Lessphic: A disposable, light-weight graphical environment for FoNC

$Id: canvas.html.in 330 2006-07-22 02:47:43Z piumarta $
last updated for idst-5.8 release

Contents:

1 Introduction

This document describes a graphical system whose only purpose is to bootstrap the FoNC programming environment. It deals only with the 2-D vector graphics subset of the system. A description of the 3-D support will be added later if necessary.

Please, please replace the system described herein at your earliest convenience with something infinitely better (maybe using generalised relationships and constraints rather than hard-wired structure).

I'm very sorry about the name, but it was the only obvious choice.

2 Geometry

Any `Number` can represent an angle. All angles are in radians, increasing anti-clockwise relative to the positive x-axis (which is at absolute angle zero).

The `Point` is the primitive unit of geometry and provides similar protocol to the points implemented by Smalltalk. The default coordinate system has *x* increasing to the right and *y* increasing upwards. For physical surfaces (such as windows) the origin is mapped to the bottom-left which places the entire surface in non-negative coordinates.

Note: Integral coordinates address the boundaries between pixels (*not* the pixels themselves). This means that rectangular fills aligned on integral coordinates will have sharp edges, but stroked single-pixel horizontal or vertical lines at integral coordinates will paint pixels on both sides of the boundary coincident with the line. To make sharp lines either use a line width of `2` or transform the coordinate system by `0.5@0.5` before using a line width of `1`.

3 Colour

`Colour` responds to the following messages to create a new object representing four channels of red, green, blue and alpha:

```
Colour withR: r G: g B: b A: a
Colour withH: h S: s B: b A: a
```

All four arguments are numbers between `0` and `1`. The final (alpha) argument is optional and defaults to `1` if omitted.

`Colour`s respond to at least the following messages:

`r`
`g`
`b`
`a`
: answer the receiver's red, green, blue or alpha channel, respectively.

`mixedWith: aColour ratio: aFraction`
: answers a new `Colour` formed by mixing `aFraction` of the receiver with `1 - aFraction` of `aColour`.

`lighter`
`darker`
: answer a new `Colour` formed by mixing the receiver in equal proportions with white or black, respectively.

`red`
`green`
`blue`
`yellow`
`magenta`
`cyan`
`black`
`darkGrey`
`grey`
`lightGrey`
`white`
: answer a predefined `Colour` with alpha of `1`. After the primary and secondary colours in the above list are 0, 25, 50, 75 and 100 percent greys.

4 Shapes

A `Shape` is (literally or conceptually) a collection of contours (often just line segments described by a set of vertices). `Shapes` implement protocol to enumerate, edit, and transform their vertices. Since they are a fundamental part of the graphical system all `Shapes` respond to the following messages:

`pathOn: aCanvas`
: appends a path to `aCanvas` (using the protocol described in the Canvas section) appropriate to the visual representation of the receiver.

`bounds`
: answers the smallest `Rectangle` (whose sides are parallel to the receiver's coordinate axes) encompassing the path

appropriate to the visual representation of the receiver.

`Polygon` provides simple generic `Shape` as a set of `Point`s representing vertices joined by line segments. Polygons can contain any number of vertices an are implicitly closed.

`Rectangle` provides an optimised representation for `Shape`s that are Polygons with four vertices and whose line segments are parallel to the coordinate axes; they store an *origin* and *corner* `Point`. A `Rectangle` is *normal* when the corner is not less than (below or to the left of) the origin.

5 Canvas

`aCanvas newPath`
Clears the current path (if any) and current position (if any) from the canvas.

`aCanvas moveTo: aPoint`
Moves the current position to `aPoint` without extending the current path.

`aCanvas moveBy: aPoint`
Moves the current position to the vector sum of the current position and `aPoint` without extending the current path.

`aCanvas lineTo: aPoint`
Extends the current path with a line segment from the current position to `aPoint`. The current position becomes `aPoint`.

`aCanvas lineBy: aPoint`
Extends the current path with a line segment from the current position to the vector sum of the current position and `aPoint`. The current position is incremented by `aPoint`.

`aCanvas rectangle: aRectangle`
Convenience method that extends the current path with a closed rectangle. The current position becomes the origin of `aRectangle`.

`aCanvas curveThrough: aPoint through: bPoint to: cPoint`
If there is no current position then the current position is set to `aPoint`. The current path is then extended with a cubic Bézier spline from the current position to `cPoint` with control points at `bPoint` and `cPoint`. The current position becomes `cPoint`.

`aCanvas arc: aPoint radius: r from: a1 to: a2`
`aCanvas arcNegative: aPoint radius: r from: a1 to: a2`
Extends the current path with a circular (in *user* space) arc, centred at `aPoint` with radius `r`, bounded by the radials `a1` and `a2` expressed in radians anticlockwise from the positive X axis. The first form sweeps the arc through increasing angles; the second form sweeps through decreasing angles. If there is a cuurent point before this operation then a line segment is first drawn from it to the first point on the arc's circumference. After the operation the current position becomes the final point on the arc's circumference.

`aCanvas setSource: pixelSource`
Sets the source of pixel values for subsequent drawing operations (`stroke`, `fill`) to `pixelSource`. The `pixelSource` can be any object mapping coordinates to pixel values. `Colour`s are already supported, answering a constant pixel value for all coordinates. Gradient fills and images are planned but not yet implemented.

`aCanvas setStrokeWidth: aNumber`
`aCanvas getStrokeWidth`
Sets or retrieves the line width currently in effect for `stroke` operations. The default stroke width is 2. (This ensures that vertical and horizontal lines with integral coordinates generate sharp lines with pixels that are fully-on, since integral coordinates address *boundaries between pixels* and not the pixels themselves.)

`aCanvas stroke`
Strokes current path according to the current line width. The current path and point are cleared from the canvas.

`aCanvas fill`

Closes the current path if necessary (by adding a line segment from the current position to the first point of the path) and then fills the interior of the path with the current source colour. The current path and point are cleared from the canvas.

`aCanvas setClipRectangle: aRectangle`
Sets a clipping rectangle within which all subsequent drawing takes place. Any existing clipping region is replaced. Note that this operation may (or may not) destroy the current path.

`aCanvas clearClipRectangle`
Clears the current clipping rectangle, if any.

`aCanvas identityMatrix`
Sets the current transformation to the identity matrix, causing user and device coordinates to coincide.

`aCanvas rotate: angle`
Rotates (by modifying the current transformation matrix) the user space coordinate system about the current origin by the given `angle`, measured in radians. Positive angles correspond to anticlockise rotation.

`aCanvas translate: aPoint`
Translates the user space coordinate system, moving the new origin to `aPoint` (relative to the old origin).

`aCanvas scale: aPoint`
Scales the user space coordinate system by `aPoint`.

`aCanvas userToDevice: aPoint`
Answers the coordinate in device space corresponding to the user space coordinate `aPoint`.

`aCanvas deviceToUser: aPoint`
Answers the coordinate in user space correspinding to the device coordinate `aPoint`.

`aCanvas save`
Pushes the current canvas state (transformation matrix, source colour, line width, current path and point) onto a per-canvas stack of saved states.

`aCanvas restore`
Restores a previously-saved canvas state (transformation matrix, source colour, line width, current path and point) by popping the topmost state from the per-canvas stack of saved states.

`aCanvas destroy`
Releases all resources associated with `aCanvas`. Note that these resources may or may not be 'weakly' associated with the canvas. A client cannot rely on garbage collection to automatically recover these resources and should assume that continually creating new canvases for some physical medium, without destroying every on of them in a timely fashion, *will* result in unbounded resource consumption and eventual failure due to starvation. (Garbage-collected systems that have support for finalisation might choose to send this message from within a finaliser.)

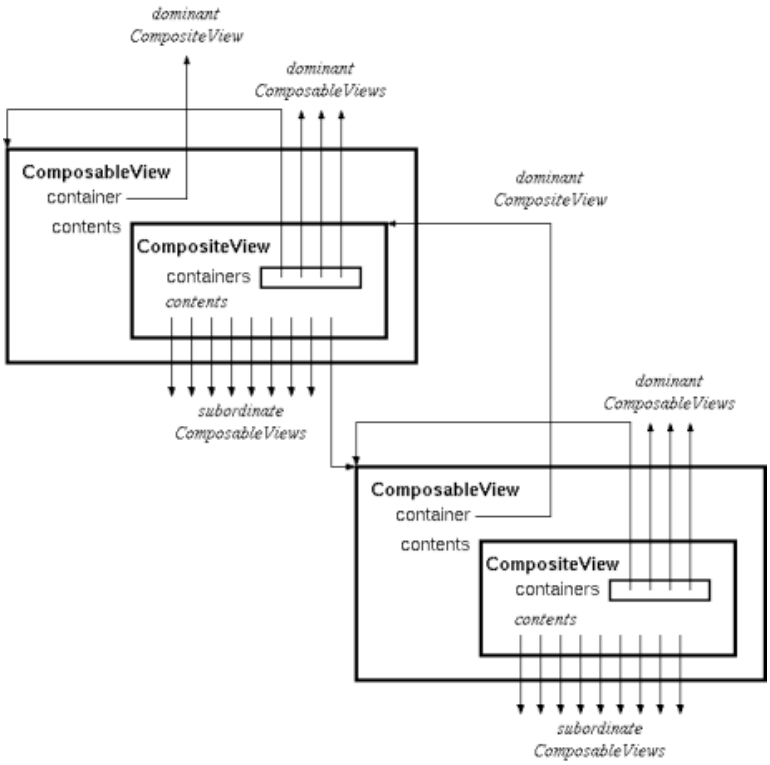
Note that canvases provide no facilities whatsoever for rendering text. It is the client's responsibility to decompose glyphs (representing character shapes within a typeface) into stroked and/or filled paths constructed from the move, line, curve and arc operations described above.

6 Views

The graphics system is ultimately concerned only with `Canvas`ses and `Shape`s. Interactive interface components do however impose some additional structure. This structure is provided by encapsulating `Shapes` whithin the nodes of a rooted graph of interconnected views.

6.1 ComposableView and CompositeView

`ComposableView` represents an edge in the graph, maintaining a one-to-one relationship between a 'dominant' `container` (a `CompositeView`) and a 'subordinate' `contents` collection (also a `CompositeView`) of composable views.



A `CompositeView` is the collection of 'subordinate' `ComposableViews` that are stored in some other 'dominant' `ComposableView`. It supports a many-to-many relationship between any number of 'dominant' `ComposableViews` (all sharing the same `CompositeView`) and indirectly its 'subordinate' `ComposableViews`).

Subordinate `ComposableViews` are added to a `CompositeView` by seding it `addFirst:` or `addLast:`. Sending `addFirst:` places the new view at the 'front' of the z-order, 'above' all other views; `addLast:` places the new view at the 'back', 'behind' the other views.

Both `CompositeView` and `ComposableView` respond to the following collection-like messages:

`addFirst: aComposableView`
: adds `aComposableView` to the receiver's subordinate views (in front of all others, if stacking order is significant).

`addLast: aComposableView`
: adds `aComposableView` to the receiver's subordinate views (behind of all others, if stacking order is significant).

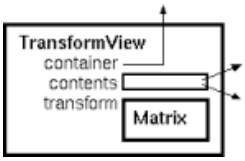
`add: aComposableView`
: is synonymous with `addFirst:`.

When sent to a `ComposableView` the above messages are forwarded immediately to the `CompositeView` that holds its subordinate views.

Note that `ComposableViews` are `Links` and `CompositeViews` are `LinkedLists`. A `ComposableView` can therefore obtain its predecessor (`prevLink`) and successor (`nextLink`) views directly, without recourse to its dominant `CompositeView`.

6.2 TransformView

A `TransformView` has no graphical representation. It is a kind of `ComposableView` and therefore contains other views. Its purpose is to transform the coordinate system of its contents relative to that of its container, according to its stored transformation `Matrix`. The transformation is reversible and coordinates can be propagated (with appropriate transformation) either from container to contents or from contents to container.



For example, when traversing the view graph to draw on a canvas the `TransformView` will transform the canvas' coordinate system before drawing each of its content views. In the other direction, when propagating geometric information (such as damage regions) from its contents up to its container, the `TransformView` transforms all coordinates by the inverse of its transformation `Matrix`.

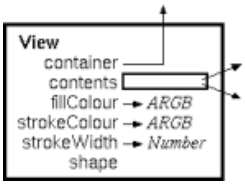
`TransformView` reponds to `new`; the resulting object can have any number of `ComposableView`s added to it. For convenience, `ComposableView` responds to `transformView` by creating a new `TransformView` and immediately adding itself. For example, the following two expressions have precisely the same effect:

```
(aView := TransformView new) add: subView.
aView := subView transformView.
```

Note that there is no *requirement* to use `TransformView`s. An application can choose to ignore them entirely and build a view graph in which all bounds are represented in global coordinates. This might, however, require new types of view to be defined to store the view's (global) origin (which would otherwise be supplied implicitly by a `TransformView`'s translation). It would also preclude the sharing of subgraphs to create duplicated graphical representations at different locations, scales or rotations.

6.3 View

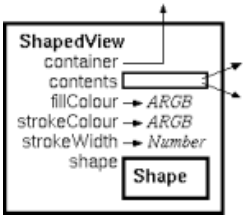
This is an abstract type storing the graphical characteristics that will be used to fill (`fillColour`) and/or stroke (`strokeColour` and `strokeWidth`) a path.



Concrete subtypes *must* implement `pathOn:` and `bounds` to describe the shape that must be drawn.

6.4 ShapedView

This is a kind of `View` that provides `bounds` and `pathOn:` trivially by forwarding them to a stored `Shape` (which might typically be a `Rectangle` or `Polygon`).



Other than responding to `pathOn:` and `bounds` (as must all `Shape`s) there are no constraints on the complexity of the `shape`. `ShapeView`s are kinds of `ComposableView` and can therefore contain subordinate `CompositeView`s.

`ShapedView` responds to the constructor `withShape: aShape`. They can also be created implicitly by sending any `Shape` the message `shapedView`. For example, the following two expressions have precisely the same effect:

```
aShape := ShapedView withShape: (0@0 corner: 100@100).
aShape := (0@0 corner: 100@100) shapedView.
```

Note that the shape can be anything from a simple geometric figure to a complex compound shape made up of many 'sub-shapes'. If it is a compound shape then any hierarchical (or other) structure within the shape is completely independent of the graph of views of which its `ShapedView` is part.

Note also that a `ShapedView` (or its stored `Shape`) is neither *required* (by convention) nor constrained (by clipping, for example) to draw strictly within its bounds. The bounds reported by a `ShapedView` are used only to determine if it needs to be redraw for a given damaged region. Drawing outside the reported bounds, while not forbidden, is however dangerous because of the risks of either not being asked to redraw (when in fact the graphical representatin has been damaged) or inadvertently drawing over the bounds of an unrelated view that will not be asked to redraw itself (and thus damaging its graphical representation 'permanently').

6.5 Drawing

All views implement the message `drawOn: aCanvas in: aRectangle`.

- `CompositeView` enumerates its subordinate views and forwards the message to any whose bounds intersect `aRectangle`.
- `TransformView` transforms `aRectangle` and then forwards the message, with the transformed rectangle, to its `CompositeView`.
- `View` tries to draw a graphical representation of itself on the canvas. If `fillColour` is not `nil` then the view sends itself `pathOn: aCanvas`, sets the canvas' source colour to its `fillColour` and then immediately fills the path on the canvas. It then forwards the `drawOn:in:` message to its subordinate `CompositeView`. Finally, if `strokeColour` is not `nil` then the view sends itself `pathOn: aCanvas`, sets the canvas' source colour to its `strokeColour` and then immediately strokes the path on the canvas. (The `pathOn:` method is not implemented by `View` and must be supplied by a concrete subtype of which the receiver is a member.)

`ShapedView` implements `pathOn:` by forwarding the message to its `shape`.

The `bounds` message is handled accordingly:

- `CompositeView` enumerates its subordinate views and builds the smallest `Rectangle` encompassing their individual bounds.
- `TransformView` applies the inverse of its transformation to the bounds answered by its subordinate `CompositeView`.
- `ShapedView` just forwards the message to its `shape`.

6.6 Properties

Each `View` maintains a set of named properties. These can be used to decorate the view graph with arbitrary per-object information. All `View`s respond to the following messages:

`propertyAt: aKey ifAbsent: errorBlock`
`propertyAt: aKey`
: answers the receiver's property named by `aKey` if it exists, otherwise answers the value of `errorBlock` (if specified) or `nil` (if `errorBlock` is not specified).

`propertyAt: aKey put: anObject`
: associates `anObject` with the receiver as a property named by `aKey`.

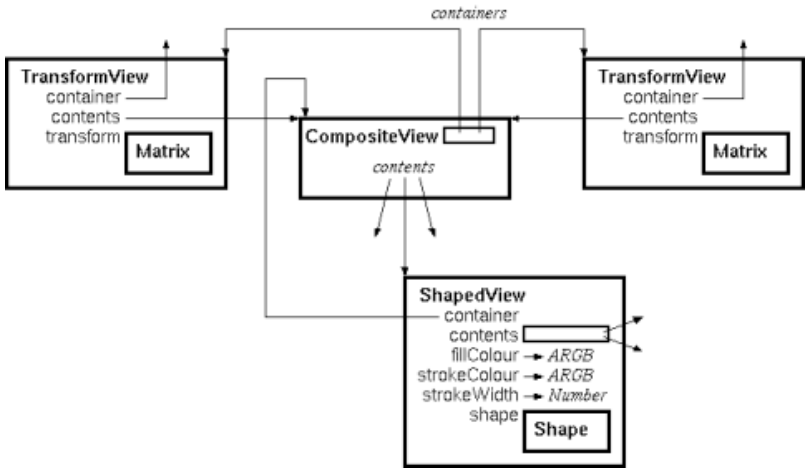
`removeProperty: aKey`
: removes any association in the receiver named by `aKey`. No error is signaled if `aKey` does not name a property of the receiver.

6.7 Discussion

In many graphics frameworks each node in the tree structure provides both a graphical representation (e.g., a background fill) and substructure (e.g., buttons, lines of text, etc.). In contrast, Lessphic deliberately separates graphical properties (`ComposableView`) and aggregation (`CompositeView`) into two distinct kinds of node within the graph. `ComposableView`s can provide graphical content and/or coordinate system transformations but they cannot aggregate multiple subordinate views into a more complex structure. `CompositeView`s aggregate many subordinate views into a more complex, logically single, structure but cannot (in and of themselves) perform any coordinate transforms on their contents (or directly provide graphical content).

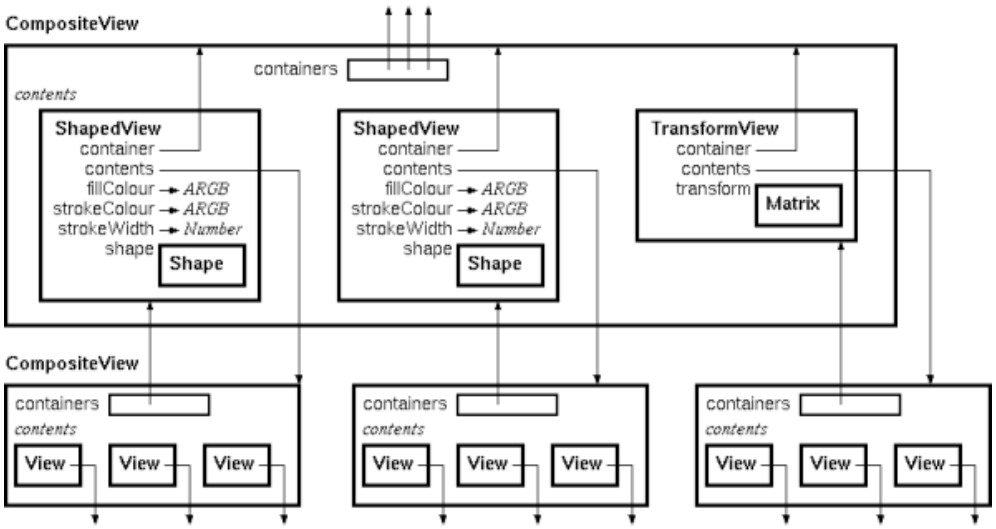
This was done to minimuse the confusion that usually arises when trying to share substructure between multiple dominant 'parent' views. For example, a user interface component will almost always be a `TransformView` on a `CompositeView` whose contents draw the various parts of the component. A layout algorithm places the component within the user interface by modifying the `TransformView`'s

matrix with apporpriate translations. Another `TransformView` can be created and made to share the same `CompositeView`.



The result is that two copies of a single structure will be visible. Each copy behaves (display, interaction, etc.) normally and neither is 'privileged' or distinguishable in any way from the other. Each will however have completely independent placement (and other transformations) applied. One obvious use for this is to create miniature views of larger components (or even entire windows) in which (unless deliberately disabled) display updates and user interactions are fully functional. Another obvious use is the convers: a 'magnifying glass' onto the contents of any view in which display update and user interaction work exactly as in the original (un-magnified) view.

It might sometimes help to turn the relationship between a `ComposableView` and its `CompositeView` (containing all of its subordinate `ComposableViews`) 'inside-out' and think about things like this:



(The leaves of the graph will therefore all be empty `CompositeViews`.)

7 Events

Events are represented by objects of the following types:

```
Event : Object ( context handled globalPosition localPosition )
  PointerEvent : Event ( state )
    PointerMotionEvent : PointerEvent ()
    ButtonEvent : PointerEvent ( button )
      PointerDownEvent : ButtonEvent ()
      PointerUpEvent : ButtonEvent ()
  KeyEvent : Event ( state key ucs4 )
    KeyDownEvent : KeyEvent ()
    KeyRepeatEvent : KeyEvent ()
```

```
KeyUpEvent : KeyEvent ()
```

The common members are set as follows for all events:

`context`
: is ignored by the trivial default event dispatch mechanism. A top-level window might set `context` to an object that globally manages events as part of the implementation of pointer grabs, focus control, etc.

`handled`
: is `nil` until an event has been handled. By convention `handled` is set to the object (usually a view) that took responsibility for the event.

`globalPosition`
: is a `Point`, in window coordinates, associated with the event.

`localPosition`
: is a `Point`, in view coordinates, associated with the event.

The type-specific members are set as follows:

`PointerEvent`
: The `state` contains an integer bitmap of modifier keys that were active at the time the event was generated.

`ButtonEvent`
: The `button` contains an integer identifying the particular button that was pressed or released.

`KeyEvent`
: The `state` is an integer bitmap of modifier keys that were active at the time the event was generated. The `key` contains a raw integer code identifying the particular key that was pressed, repeated or released; for control keys (enter, tab, escape, cursor movement, etc.) the `key` is encoded in ASCII. For printable characters and ASCII control keys `ucs4` is the ISO 10646 code point associated with the key, otherwise `nil`. For modifier keys (shift, control, etc.) the `key` will be set to the corresponding X11 KeySym (regardless of platform, and whose value is guaranteed to be outside the range of ASCII codes) and `ucs4` will be `nil`.

7.1 Event handling

Any `ComposableView` can elect to handle an event by (overriding and) answering non-nil to any of the following (event-specific) messages:

```
pointerMotionEvent: theEvent
pointerDownEvent: theEvent
pointerUpEvent: theEvent
keyDownEvent: theEvent
keyRepeatEvent: theEvent
keyUpEvent: theEvent
```

`ComposableView` provides default implementations that immediate answer `nil` in response to any of these messages. A view that handles an event in one of the above messages should answer non-nil; the answer will be stored as the `handled` propery of `theEvent` and the traversal of the view graph (see below) terminated.

The above messages are sent during a preorder traversal (frontmost first in z-order within `CompositeViews`) of the view graph, starting at the root, by (recursively) sending `handleEvent: theEvent at: aPoint` to each view. The locally-transformed position of `theEvent` is tracked in `aPoint`, updating the `localPosition` stored in the `theEvent` whenever the traversal passes a `TransformView`. (After updating `localPosition`, if necessary, `handleEvent:at:` sends `theEvent dispatchTo: self` which is overridden by each event subtype to double-dispatch one of the event-specific messages listed above back to the view.)

For convenience, a view (presumably the root) can be sent `handleEvent: theEvent` which will initiate the above recursive traversal in that view with `aPoint` initialised from the `globalPosition` stored in `theEvent`.

7.2 View-specific event handling

The above message-based event handling mechanism obviously does not distinguish between different views of the same type. Limited support for view-specific (per-instance) event handling is provided through the property mechanism (see Properties).

As the dispatch mechanism is traversing the graph, before the event is dispatched in a particualr `ComposasbleView` that view is checked to see if it satisfies two conditions:

1. the event's `localPosition` is within the view's `bounds`; and
2. the view has a property named by the type of event being dispatched.

If the view satisfies both of these conditions then the value of the property named by the event type must be a block expecting two arguments. The block is evaluated passing the view and the event (in that order) as arguments. If the block answers `non`-nil then the event has been handled; if the block answers `nil` then the dispatch mechanism continues as described in the previous section.

7.2.1 Example

A short example might help to illustrate these mechanisms. Consider a type `World` that is a kind of `View` attached to a top-level window. The `World` might set up its initial contents to include just a short text message:

```
World setup
[
    | text |
    self fillColour: Colour white.
    self add: (text := 'Hello, world' asCompositeView transformView translate: 100@100).
    text
        propertyAt: #pointerDownEvent
        put:        [:view :event | self startDragging: view at: event position].
]
```

The `World`'s `setup` method adds a block as a property to the `text` view named `pointerDownEvent`. When a `PointerDownEvent` being dispatched within the `World` reaches `text`, and that event's `localPosition` is within `text`'s `bounds`, then the block is evaluated. (Note that the `self` within the block refers to the `World`, not to the view that triggered the event.) The net effect is that when a mouse button is pressed 'over' the `text` view, the `World` is sent a `startDragging:at:` message with the `text` and the event's `position` as arguments:

```
World startDragging: textView at: textPosition
[
    self
        propertyAt: #pointerMotionEvent
        put: [:view :event |
            textView translate: textView translation + event position - textPosition.
            textPosition := event position.
            self redraw];
        propertyAt: #pointerUpEvent
        put: [:view :event |
            self
                removeProperty: #pointerMotionEvent;
                removeProperty: #pointerUpEvent]
]
```

The `World` responds by setting itself up to intercept all pointer motion events, changing the `textView`'s transformation to 'track' mouse movement in response to all `PointerMotionEvents`. When the pointer is released, the `World`'s `pointerUpEvent` block simply removes the tracking behaviour (the `pointerMotionEvent` and `pointerUpEvent` properties) from the `World`, placing it back in its initial state.

8 Typefaces, Fonts and Glyphs

First a few terms must be defined.

- A *family* is a single design for the visual representation of a set of printable characters. The family specifies the overall style of the font: times, helvetica, courier, etc.
- A *shape* is a variation in form within a family: roman, italic, etc.

- A *series* is a variation in weight within a family: light, bold, etc.
- A *typeface* is a particular combination of font family, shape and series.
- A *font* is a particular typeface at a fixed size: 10pt, 24pt, etc.
- A *glyph* is the shape of a single character within a particular font.

8.1 Typeface

A new `Typeface` is created from the name of its family with the message:

```
Typeface family: familyNameString
Typeface family: familyNameString ifAbsent: errorBlock
```

The `familyNameString` might be `'times'` or `'helvetica'`.

The following messages answer `Typefaces` with the indicated characteristics:

`Typeface serif`
: a member of a serifed family (each beam finishing with a decorative stroke) such as Times, Palatino or Century Schoolbook.

`Typeface sans`
: a member of a non-serifed family (each beam finishing 'cleanly') such as Helvetica, Avant Garde or Geneva.

`Typeface mono`
: a member of a monospaced family (all characters having the same `hAdvance`) such as Courier, Lucida Typewriter or Monaco.

Typical requests for `Typeface`s might therefore resemble

```
bodyFace := Typeface family: 'times' ifAbsent: [Typeface serif].
headingFace := Typeface family: 'helvetica' ifAbsent: [Typeface sans].
codeFace := Typeface family: 'courier' ifAbsent: [Typeface mono]
```

A `Typeface` is created with a default series and shape, usually 'medium-roman'. To create a `Typeface` in a particular family with a non-default shape and/or series, specify all three parameters at once:

```
myFace := Typeface
              family: 'times' series: 'bold' shape: 'roman'
              ifAbsent: [Typeface serif bold].
```

A `Typeface` responds to the messages `family`, `series` and `shape` with the names for those parameters that precisely describe the receiver.

Given a `Typeface`, another one differing only in family, series or shape can be created with the following messages:

`aTypeface family: familyName`
: where `familyName` can be `'serif'`, `'sans'`, `'mono'`, or the name of any locally-available family (dependent on your particular installation). Some common substitutions ('serif' for 'times', 'sans' for 'helvetica', etc.) will be made automatically if the exact typeface requested is unavailable.

`aTypeface series: seriesName`
: where `seriesName` can be `'light'`, `'medium'`, `'bold'` or `'black'`. Some common substitutions ('bold' for 'black', or any weight to its immediate neighbour) will be made automatically if the exact typeface requested is unavailable.

`Typeface shape: shapeName`
: where `shapeName` can be `'roman'`, `'italic'`, `'sloped'`, or the name of a shape specific to the particular family. (Modern typefaces often contain *faux italics* made from sloped roman glyphs and glibly labeled 'oblique' or 'slanted' rather than 'sloped'. All three names names are acceptable and will differentiate between typefaces if installed with one name rather than another, but in general they all mean 'sloped' are are treated as synomymous when making shape substitutions.) Some rational substitutions ('italic' for 'sloped' and vice versa, for example) will be made automatically if the exact typeface requested is unavailable.

A change of family, series or shape can be made by sending a `Typeface` one of the following convenience messages:

`serif`
`sans`
`mono`
: to change family,

`light`
`medium`
`bold`
`black`
: to change series, or

`roman`
`italic`
`sloped`
: to change shape.

For example:

```
titleFace := Typeface sans bold italic.
```

A `Typeface` responds to two additional messages:

`name`
: answers a `String` that encodes the family, series and shape of the receiver.

`metrics`
: answers the receiver's `FontMetrics`. (Note that the `FontMetrics` associated with a `TypeFace` are always presented in *unscaled* (integer) font units and are intended for sophisticated clients that need to determine the exact point size of `Font` that will satisfy some layout constraint. See FontMetrics below for more information on the contents of the `FontMetrics` object.)

`characterMap`
: answers the receiver's `CharacterMap`.

The prototype bound to the name `Typeface` is a fully-initialised `Typeface` object with default family, series and shape ('serif-medium-roman').

8.2 Font

A `Font` is created by fixing a `Typeface` at a particular point size.

```
titleFont := Typeface sans bold italic pointSize: 24.
```

A `Font` provides information about itself in response to the following messages:

`typeface`
: answers the `Typeface` from which the receiver was created.

`pointSize`
: answers the point size of the receiver.

`metrics`
: answers the receiver's `FontMetrics`.

A `Font` can be mutated into another related `Font` with the following messages:

`family: familyName`
: answers a `Font` similar to the receiver (same series, shape and point size) but whose typeface belongs to the given `familyName`.

`series: seriesName`
answers a `Font` similar to the receiver (same family, shape and point size) but whose typeface has a style corresponding to the given `seriesName`.

`shape: shapeName`
answers a `Font` similar to the receiver (same family, series and point size) but whose typeface has a style corresponding to the given `shapeName`.

`pointSize: newPointSize`
answers a `Font` similar to the receiver (same typeface) but scaled to the given `newPointSize`.

As with `Typeface`s a `Font` responds to the following convenience messages that change family, series or shape to one of the standard values: `serif`, `sans`, `mono`, `light`, `medium`, `bold`, `black`, `roman`, `italic`, and `sloped`.

The prototype bound to the name `Font` is a fully-initialised `Font` object: the default `Typeface` (see above) at 12 points.

8.3 Glyph

A `Glyph` can be obtained from a `Font` in one of two ways:

`aFont glyphAt: codePoint ifAbsent: errorBlock`
answers the glyph at the given code point (numeric index) or the value of `errorBlock` if the `codePoint` lies outside the legal range for the receiver. The first 127 code points typically coincide with ASCII. The first 256 code points typically coincide with the ISO8859-15 (aka Latin 9).

`aFont glyphNamed: glyphName ifAbsent: errorBlock`
answers the glyph with the given Unicode `glyphName` or the value of `errorBlock` if no glyph with the given name exists. (For example, the first three non-control characters, with code points 32 through 35, are named `'space'`, `'exclam'`, `'quotedbl'` and `'numbersign'`).

`aFont glyphAt: codePoint`
`aFont glyphNamed: glyphName`
answer as above unless the glyph does not exist within the receiver in which case they answer the glyph `#'.notdef'` (code point zero).

Given a `Glyph` you can find out where it came from by asking it for its `font`, `codePoint` and `name`.

`Glyph`s respond to the usual typeface and style changing messages:

`aGlyph font: newFont`
answers the glyph at the same code point as the receiver in the `newFont`.

`aGlyph family: familyName`
`aGlyph series: seriesName`
`aGlyph shape: shapeName`
answers the `Glyph` at the same code point as the receiver in a `Font` related to the receiver's `Font` (and at the same point size) but differing in family, series or shape.

`aGlyph pointSize: newPointSize`
answers a `Glyph` with the same code point and typeface as the receiver but from a `Font` scaled to `newPointSize`

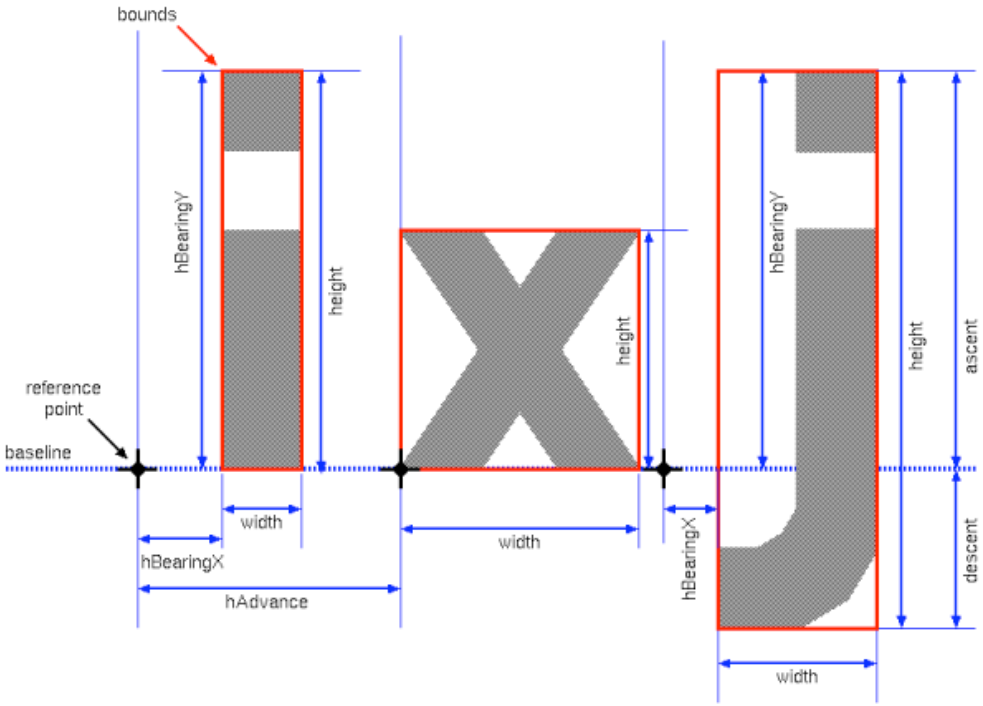
`Glyph`s are `Shape`s and respond to `bounds` and `pathOn:`. They can be attached to a `ShapedView`, (and respond appropriately to `shapedView`) just like any other `Shape`.

8.4 Metrics

Glyph and font metrics provide information needed for laying out text, determining the screen coordinates covered by a particular glyph when rendered, and so on.

8.4.1 Glyph metrics

Each `Glyph` provides information about the following properties:



`width`, `height` and `bounds`
: define the smallest rectangle that encompasses the painted area of the glyph.

`ascent`
: is the height of the glyph's ascender (the painted area above the baseline).

`descent`
: is the height of the glyph's descender (the painted area below the baseline).

`hBearingX`
: is the horizontal distance from the reference point to the left edge of the painted area of the glyph when layed out horizontally.

`hBearingY`
: is the vertical distance from the reference point to the top edge of the painted area of the glyph for horizontally layed-out text.

`hBearing`
: is a `Point` combining `hBearingX` and `hBearingY`.

`hAdvance`
: is the horizontal distance between the glyph's reference point and that of the next glyph on the line (ignoring kerning) when laying out text horizontally.

`vBearingX`
`vBearingY`
`vBearing`
`vAdvance`
: are the bearing and advance distances for the receiver when text is being layed out vertically (rather than horizontally).

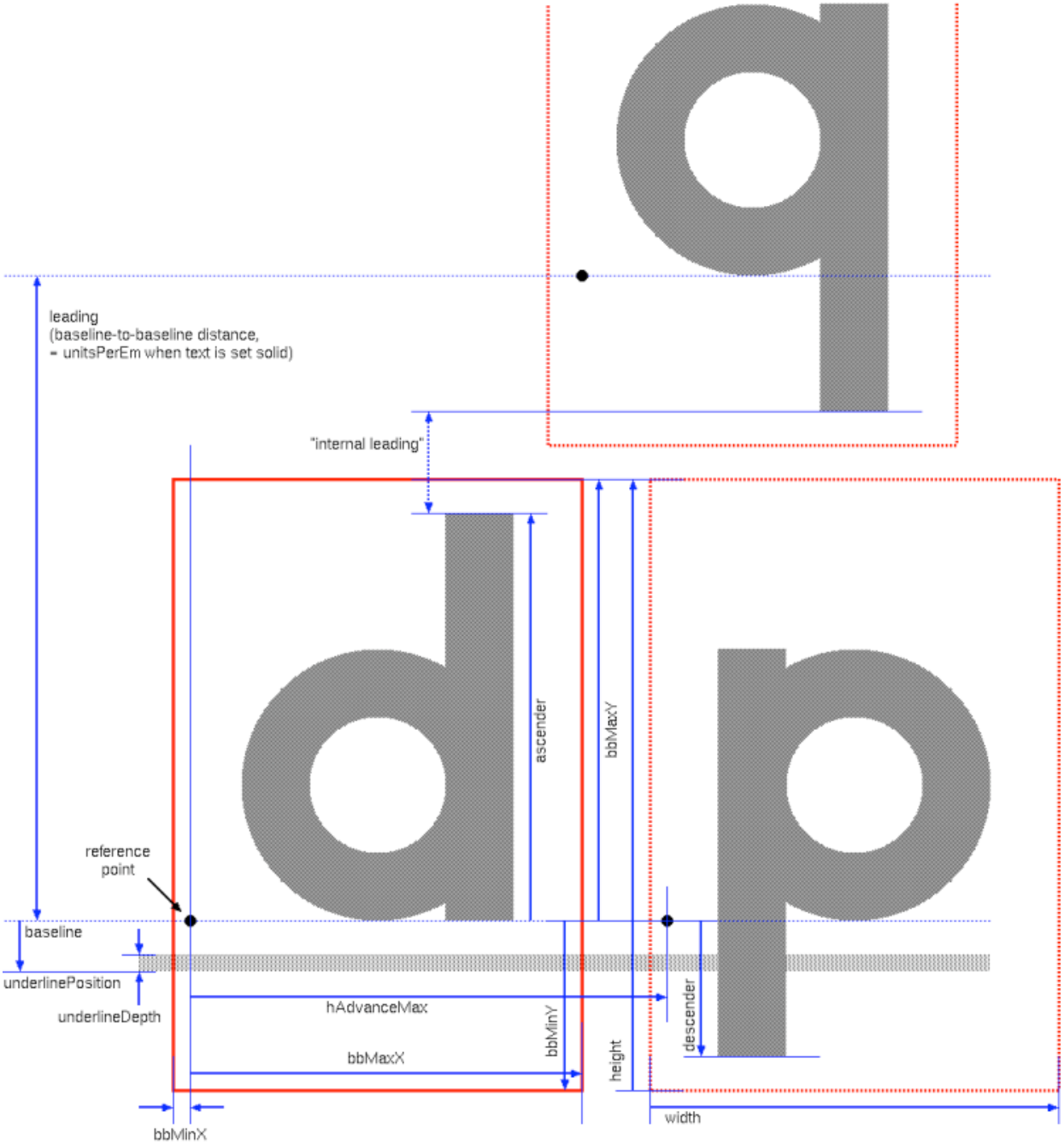
`GlyphMetrics` are expressed in points; there are 72 points in one inch.

Note that if dimensions are to be accurately represented then each `Canvas` must scale its coordinate system according to the physical resolution of the destination device before rendering `Glyphs`. For a typical LCD screen at 116 dpi its `Canvas` would scale the coordinate

system by a factor of 116/72 (approximately 1.61) before rendering a `Glyph`.

8.4.2 Font metrics

A `Font` provides information about maxima and minima of various metrics aggregated over all glyphs within it:



`size`
: is the number of glyphs within the font.

`unitsPerEm`
: is the length equal to the type size (one *em*) expressed in whichever units are relevant for the receiver: font units for a `Typeface`, and points for `Fonts` and `Glyphs`. (One em is normally the minimum distance between baselines, yielding lines that are 'set solid'. Any additional space between baselines is *leading*.)

`width` and `height`

are the `width` and `height` of the smallest rectangle that encompasses the bounding boxes of all glyphs in the font.

`bbMinX`, `bbMaxX`, `bbMinY` and `bbMaxY`
: are the horizontal and vertical limits, relative to the reference point, of the smallest rectangle that encompasses the bounding boxes of every glyph in the font. Note that `ascent` and `descent` will not necessarily be the same as `bbMaxY` and `bbMinY` since the latter might include space for any 'absolute minimum external leading' incorporated explicitly by the font designer for aesthetic reasons.

`ascender` and `descender`
: are the maximum height above the baseline of the tallest ascender, and the maximum depth below the baseline of the deepest descender, of all glyphs in the font.

`hAdvanceMax` and `vAdvanceMax`
: are the maximum `hAdvance` and `vAdvance` of all glyphs in the font.

A `Font` will also recommend the placement and thickness of a horizontal rule for underlining:

`underlinePosition`
: is the distance from the baseline to the bottom of the underline rule.

`underlineDepth`
: is the depth of the underline rule.

When associated with a `Font`, `FontMetrics` are expressed in points. There are 72 points in one inch.

When associated with a `Typeface`, `FontMetrics` are expressed in integral *font units*. The resolution is not fixed, but there are usually 2048 font units in one *em* (a length equal to the type size of a font). Metrics for `Fonts` and `Glyphs` are derived from those for their `Typeface` by scaling the latter's dimensions (in font units) to points for layout and rendering. This places a typeface-dependent upper bound on the resolution of all metrics, whether or not they have been scaled from font units to points.

9 Character maps

Each `Typeface` has an associated `CharacterMap` that associates glyph names with integer code points. It can be retrieved by sending `characterMap` to the `Typeface`.

A `CharacterMap` responds to the following messages:

`nameAtCodePoint: codePoint ifAbsent: errorBlock`
: answers the name (a `Symbol`) corresponding to the integer `codePoint`, or the value of `errorBlock` if the `codePoint` is not defined by the receiver.

`codePointAtName: name ifAbsent: errorBlock`
: answers the integer code point corresponding to the given `name` (a `String` or `Symbol`), or the value of `errorBlock` if the `name` is not known to the receiver.

`nameAtCodePoint: codePoint`
`codePointAtName: name`
: are as above, but return the name '.notdef' or codePoint zero (respectively) if the argument is not defined in the receiver. (By convention most typefaces place the glyph '.notdef' at codePoint zero, whose shape is either empty or the small rectangle often seen when trying to display an 'unprintable' character.)

`includesName: name`
`includesCodePoint: codePoint`
: answer non-`nil` if the argument is defined in the receiver.

Pre-defined `CharacterMaps` are provided via the following messages:

`CharacterMap iso8859_15`
: answers a map corresponding to the 8-bit ISO8859-15 (aka Latin-9) character set.

`CharacterMap default`

answers the platform (or locale) default map (currently hard-wired to ISO8859-15, because I'm a Western-European tyrant).

New `CharacterMap`s can be created and manipulated using the following messages:

`CharacterMap new`
: answers an empty `CharacterMap` ready to be populated with the following messages.

`aCharacterMap at: codePoint put: name`
: associates the integer `codePoint` with the given `name asSymbol`, and vice-versa, in the receiver.

`aCharacterMap addNames: nameArray`
: enumerates `nameArray` (which must contain only `Symbol`s or `String`s representing names and `SmallInteger`s representing code points) adding each name to the receiver's maps at successive code points. The first name is placed at code point zero. If a `SmallInteger` code point occurs in the array then the 'current code point' is set to its value, facilitating sparsely-populated name arrays and/or the construction of maps from multiple independent arrays each covering a particular range of characters.

`CharacterMap withNames: nameArray`
: is a convenience method that answers a new `CharacterMap` initialised from `nameArray` as described for `addNames:`.

`CharacterMap`s are sparse; given sufficient boredom a map covering the whole of Unicode could be produced.

10 Text

There is no 'special' type of view for dealing explicitly with text. A stored line of 'text' within a graph of views is structurally a `CompositeView` containing several `TransformView`s (as many as there are characters in the 'text') each of which contains one `ShapeView` whose shape happens to be a `Glyph`.

This is a huge simplification (and generalisation) compared to a system in which text is 'special', and might even seem inconsistent with the quantity and quality of information contained in glyph and font metrics. However, the intention is to provide the necessary information for high-quality text and page layout *without* dictating any mechanisms (or optimisations, neither space nor speed). The one mechanism that *is* provided for 'retained text' is the simplest one that can possibly work (even though far from the most space efficient).

For example: if, in the above structure, each `TransformView`'s transformation moves the origin to the right by the `hAdvance` of its predecessor's subordinate `Glyph` then a 'line of text' will be rendered. Preiodically resetting the horizontal component of the transformations will yield a 'paragraph'. Juggling the translations of `TransformView`s whose glyph is whitespace so that the rightmost `hAdvance`s (before resetting the horizontal translation) coincide with the available width will yield a filled and justified 'paragraph'.

That being said, a few convenience methods are nonetheless provided for the impatient and the lazy:

`sString asCompositeViewWithFont: aFont`
`aString asCompositeView`
: for a string of size *N* answers a `CompositeView` containing *N* `TransformView`s each of which contains a single `ShapeView` whose `fillColour` is black and whose `shape` is a `Glyph` from `aFont`. The code points of successive `Glyph`s correspond to the characters in `aString`. After the first `TransformView` (which has no transformation) each successive `TransformView` moves the origin right by the `hAdvance` of the `Glyph` subordinate to the preceding `TransformView` (in other words, the text is arranged in a line and ready to be rendered onto a light background). The second form (in which `aFont` is omitted) obtains `Glyph`s from the default `Font`.

`aCompositeView asString`
: answers a `String` whose contents are the `codePoint`s of any `Glyph`s within the receiver. (Non-`Glyph`s are discarded and therefore a `CompositeView` that contains only non-`Glyph` `Shape`s will answer this message with an empty `String`.) Note that

```
aString asCompositeView asString
```

: should always answer (a verbatim copy of) `aString`.

`aCanvas string: aString font: aFont`

`aCanvas string: aString`
: extends the current path of the receiver with the paths of the `Glyph`s in `aFont` whose code points correspond to successive characters in `aString`. The first `Glyph`'s reference point is placed at the origin. Succeeding reference points are translated along the positive x axis by the `hAdvance` of the preceding `Glyph`. No line breaking is attempted: carriage returns and line feeds in `aString` will be rendered like any other character (with typeface-dependent results). The second form (in which `aFont` is omitted) obtains `Glyph`s from the default `Font`.

11 Layout

The built-in minimalist layout mechanism has the following objectives:

- To be the simplest mechanism that can 'intelligently' position an arbitrary admixture of `Shape`s.
- To do *approximately* 'the right thing' when some (or all) of those `Shape`s are `Glyph`s (i.e., text). The result will not be publication quality but will be sufficient for trouble-free editing of simple text (code, etc.) and for constructing user interface components.
- To be easily and selectively replaceable by more sophisticated layout engines.

12 Resources

The FoNC mailing list: http://vpri.org/mailman/listinfo/fonc.