel value to the background color (often white) and anything else to the foreground color (often black).

4 Boolean Operations

A Pen's boolean operation determines how setting (and unsettling) a new value into a slate pixel (by a drawing operation) combines with the existing state of the pixel. For now, the set of logical operations is the set of binary truth tables, specified by the 16 "boolean" constants in Common Lisp (CLL, section 12.7, [19, 20]).

The whole notion of "logical operation" is somewhat problematic for color and Postscript devices, so this may need to be changed in the future.

Note that it doesn't make sense to define copy-boolean-operation and make-boolean-operation.
(defparameter slate:*boolean-operations*  
  (list boole-clr boole-set boole-1 boole-2  
    boole-c1 boole-c2 boole-and boole-ior  
    boole-xor boole-eqv boole-nand boole-nor  
    boole-andc1 boole-andc2 boole-orc1 boole-orc2)  
  "A list of the standard CL boolean ops, for convenience.")

(deftype slate:Boolean-Operation ()  
  '(member #.slate:*boolean-operations*))

5 Fonts

A Font is essentially a triple of family, face and size:

(deftype slate:Font-Family () '(member :helvetica :times :fix))

(deftype slate:Font-Face () '(member :roman :bold :italic :bold-italic))

(deftype slate:Font-Size () '(Integer 0 64))

The set of possible families and faces may be extended in the future.

(deftype slate:Font ()

(defun slate:font-family (font)  
  (tools:type-check slate:Font font)  
  (values slate:Font-Family))

(defun slate:font-face (font)  
  (tools:type-check slate:Font font)  
  (values slate:Font-Face))

(defun slate:font-size (font)  
  (tools:type-check slate:Font font)  
  (values slate:Font-Size))

(defun slate:font (&key (family :fix) (face :roman) (size 12))  
  (tools:type-check slate:Font-Family family)  
  (tools:type-check slate:Font-Face face)  
  (tools:type-check slate:Font-Size size)  
  (values slate:Font))

12
Font's are an example of immutable objects, as discussed in [11]. In other words, neither the family, face, or size of a Font can be changed. The intention is that Font's should be unique, so that there is only one Font (in the sense of eq) for a given triple of family, size, and size. Because of this there is no make-font function. Instead, to get the Font corresponding to a given triple of family, size, and size, use the font function, which will return the font corresponding to family, face, and size, if it already exists, or create a new one if necessary.

With the current definitions for Font-Family, Font-Face, and Font-Size, there can be at most 768 different fonts. Keeping a table of 768 font objects should not be too great a burden on most Lisp environments and, in addition, it's unlikely that more than few dozen fonts would be instantiated by most applications.

6 Line Styles

This is the beginnings of an abstract type used by Pens to specify how lines should be drawn, based on line styles in X (see section 5.5 in [18]). It is not used, or at least not used consistently, by most of the implementations.

(defstruct (slate:Line-Style
  (:conc-name nil)
  (:constructor slate:make-line-style)
  (:copier Slate:copy-line-style)
  (slate:line-style-dash-rule :solid :type '(Member :solid :on-off-dash :double-dash))
  (slate:line-style-thickness 1 :type g:Positive-Screen-Coordinate)
  (slate:line-style-dashes nil :type List)
  (slate:line-style-dash-offset 0 :type g:Positive-Screen-Coordinate)
  ;; for future expansion:
  (slate:line-style-cap :butt :type '(Member :not-last :butt :round :projecting))
  ;; for future expansion
  (slate:line-style-join :miter :type '(Member :miter :round :bevel))
}
(slate:line-style-arrowhead-length
  8
  :type g:Positive-Screen-Coordinate)
(slate:line-style-arrowhead-angle
  (/ (float pi 1.0) 12.0)
  :type Float))

(defun slate:line-style? (ls)
  (declare (type T ls)
    (values (or T Nil))))
(defun slate:equal-line-styles? (ls0 ls1)
  (tools:type-check slate:Line-style ls0 ls1)
  (values (or T Nil)))
(defun slate:copy-line-style (ls &key result)
  (tools:type-check slate:Line-style ls result)
  (values result))

7 Tiles

Tiles are used to specify a pixel pattern that can be used by Pen's for filling regions with textures.

It might seem, at first thought, that a Pen's filling texture could be specified with an Invisible-Slate and that a new type is not necessary. However, the contents of an Invisible-Slate can only be copied to other slates with the same slate-screen (or, at least, the same type of slate-screen). We require that Pen's be device-independent, i.e., any Pen can be used on any Slate on any Screen.

To get around this problem, each Tile contains a description of its appearance in the tile-draw-function, which is a function of one argument, a slate, and should make no assumptions about the type of slate it will be applied to. The first time a Tile is used on a particular Screen, it makes an Invisible-Slate for that Screen, applies tile-draw-function to the Invisible-Slate and saves the Invisible-Slate in a cache indexed by Screen.

Tiles are assumed to be immutable, that is, neither the height, width, or draw function can be changed after a tile is created.

(deftype slate:Tile ()
  (defun slate:make-tile (&key
    tile-width
    tile-height
    (values slate:Tile)))

14
(tools:type-check g:Positive-Screen-Coordinate tile-width tile-height)
(tools:type-check (Function (slate:Tile)) tile-draw-function)
(values slate:Tile))

(defun slate:equal-tiles? (t0 t1)
  (tools:type-check slate:Tile t0 t1)
  (values (or T Nil)))

8 Fill Styles

This is the beginnings of an abstract data type used by Pens to specify how area should be filled. Only the filled? and tile slots are used at present.

(defun slate:fill-style? (Is)
  (declare (type T Is)
          (values (or T Nil))))

(defun slate:equal-fill-styles? (ls0 ls1)
  (tools:type-check slate:Fill-style ls0 ls1)
  (values (or T Nil)))

(defun slate:copy-fill-style (ls &key result)
  (tools:type-check slate:Fill-style ls result)
  (values result))
9 Pens

Pens are something like the graphics contexts in X. Pens package together all the drawing parameters that might be needed by drawing operations, except those specifying position or shape. They are specified in a completely device independent way. They may cache information that speeds their use on particular screens (e.g., the device font corresponding to a given slate:Font).

At present there is only one Pen type. It is an abstract data type, as discussed in [11]. The representation of Pens (that is, defstruct, defclass, etc.) is not intended to be visible to the user.

(deftype slate:Pen ()

(defun slate:pen? (x)
  (tools:type-check T x)
  (values (or T Nil)))

(defun slate:copy-pen (pen &key result)
  (tools:type-check slate:Pen pen result)
  (values result))

(defun slate:equal-pens? (pen0 pen1)
  (tools:type-check slate:Pen pen0 pen1)
  (values (or T Nil)))

(defun slate:make-pen (&rest options
  &key
  pen-font
  pen-operation
  pen-pixel-value
  pen-plane-pask
  pen-color
  pen-color-metric
  pen-line-style
  pen-fill-style
  &allow-other-keys)
  (tools:type-check slate:Font pen-font)
  (tools:type-check slate:Boolean-Operation pen-operation)
  (tools:type-check slate:Pixel-Value pen-pixel-value pen-plane-mask)
  (tools:type-check slate:Color pen-color)
  (tools:type-check (or Symbol
    (Function (slate:Color slate:Color) (Float 0.0 *)))
    pen-color-metric)

16
10 Screens

The reason for having Screen objects is to have a place to collect information about the underlying display device that is not specific to any Slate and that can be used before any Slates are made.

10.1 Getting a screen to use.

The recommended way to get a screen to use is to call `default-screen` which will usually return the only screen on the system.

For the rarer case where a system has more than one screen available, one can call `all-screens` to get a list of all the available screens.

```lisp
(defun slate:default-screen () (values slate:Screen))

(defun slate:all-screens () (values (List slate:Screen)))
```

10.2 Screen class

```lisp
(defclass slate:Screen (Standard-Object))
```

10.3 Screen dimensions

```lisp
(defgeneric slate:screen-pixel-depth (screen)
  (declare (type slate:Screen screen))
  (:documentation "How many bits per pixel?"))

The `screen-rect` is the area of the screen that’s visible. The `screen-origin` is the point at the upper left corner of the screen. The `screen-extent` is a vector whose coordinates are the width and height of the screen.

```lisp
(defgeneric slate:screen-rect ((screen Screen) &key result)
  (declare (type slate:Screen screen)
    (type g:Screen-Rect result)
    (values result)))

(defgeneric slate:screen-origin (screen &key result)
  (declare (type slate:Screen screen)
    (type g:Screen-Point result)
    (values result)))
```
These functions give the conversion between screen coordinates and real world dimensions (at least approximately). It's not yet decided if we should restrict these to the best Fixnum approximation, or allow Rationals, or even Floats. Most of the implementations don't support this yet.

(declare (type slate:Screen screen) 
    (type g:Screen-Vector result) 
    (values result)))

10.4 Text measurements

The size and exact placement of text does not depend on the specific slate that it's drawn, but only on how the abstract specification of font, etc., is translated for the screen the slate is on. Therefore functions to determine the size and shape of regions affected by drawing characters and strings are referred to a screen object rather than a slate. Another reason for making them functions of the screen rather than a slate is so they can be used internally, before any slates are created.

To simplify matters a little, we require that character and string size and placement depend only on the font (and the screen, of course) and not on any other pen parameters. Therefore the text placement functions take a font argument rather than a pen argument.

Character placement is illustrated in figure 1. (This discussion is based to some degree on the way character and font metrics are handled in X11 [18].) Suppose we draw the character q at a point (x,y) on some slate. There are a number of functions that allow
Figure 1: Placement of character drawn at \((x, y)\). Pixels in the \texttt{character-screen-rect} are marked with a dash; pixels outside are marked with period.

us to find out which pixels will be affected. \texttt{character-screen-rect} returns a screen rect that contains all the pixels that are changed; however, it is not necessarily the minimal such rect, but rather a rect that is chosen so that drawing sequences of characters will look nice if their rects are pasted together without any space between, at least horizontally, if not vertically.

The \texttt{character-screen-rect} of a character drawn at \((x, y)\) is not placed, as one might naively expect, so that the rect’s origin (the upper left corner) is at \((x, y)\). To make it easier to line up a sequence of characters, it is drawn so that the left most column of the rect is at \(x\) and the \texttt{baseline} column is at \(y\).

\begin{verbatim}
(defun slate:character-screen-rect (screen char font &key position result)
  (tools:type-check slate:Screen screen)
  (tools:type-check Character char)
  (tools:type-check slate:Font font)
  (tools:type-check g:Screen-Point position)
  (tools:type-check g:Screen-Rect result)
  (values result))
\end{verbatim}

We provide functions for computing \texttt{character-width} and \texttt{character-height} for convenience and because it can often be done much more efficiently than first computing the \texttt{character-screen-rect} and then taking the \texttt{g:screen-rect-width} and/or \texttt{g:screen-rect-height}.

19
The position of the baseline can be determined from the character-ascent using the 
fact that the origin of character's screen rect is \((x, y - \text{character-ascent})\).

The character-descent is defined as \((\text{character-height} - \text{character-ascent})\); we 
provide a separate function for convenience and because it can sometimes be computed 
without first computing height and ascent and subtracting.

Conceptually, the behavior of \text{string-screen-rect}, \text{string-width}, and so on, can be 
derived from the analogous character functions in the following way: Paste together the 
screen rects for the characters so that the baselines match, leaving no room between the 
 rects. \text{string-screen-rect} is the minimal screen-rect that contains all the characters' 
screen-rects. \text{string-width} is the sum of the characters' widths. \text{string-ascent} is 
the maximum character-ascent, \text{string-descent} the maximum character-descent. 
\text{string-height} is the sum of \text{string-ascent} and \text{string-descent}, which is equal to the 
height of the \text{string-screen-rect}, (and which might be larger than the largest 
character-height).
(tools:type-check slate:Screen screen)
(tools:type-check String string)
(tools:type-check slate:Font font)
(tools:type-check g:Screen-Point position)
(tools:type-check g:Screen-Rect result)
(values result))

(defvar slate:string width (screen string font)
  (tools:type-check slate:Screen screen)
  (tools:type-check String string)
  (tools:type-check slate:Font font)
  (values g:Screen-Coordinate))

(defvar slate:string height (screen string font)
  (tools:type-check slate:Screen screen)
  (tools:type-check String string)
  (tools:type-check slate:Font font)
  (values g:Screen-Coordinate))

(defvar slate:string ascent (screen string font)
  (tools:type-check slate:Screen screen)
  (tools:type-check String string)
  (tools:type-check slate:Font font)
  (values g:Screen-Coordinate))

(defvar slate:string descent (screen string font)
  (tools:type-check slate:Screen screen)
  (tools:type-check String string)
  (tools:type-check slate:Font font)
  (values g:Screen-Coordinate))

(defvar slate:maximum string width and maximum string height are useful for drawing a series of strings that, for example, should be equally spaced vertically and centered horizontally, as in a menu.

(defvar slate:maximum string width (screen strings font)
  (tools:type-check slate:Screen screen)
  (tools:type-check List strings)
  (tools:type-check slate:Font font)
  (values g:Screen-Coordinate))

(defvar slate:maximum string height (screen strings font)
A vertical string’s rect has the screen point at its upper left corner, not the left baseline of the first character. This is done to make it easier to draw a sequence of vertical strings without overlapping. vertical-string-ascent is therefore always zero. vertical-string-descent is the sum of the heights of all the characters.
(defun slate:maximum-vertical-string-width (screen strings font)
  (tools:type-check slate:Screen screen)
  (tools:type-check List strings)
  (tools:type-check slate:Font font)
  (values g:Screen-Coordinate))

(defun slate:maximum-vertical-string-height (screen strings font)
  (tools:type-check slate:Screen screen)
  (tools:type-check List strings)
  (tools:type-check slate:Font font)
  (values g:Screen-Coordinate))

10.5 Screen class hierarchy

The top of the screen class hierarchy is a set of abstract classes that correspond directly to the basic colormap classes (see section 3). Every screen has a slate:screen-colormap which is an instance of the corresponding colormap class.

(defclass slate:Screen (Standard-Object)
  ((slate:screen-colormap
    :type slate:Colormap
    :reader slate:screen-colormap)))

make-colormap-for-screen makes a colormap of the appropriate type, possibly using options to initialize the map.

(defun slate:make-colormap-for-screen (screen &rest options)
  (declare (type slate:Screen screen)))

Screens can be classified into a 2 dimensional table of basic types corresponding to the two-dimensional table of Colormap types (see section 3 and [18], ch. 7, tables 7-1 and 7-2). One dimension of this table determines whether the colormap is writable. This dimension of the table is represented in Slate by the following two abstract classes:

(defclass slate:Read-Only-Colormap-Screen (slate:Screen)
  ((slate:screen-colormap
    :type slate:Read-Only-Colormap
    :reader slate:screen-colormap)))

23
The other dimension of the table reflects how the colormap translates pixel values into color. Slate represents this dimension by the following 4 classes:

(defclass slate:Monochrome-Screen (slate:Screen)
 ((slate:screen-colormap
     :type slate:Monochrome-Colormap
     :reader slate:screen-colormap)))

(defclass slate:RGB-Screen (slate:Screen)
 ((slate:screen-colormap
     :type slate:RGB-Colormap
     :reader slate:screen-colormap)))

(defclass slate:Single-Index-RGB-Screen (slate:RGB-Screen)
 ((slate:screen-colormap
     :type slate:Single-Index-RGB-Colormap
     :reader slate:screen-colormap)))

(defclass slate:Decomposed-Index-RGB-Screen (slate:RGB-Screen)
 ((slate:screen-colormap
     :type slate:Decomposed-Index-RGB-Colormap
     :reader slate:screen-colormap)))

Considering all possible combinations of the above classes (excluding RGB-Screen) gives us 6 basic abstract screen classes:

(defclass slate:Static-Gray-Screen (slate:Read-Only-Colormap-Screen
    slate:Monochrome-Screen)
 ((slate:screen-colormap
     :type slate:Static-Gray-Colormap
     :accessor slate:screen-colormap)))
(defclass slate:Static-Color-Screen (slate:Read-Only-Colormap-Screen slate:Single-Index-RGB-Screen)

(defclass slate:True-Color-Screen (slate:Read-Only-Colormap-Screen slate:Decomposed-Index-RGB-Screen)

(defclass slate:Gray-Scale-Screen (slate:Read-Write-Colormap-Screen slate:Monochrome-Screen)
  ((slate:screen-colormap :type slate:Gray-Scale-Colormap :accessor slate:screen-colormap)))

(defclass slate:Pseudo-Color-Screen (slate:Read-Write-Colormap-Screen slate:Single-Index-RGB-Screen)

(defclass slate:Direct-Color-Screen (slate:Read-Write-Colormap-Screen slate:Decomposed-Index-RGB-Screen)
  ((slate:screen-colormap :type slate:Direct-Colormap :accessor slate:screen-colormap)))

The 2 most common special cases of above:

(defclass slate:B&W-Screen (slate:Static-Gray-Screen)

(defclass slate:Byte-Pseudo-Color-Screen (slate:Pseudo-Color-Screen)

The instantiable screen classes are subclasses of the above, specialized for lisp platforms. Examples are:
- X11 (Franz Allegro Common Lisp CLX)
  (defclass slate:X-Screen
    (slate:Screen))

  (defclass slate:B&W-X-Screen
    (slate:B&W-Screen
     slate:X-Screen))

  (defclass slate:RGB-X-Screen
    (slate:Byte-Pseudo-Color-Screen
     slate:X-Screen))

- Symbolics
  (defclass slate:STV-Screen
    (slate:Screen))

  (defclass slate:B&W-STV-Screen
    (slate:B&W-Screen
     slate:STV-Screen))

  (defclass slate:RGB-STV-Screen
    (slate:Byte-Pseudo-Color-Screen
     slate:STV-Screen))

- Macintosh Allegro Common Lisp
  (defclass slate:Mac-Screen
    (slate:Screen))

  (defclass slate:B&W-Mac-Screen
    (slate:B&W-Screen
     slate:Mac-Screen))

  (defclass slate:Color-Mac-Screen
    (slate:Static-Color-Screen
     slate:Mac-Screen))

11 Slates

11.1 The Slate class
  (defclass slate:Slate (Standard-Object))
11.2 Invisible Slates

(defclass slate:Invisible-Slate (Slate))

Every class of (visible) slate is required to have a (at least one) corresponding invisible slate class. Instances of an invisible slate class accept all the same output operations as visible slates. (The reason for “invisible-slate” rather than “bitmap” is to encourage extension to Postscript devices.)

There will be methods for:

\begin{verbatim}
(copy-slate-rect invisible-slate invisible-rect visible-slate visible-point)
\end{verbatim}

for every legal invisible/visible slate class pair. copy-slate-rect will copy the contents of invisible-rect on invisible-slate to visible-slate, copying the origin of invisible-rect to visible-point.

11.3 Getting a slate to use

To get a slate to use, call make-slate or make-invisible-slate. The exact set of options allowed is not yet specified and may vary with the screen type.

\begin{verbatim}
(defun slate:make-slate (&rest options &key (screen (slate:default-screen))
                        (tools:type-check slate:Screen screen))

(defun slate:make-invisible-slate (&rest options &key (screen (default-screen))
                               (tools:type-check slate:Screen screen))
\end{verbatim}

11.4 Generalized Accessors

11.4.1 What screen is it on?

\begin{verbatim}
(defun slate:slate-screen (slate)
    (declare (type slate:Slate slate)
             (values slate:Screen)))
\end{verbatim}

11.4.2 Size and shape

At present, there is no support for changing the size of an existing slate, or changing where it is on the screen. The size and location of a slate may be changed by the underlying window system. To see how slates can respond to external reshape and resize events, see section 13.

The area covered by the slate, in its own coordinate system, is given by:
11.4.3 Name

(defgeneric slate:slate-name (slate)
  (declare (type slate:Slate slate)
    (values (or String Symbol))))

(defgeneric (setf slate:slate-name) (name slate)
  (declare (type (or String Symbol) name)
    (type slate:Slate slate)))

12 Drawing on Slates

Most drawing functions take (1) a pen argument that determines the "style" of drawing, that is, color, font, filling regions vs. outlining, and so on, (2) a slate argument that is the surface to be drawn on, and (3) one or more additional arguments that determine which pixels on the slate are to be altered.

No drawing operation will affect any pixels outside the slate's clipping region.

Most drawing operations come in both singular and plural forms (e.g. draw-point and draw-points). The reason for this is that it is often possible to implement the plural operations much more efficiently than repeated calls to the singular version. (X also has both singular and plural drawing operations.)

12.1 Clipping

All drawing on a slate is clipped to the slate's clipping region, which will always be a subset of the slate's slate-rect.

The clipping region can be changed by hand using:

(defun slate:slate-clipping-region (slate &key result)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Rect result)
  (values result))

(defun (setf slate:slate-clipping-region) (region slate)
  (tools:type-check g:Screen-Rect region)
  (tools:type-check Slate slate)
  (values region))
However, it’s preferable to bind it temporarily, using:

(defmacro with-clipping-region ((slate region) &body body))

In the future, clipping regions may be generalized to shapes other than simple g:Screen-Rect’s.

12.2 Flushing output

In most implementations, drawing operations will be buffered and you will get the best performance if you let the system decide exactly when to draw on the screen. However, sometimes it’s necessary to be able to guarantee that all pending drawing operations have completed, for example, to synchronize output with input events. For this purpose, we provide two operations for flushing the output buffers:

(defun finish-drawing (slate)
  (tools:type-check Slate slate)
  (values nil))

finish-drawing is analogous to the standard CL finish-output ([20], p. 579), that is, it attempts to ensure that all the drawing done on slate is completed before it returns (the returned value is always nil).

(defun force-drawing (slate)
  (tools:type-check Slate slate)
  (values nil))

force-drawing is analogous to the standard CL force-output ([20], p. 579), that is, it initiates the emptying of any internal buffers, but returns nil immediately, without waiting for the drawing to actually complete.

12.3 Pen position

The pen’s current position on the slate is used by relative drawing operations:

(defun slate:slate-point (slate &key result)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point result)
  (values result))

(defun (setf slate:slate-point) (pos slate)
(tools:type-check g:Screen-Point pos)
(tools:type-check slate:Slate slate)
(values g:Screen-Point))

(defun slate:move-to (slate pos)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point pos)
  (values g:Screen-Point))

(defun slate:move-by (slate dp)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Vector dp)
  (values g:Screen-Point))

12.4 Points

(defun slate:draw-point (pen slate point)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point point))

(defun slate:draw-points (pen slate point-list)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point-List point-list))

draw-point should draw a square of pixels, centered on point. The width of the square should be (line-style-thickness (pen-line-style pen)) draw-point ignores the tiling and filling properties of the pen.

12.5 Lines

(defun slate:draw-to (pen slate point)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point point))

draw-to draws a line of width (line-style-thickness (pen-line-style pen)) from the slate's current point to the destination point. Zero thickness is interpreted to mean draw the fastest possible visible line. draw-to ignores the tiling and filling properties of the pen.
(defun slate:draw-by (pen slate v)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Vector v))

draw-by draws a line of width (line-style-thickness (pen-line-style pen)) from the
slate's current point to the current point plus the displacement vector v. Zero thickness
is interpreted to mean draw the fastest possible visible line. draw-by ignores the tiling and
filling properties of the pen.

(defun slate:draw-line (pen slate p0 p1)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-point p0 p1))

draw-line draws a line of width (line-style-thickness (pen-line-style pen)) from
p0 to p1. Zero thickness is interpreted to mean draw the fastest possible visible line.
draw-line ignores the tiling and filling properties of the pen.

(defun slate:draw-lines (pen slate points0 points1)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point-List points0 points1))

draw-lines is the plural version of draw-line. It draws several (unconnected) lines with
a given pen on a given slate. This is different from draw-polyline!

(defun slate:draw-arrow (pen slate p0 p1)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point p0 p1))

(defun slate:draw-arrows (pen slate points0 points1)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point-List points0 points1))

draw-arrow draws an arrow whose tail is at p0 and head at p1.
(defun slate:drav-polyline (pen slate point-list)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point-List point-list)
  (%draw-polyline pen slate point-list))

(defun slate:drav-polylines (pen slate point-list-list)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check List point-list-list)
  (%draw-polylines pen slate point-list-list))

draw-polyline draws as piecewise linear curve interpolating the points in point-list.

12.6 Geometric Shapes

(defun slate:drav-rect (pen slate rect)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Rect rect))

(defun slate:drav-rects (pen slate rects)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Rect-List rects))

(defun clear (slate &key rect)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Rect rect))

(defun slate:drav-ellipse (pen slate rect)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Rect rect))

(defun slate:drav-ellipses (pen slate rects)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Rect-List rects))

draw-ellipse draws the largest ellipse that will fit within rect.
(defun slate:draw-circle (pen slate point r)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point point)
  (tools:type-check g:Positive-Screen-Coordinate r))

(defun slate:draw-circles (pen slate points rs)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point-List points)
  (tools:type-check g:Positive-Screen-Coordinate-List rs))

For convenience, we also provide draw-circle, which draws a circle centered on point of radius r.

(defun slate:draw-polygon (pen slate point-list)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check g:Screen-Point-List point-list))

(defun slate:draw-polygons (pen slate point-list-list)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check List point-list-list))

12.7 Characters and Strings

(defun slate:draw-character (pen slate char p)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check Character char)
  (tools:type-check g:Screen-Point p))

(defun slate:draw-characters (pen slate chars points)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check (List Character) chars)
  (tools:type-check g:Screen-Point-List points))

The exact placement of the character relative to point is described in detail in section 10.4 and illustrated in figure 1.
(defun slate:draw-string (pen slate string point)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check String string)
  (tools:type-check g:Screen-Point point))

As described in section 10.4, the text is drawn with the left baseline starting at point. When the drawing is done, the slate-point of the slate is at the baseline and the first pixel to the right of the last character.

(defun slate:draw-centered-string (pen slate string point)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check String string)
  (tools:type-check g:Screen-Point point))

The text is drawn with the center of the string's screen rect at point. When the drawing is done, the slate-point of the slate is point.

(defun slate:draw-vertical-string (pen slate string point)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check String string)
  (tools:type-check g:Screen-Point point))

The text is drawn with the point at the upper left hand corner, not the left baseline, aligned with point. When the drawing is done, the slate-point is just below the lower left corner of the last character's screen rect.

(defun slate:draw-centered-vertical-string (pen slate string point)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate slate)
  (tools:type-check String string)
  (tools:type-check g:Screen-Point point))

The text is drawn with the center of the string's screen rect at point. When the drawing is done, the slate-point of the slate is point.
12.8 Bitblt-like operations

(defun slate:copy-slate-rect (pen
    from-slate from-rect
    to-slate to-point)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate from-slate to-slate)
  (tools:type-check g:Screen-Rect from-rect)
  (tools:type-check g:Screen-Point to-point))

(defun slate:copy-slate-rects (pen
    from-slate from-rects
    to-slate to-points)
  (tools:type-check slate:Pen pen)
  (tools:type-check slate:Slate from-slate to-slate)
  (tools:type-check g:Screen-Rect-List from-rects)
  (tools:type-check g:Screen-Point-List to-points))

copy-slate-rect copies the source pixels in from-rect on from-slate to to-slate so that the rect of affected destination pixels has its origin at to-point. The source pixels are combined with the destination pixels using the (pen-boolean-operation pen). from-slate and to-slate must have the same slate-screen. (This may be relaxed in the future.)

13 Input

Input is probably the least well thought out part of Slate at present and can be expected to change significantly in the future. We provide three parallel approaches to dealing with user input.

- Event Driven
  An application can define functions to be called in response to certain input events in a particular slate, e.g., reshaping via the underlying window system.

- Polling
  An application can poll the state of the input devices (mouse and keyboard).

- High Level
  An application can request higher level forms of user input (e.g. menus).

13.1 Event driven input
(defgeneric slate:redraw-eventfn (slate)
  (declare (type slate:Slate slate)
    (values (Function (slate:Slate)))))

(defgeneric (setf slate:redraw-eventfn) (f slate)
  (declare (type (Function (slate:Slate)) f)
    (type slate:Slate slate)))

(defgeneric slate:reshape-eventfn (slate)
  (declare (type slate:Slate slate)
    (values (Function (slate:Slate)))))

(defgeneric (setf slate:reshape-eventfn) (f slate)
  (declare (type (Function (slate:Slate)) f)
    (type slate:Slate slate)))

(defgeneric slate:mouse-down-eventfn (slate)
  (declare (type slate:Slate slate:Slate)
    (values (Function (slate:Slate)))))

(defgeneric (setf slate:mouse-down-eventfn) (f slate)
  (declare (type (Function (slate:Slate)) f)
    (type slate:Slate slate)))

13.2 Polling Input Devices

slate:selected? tests whether the given slate is the “input focus” of the underlying window system. The other functions should be self explanatory.

(defgeneric slate:selected? (slate)
  (declare (type slate:Slate slate)
    (values (or T Nil))))

(defgeneric slate:mouse-down? (slate)
  (declare (type slate:Slate slate)
    (values (or T Nil))))

(defgeneric slate:mouse-up? (slate)
  (declare (type slate:Slate slate)
    (values (or T Nil))))

(defgeneric slate:mouse-screen-point (slate &key result)
13.3 High level input functions

menu takes a list of menu items, each of which is a list whose car is a string, and whose cadr is a function to be applied to the caddr of the list if the item is selected. If the machine-specific slate doesn't support menus, we provide a simple tty-oriented menu.

(defgeneric slate:menu (pen slate menu-list)
  (declare (type slate:Pen pen)
           (type slate:Slate slate)
           (type List menu-list)))

input-string prints out a prompt string, and allows the user to type in a string in response. If the machine-specific slate doesn't support special dialog boxes, we provide a simple tty-oriented interface.

(defgeneric slate:input-string (pen slate prompt-string)
  (declare (type slate:Pen pen)
           (type slate:Slate slate)
           (type String prompt-string)
           (values String)))

References


