The **Glyph** User-Interface Library for Scala

Bernard Sufrin*

Draft of April 17, 2025

Abstract

Here we introduce the **Glyph** user-interface library for Scala. Its graphical features are geometrically compositional: rendered using a small subset of the facilities of Google's **Skia**[1], as delivered (in the JVM) by the **Skija** library.[2] Our motivation for developing the library was our frustration, over many years, with complex UI frameworks that impose a uniform style, and that make it hard to develop novel modes of interaction.[3]

What has guided this work is the idea that a rich set of interactive interface components can be composed using a small collection of combining forms, and a suitable collection of elementary visual and reactive components.

Instead of (just) providing a uniformly-styled "high level" toolkit, we have provided enough elements and combining forms for an innovative UI designer to experiment with, and a collection of implementations of conventional components (for example buttons, checkboxes, text input components, notebooks) to serve as models.

Code, along with several example programs to demonstrate principles, can be found in the public repository at

https://github.com/sufrin/Glyph.

---

*Emeritus Fellow: Department of Computer Science & Worcester College, Oxford; Tutor in Computer Science, Magdalen College, Oxford

[1] **Skia** is a highly portable 2-D graphics library used in the implementation of several browsers, including Chrome.

[2] The present prototype runs on Linux, Windows, OS/X (both x86 and Apple Mx processors)

[3] There is nothing wrong with uniform styling: but the cost of straying outside the styling envelope needs to diminish. An interface designer who doesn't mind learning a new language (Dart) and staying within its envelope might be rewarded by investigating Google's Flutter.

Contents

List of Figures

1 User Interfaces with Glyph

1.1 Introduction

A complete **Glyph** user interface (GUI) is specified as a tree of Glyph components, that may be *passive*, *active*, or *reactive*. Composite nodes of the tree consist of collections of components that share the same bounding box – being juxtaposed geometrically or temporally within it.

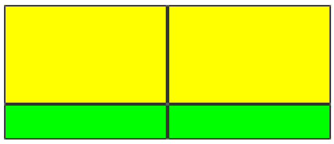
For example, suppose a's bounding box is 5 units wide and 3 units high; and that of b is the same width and 1 unit height – perhaps as painted[4] respectively yellow and green) by:

```
def a = FilledRect(5*units, 3*units, fg=yellow)
def b = FilledRect(5*units, 1*units, fg=green)
```

then

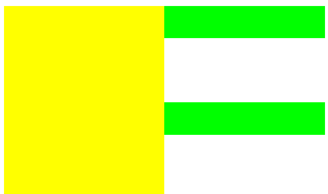
```
Row(Col(a, b), Col(a, b))
```

has a bounding box that is the horizontal catenation of 2 copies of the bounding box of Col(a, b); the latter has a bounding box that is the vertical catenation of the bounding boxes of a, b. The outcome will be drawn as:[5]



The height of a row is the maximum height of its components; its width is the sum of its components' widths; and a row is normally drawn with its components aligned along its top edge.

So exchanging Col and Row in the above leads to:



[4]A glyph's colour and visual texture is specified by its foregound and background: both determined by properties of the Brushes used to paint it.

[5]We have added thin frames around a and b to show their extents within the bounding box.

The glyph trees for the two images are:

```
Row                          Col
+--Col                       +--Row
|   +--  a                   |   +--a
|   +--  b                   |   +--b
+--Col                       +--Row
    +-- a                        +--a
    +-- b                        +--b
```

We will later show other ways of composing glyphs.

1.2 Reactive Glyphs and Focus

Reactive glyphs are the means by which user actions, such as mouse gestures and keystrokes, are coupled to the semantic actions of the application they control. As usual these actions can result in changes to the appearance of the interface.

Unless it is just a passive image in a window, one or more of the nodes in the glyph tree of a GUI will be a ReactiveGlyph – designed to respond to specific user actions such as gesturing at a window with mouse or trackpad, or typing a keystroke.

Interaction with the GUI in a window is mediated by its associated Interaction component, whose primary role is to determine which reactive glyph a user's action or a system-reported event is to be directed at, and to direct it there. To this end with each window is associated a keyboardFocus, and a mouseFocus variable – both of type Option[ReactiveGlyph]. These are managed by the EventHandler module of its associated Interaction, which implements the *Focus Protocol* described in detail in Appendix D.

Mouse Focus

Normally, a mouse event (mouse motion, mouse button press or release) is directed at the reactive glyph that has the mouse focus – this is almost always the reactive glyph within whose bounding box the mouse cursor is shown. When the mouse cursor strays outside the currently mouse-focussed glyph a GlyphLeave event is directed at the glyph, and we say that the mouse focus is *uncommitted*. The mouse focus stays uncommitted until the cursor moves into a(nother) reactive glyph. The focus is now committed to this glyph, which is informed of it by being sent a GlyphEnter event.

Keyboard Focus

A reactive glyph, such as one that is going to respond to typing, will normally "grab" the keyboard focus when it receives a GlyphEnter event, and *may* give it up when it receives a GlyphLeave event.[6] Any reactive component can acquire or be given the keyboard focus at any time; and can give it up or give it away at any time.[7]

Normally a keyboard event is directed at the reactive glyph that has the keyboard focus; when there is no such glyph, then it is first directed at the glyph that previously held the keyboard focus; and if there is no such glyph then the "orphan" event is (usually) ignored.[8]

[6] We write *may* because some glyphs do not require the mouse to be inside them in order to respond to the keyboard.

[7] Perhaps surprisingly, keyboard-focus-gained and keyboard-focus-lost events have not, so far been needed.

[8] We write *usually* because there is additional provision for catching and acting upon unfocussed keyboard events that can in principle be used to give some sort of feedback.

2 Glyphs

2.1 The Glyph Interface

The unit of graphical composition is the Glyph. All implemented Glyphs:

1. Define how they are to be “painted” on a Surface[9]

   Painting instructions always use a *local co-ordinate space* with origin at $(0, 0)$.

2. Define the Brushes to be used when they are painted: usually by specifying foreground and background brushes.

3. Define the diagonal of the rectangular bounding box they will occupy on the surface.

4. Define the location of the top left corner of the abovementioned bounding box relative to the origin of the bounding box of their parent in the glyph-tree. This is usually determined when the parent glyph is laid out; and happens for the first time during the composition of the parent.

5. Define a method that yields a *structurally identical* copy of the glyph: perhaps one that uses different brushes.

A purely passive graphical glyph may be elementary (simple), or composite. Simple glyphs are constructed by *Glyph factories*, many of which require no more than a specification of the diagonal of the bounding box of the graphic: its foreground and background brushes can be specified explicitly or by default. The actual location of a glyph when painted on its surface is usually defined by its location relative to its parent in the glyph tree.

Although a glyph type is usually *defined* by a Scala class, our API convention is that instances used in application GUIs are almost invariably *constructed* by one of the methods of the Scala companion object of its class. For example

```
FilledRect(w: Scalar, h: Scalar,      fg: Brush=···, bg: Brush=···)
Rect(w: Scalar, h: Scalar,            fg: Brush=···, bg: Brush=···)
FilledOval(w: Scalar, h: Scalar,      fg: Brush=···, bg: Brush=···)
Label(text: String, font: Font = ···fg: Brush=···, bg: Brush=···)
Polygon(w: Scalar, h: Scalar,
        fg: Brush=···, bg: Brush=···)(points: ···)
```

[9]A Surface implements the primitive methods that are used to paint shapes.

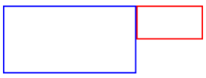
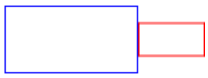
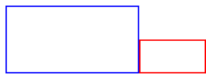
2.2 Glyph Composition

Natural Size

Composite glyphs are specialised instances of the Composite extension of Glyph, and are also usually constructed by *Glyph factories*. These include:

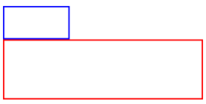
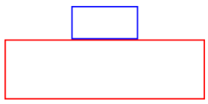
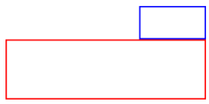
```
Row            Col
Concentric     OneOf
```

The NaturalSize.Row and NaturalSize.Col compositions are (almost) explainable by their names. Each has variants that explain the exact way in which components are aligned. Here are three differently-aligned rows of a pair of rectangles:[10]

`Row(align=Top)(b, r)` `Row(align=Mid)(b, r)` `Row(align=Bottom)(b, r)`

Here are three differently-aligned columns of a pair of rectangles:[11]

`Col(align=Left)(b, r)` `Col(align=Center)(b, r)` `Col(align=Right)(b, r)`

The row and column compositions described above yield glyphs that are "naturally" sized. Thus, for example, a row's width is the sum of its components' widths, and its height is the maximum of its components' heights. The complete API for these compositions is quite expressive:

```
Col(align:    Alignment  = Left,
    fg:       Brush      = nothing,
    bg:       Brush      = nothing,
    uniform:  Boolean    = false,
    frame:    Brush      = nothing,
    skip:     Scalar     = 0f)(glyphs)
```

[10] Row and Row(align=Top) mean the same. We will later meet Row(align=Baseline)(...).
[11] Col, and Col(align=Left) mean the same.

Constructs the vertical catenation of glyphs; its height is the sum of the glyphs' heights; its height is normally the sum of the glyphs' heights. The glyphs[12] are *laterally* aligned as follows in the row, as specified by align:

- Left – left edges of the bounding boxes aligned
- Right – right edges of the bounding boxes aligned
- Center – centre[13] lines of the bounding boxes aligned

If uniform, then the glyphs are all treated as if their height was the same as the maximal height of all the glyphs; and each glyph is treated as if its height were glyph.h+skip, thereby leaving a little vertical space between them in the column.

If frame is a nontrivial brush (ie with $alpha > 0$), then horizontal lines with foreground frame are drawn between glyphs.

The Row composition API is analogous.

```
Row(align:   VAlignment = Top,
    fg:      Brush      = nothing,
    bg:      Brush      = nothing,
    uniform: Boolean    = false,
    frame:   Brush      = nothing,
    skip:    Scalar     = 0f)(glyphs)
```

Constructs the horizontal catenation of glyphs; its height is the largest of the glyphs' heights; its width is normally the sum of the glyphs' widths.

The glyphs are *vertically* aligned as follows in the row, as specified by align:

- Top – tops of the bounding boxes aligned
- Bottom – bottoms of the bounding boxes aligned
- Mid – vertical midlines of the bounding boxes aligned
- Baseline – baselines are aligned when>0; otherwise as Bottom

[12]glyphs is typed either as a Seq[Glyph], or Glyph, Glyph* – one or more glyphs.
[13]Yes, I know! Dr. Webster has a lot to answer for.

Fixed Size

There are also fixed-size row (and column) compositions, whose row width and column height can be declared in advance, and that allow "expandable" spaces as components. Here we see a couple of examples:

These were constructed by:

```
import FixedSize·Space·fill
FixedSize·Row(350f, bg=grey)(fill, redR, blueR, redR)
FixedSize·Row(350f, bg=grey)(redR, fill, blueR, fill, redR)
```

with

```
def redR:  Glyph = FilledRect(50f, 25f, fg=red)
def blueR: Glyph = FilledRect(100f, 25, fg=blue)
```

The fill function is a horizontal/vertical "stretchable" filler of nominal width 10 units and stretchability 1.0.

The APIs for FixedSize.Row and FixedSize.Col are analogous.

```
Row(width: Scalar,
    align: VAlignment = Top,
    fg:    Brush      = nothing,
    bg:    Brush      = nothing)(glyphs)
```

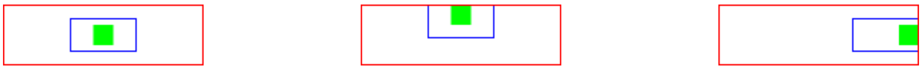
Constructs the horizontal catenation of glyphs, *vertically* aligned in the row, as specified by align. Its height is the largest of the glyphs' heights; its width is normally the larger of width and the sum of the glyphs' widths. In the former case stretchable spaces are stretched (in proportion to their horizontal stretchability) so that the bounding box is filled.

Concentric Overlays

Below we show three of the possible Concentric[14] compositions of a point and a pair of rectangles; respectively:

```
Concentric(rowAlign=Mid, colAlign=Center)(p,b,r)
Concentric(rowAlign=Mid, colAlign=Right)(p,b,r)
Concentric(rowAlign=Top, colAlign=Center)(p,b,r)
```

[14]Yes, Virginia, we probably should have called this Overlay!



The vertical and horizontal aspects of their alignments are, in general, specified independently; but there are shorthands for common cases. Here we could have written:

```
Concentric·Center(p,b,r)
Concentric·Top(p,b,r)
Concentric·Right(p,b,r)
```

Grid/Table organization

It can also be helpful to organize glyph sequences in tabular form. Several methods for doing this are provided by NaturalSize.Grid – all invoked by first specifying various aspects of the table.

```
Grid(fg:     Brush  = nothing,
     bg:     Brush  = nothing,
     padx:   Scalar = 0,
     pady:   Scalar = 0,
     width:  Scalar = 0,
     height: Scalar = 0)
```

The main *methods* of a Grid are:

```
grid  (glyphs: Seq[Glyph] or Glyph, Glyph*): Composite
table(glyphs: Seq[Glyph] or Glyph, Glyph*): Composite
rows  (glyphs: Seq[Glyph] or Glyph, Glyph*): Composite
cols  (glyphs: Seq[Glyph] or Glyph, Glyph*): Composite
```

The parameters width and height are used to interpret glyphs as follows:[15]

1. if there's a width parameter then glyphs is interpreted as a catenation of rows of the given width
2. if there's a height parameter then glyphs is interpreted as a catenation of columns of the given height
3. if there are neither width nor height parameters, then: glyphs is interpreted as width were ceiling(sqrt(glyphs.length))

- The grid method presents a grid of *identically-dimensioned* cells, each large enough for any of the glyphs. As a shorthand for this Grid(...)(data) is equivalent to Grid(...).grid(data)
- The table method presents a grid in which all cells in each column have the same horizontal dimension, and all cells in each row have the same vertical dimension.
- The rows method presents a grid of row data with columns of uniform horizontal dimension, whilst analogously cols presents a grid of column data with rows of uniform vertical dimension.

[15] In this section width means "number of columns"; and height means "number of rows.

- The grid and table methods can also supply width, and/or height parameters before their normal glyphs parameters: if they appear these take precedence over the corresponding parameters to Grid

Figures 1, 2 and 3 illustrate several ways of using the Grid methods grid, and table.

Individual glyphs in a grid can be made to fit their cell by various methods, including shifting in various directions within the cell, scaling to the size of the cell, and (the default) simply enlarging it to fit. Figure 3 illustrates annotations that can be made to individual glyphs that define how they are made to fit the cells they will inhabit. The data used here is the same as earlier, save that (except in the first grid) the 4th cell is a label of the form s"cellfit($how)" where how is one of the CellFit.Method constants:

```
case object ShiftNorth     extends Method // ··· to the top edge
case object ShiftNorthWest extends Method // ··· to the top left corner
case object ShiftWest      extends Method // ··· to the left edge ···
case object ShiftSouthWest extends Method // ··· to the bottom left
···
case object ShiftSouth     extends Method // ··· to the bottom edge ···
case object ShiftSouthEast extends Method // ··· to the bottom right ···
case object ShiftEast      extends Method // ··· to the right edge ···
case object ShiftNorthEast extends Method // ··· to the top right ···
case object Stretch        extends Method // scaled up to fit the cell
case object Enlarge        extends Method // centred in the cell
```

In each case the grid is composed by:

```
Grid(fg = red(width=0), padx=10, pady=10)·grid(width=3)(data)
```

`1.scaled(0.8)`
`1000.scaled(0.8)`
`1000000.scaled(0.8)`
`1.scaled(1.0)`
`1000.scaled(1.0)`
`1000000.scaled(1.0)`
`1.scaled(1.5)`
`1000.scaled(1.5)`
`1000000.scaled(1.5)`

`1.scaled(0.8)`
`1000.scaled(0.8)`
`1000000.scaled(0.8)`
`1.scaled(1.0)`
`1000.scaled(1.0)`
`1000000.scaled(1.0)`
`1.scaled(1.5)`
`1000.scaled(1.5)`
`1000000.scaled(1.5)`

Figure 1: Tables made with $\mathsf{.grid(width = 1)(data)}$ and $\mathsf{.rows(width = 1)(data)}$

.grid(width=3)(data) -- row data as uniform size cells

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	1000.scaled(1.0)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

.grid(height=3)(data) -- col data as uniform size cells

1.scaled(0.8)	1.scaled(1.0)	1.scaled(1.5)
1000.scaled(0.8)	1000.scaled(1.0)	1000.scaled(1.5)
1000000.scaled(0.8)	1000000.scaled(1.0)	1000000.scaled(1.5)

.rows(width=3)(data) -- row data in uniform width columns

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	1000.scaled(1.0)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

.cols(height=3)(data) -- col data in uniform height rows

1.scaled(0.8)	1.scaled(1.0)	1.scaled(1.5)
1000.scaled(0.8)	1000.scaled(1.0)	1000.scaled(1.5)
1000000.scaled(0.8)	1000000.scaled(1.0)	1000000.scaled(1.5)

.table(width=3)(data) -- row data as minimal width/height cols/rows

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	1000.scaled(1.0)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

.table(height=3)(data) -- col data as minimal width/height cols/rows

1.scaled(0.8)	1.scaled(1.0)	1.scaled(1.5)
1000.scaled(0.8)	1000.scaled(1.0)	1000.scaled(1.5)
1000000.scaled(0.8)	1000000.scaled(1.0)	1000000.scaled(1.5)

Figure 2: Table presentations of a grid

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	1000.scaled(1.0)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	fitToCell(Stretch)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	fitToCell(ShiftNorth)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	fitToCell(ShiftSouthWest)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	fitToCell(ShiftSouth)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	fitToCell(ShiftSouthEast)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

1.scaled(0.8)	1000.scaled(0.8)	1000000.scaled(0.8)
1.scaled(1.0)	fitToCell(ShiftEast)	1000000.scaled(1.0)
1.scaled(1.5)	1000.scaled(1.5)	1000000.scaled(1.5)

Figure 3: Individual cell placement with fitToCell

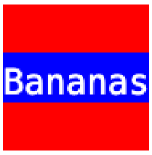
OneOf: Temporal Alternations of Glyphs

The OneOf composition is used primarily in the implementation of dynamically "paged" interfaces[16] or in the presentation of alternate captions on reactive glyphs.[17]. It constructs a glyph whose appearance is chosen dynamically from a palette of component glyphs. Its bounding box is the union of the bounding boxes of its component glyphs; and it shows only one of them at a time, namely the one selected most recently by its select method, or by one of its next() or prev() methods.

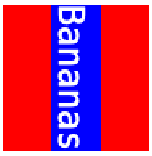
Thus the definitions[18]

```
def bananas = Label("Bananas", fg=white, bg=blue)
val oneof: OneOf =
    OneOf(bg = red)( bananas.rotated(0)
                   , bananas.rotated(1)
                   , bananas.rotated(2)
                   , bananas.rotated(3)
                   )
```

gives rise to the glyph that is initially drawn as:



and after the execution of oneof.next() is drawn as:



Subsequent invocations of oneof.next() will select successive components for drawing, and an invocation of oneof.select(n) will select its nth (modulo 4) component for drawing.[19]

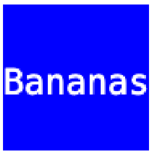
Had oneof been specified without a background brush, then the background would have been one of the backgrounds of its maximal (by area) components.

[16] Such as those provided in the Book() API.

[17] Such as those used in the Checkbox and TextToggle APIs

[18] See "Glyphs are mutable" (A.1) for an explanation of why the label is defined as a function not just as a value.

[19] Only the currently selected component of a OneOf is considered in the search for a reactive glyph to handle a user gesture or keystroke. See Appendix D for details.


Bananas

2.3 Brushes

A brush applies “paint” to a surface. The most important of its characteristics are its width (aka strokeWidth), and the colour of the paint it will apply. But it also has “shape”, in the sense that corners painted with it may be rounded, squared, mitered, *etc*; as well as having many other definable characteristics, including a human-readable name.

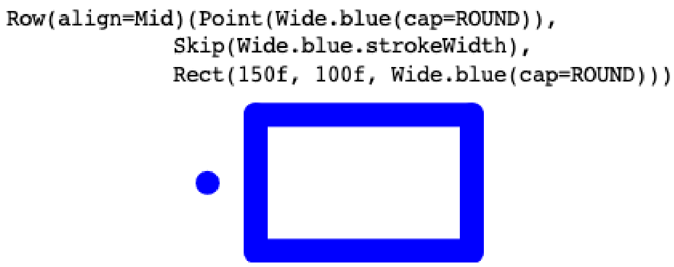
For example, here are definitions of the blue and red brushes used while preparing this paper.[20]

```
val blue = Brush("blue")(color=0xFF0000ff, width=1.0f, cap=ROUND)
val red  = Brush("red") (color=0xFFFF0000, width=1.0f, cap=ROUND)
```

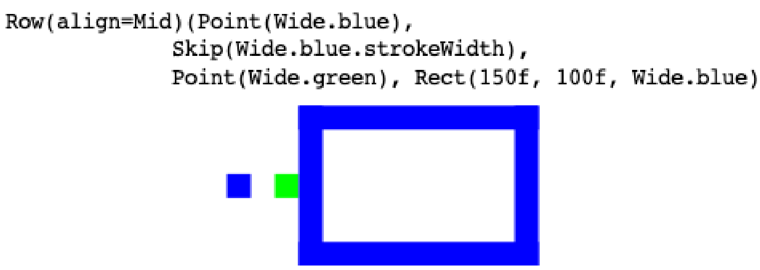
and **val** Wide.blue was defined as:

```
Brush("Wide.blue")(color = 0xFF0000ff,
                   width = 15.0f,
                   cap   = SQUARE)
```

The following two images show some brush properties in action. Notice the rounded corners of the first rectangle, and that the points have all the characteristics of the brushes they were drawn with.



Glyph eg1d



Glyph eg1e

[20]Colour is specified here by a 32-bit/4-byte integer – usually written as a hexadecimal constant. The first byte specifies its *alpha* – which is analogous to opacity or covering power; the second, third, and fourth bytes specify its red, green, and blue components.

In fact the library incorporates an embedded domain-specific language of brushes: some expressions denote new brushes or variants of existing brushes; and others denote commands that affect existing brushes.[21] For example, the Wide.blue brush could have been defined as a variant of blue:

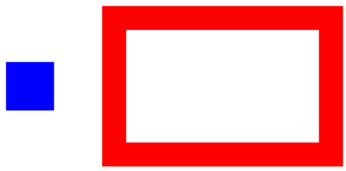
```
blue(width=15.0f, name="Wide.Blue", cap=SQUARE)
```

Brushes are cheap to build, and it is straightforward to define them *ad-hoc* while building a glyph.[22]

For example:

```
Row(align=Mid)(
    Point(Wide.blue(width=2*Wide.blue.strokeWidth, cap=BUTT)),
    Skip(2*Wide.blue.strokeWidth),
    Rect(150f, 100f, Brush("")(color=red.color,
                                  width=Wide.blue.strokeWidth, cap=SQUARE)))
```

denotes the glyph



DefaultBrushes

The Brushes module defines several predefined names for colours, as well as a constructor that supports their specification as strings. Some examples are:[23]

```
Brushes("0xFFFF0000/3/ROUND")      // red brush; width 3; round cap
Brushes("0xFFFFDD00/3/SQUARE")     // yellow brush; width 3; square cap
Brushes("red/3/ROUND")             // red brush; width 3; round cap
Brushes("red/3/ROUND-5-15")        // as above, but dashed
Brushes("red/3/ROUND-2-5")         // as above, but dotted
Brushes("green/10~4~2")            // a wobbly green brush of width 10
Brushes("nothing")                 // a transparent brush (no pigment)
```

[21] See appendix A.2 for an explanation and examples of the latter.

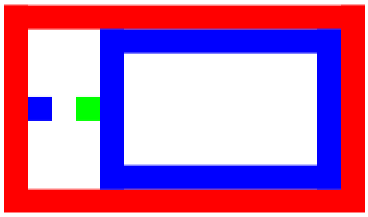
[22] Implementation details of the brush language may be of interest to the Scala programmer who likes the idea of "notationally sugared" APIs.

[23] See 4.2 **Paint Effects** for explanation of "wobbly", "dashed" and "dotted".

3 Glyph Transformers

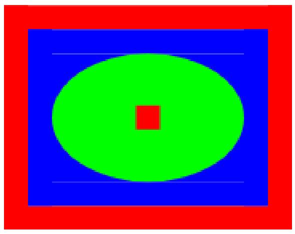
Glyph transformers are used to derive glyphs from simpler glyphs. Almost all transformers are provided as intrinsic methods of all glyphs. For example:

```
eg1e.framed(Wide.red)
```



Glyph
eg1f

```
Concentric(FilledOval(120f, 80f, green), Point(Wide.red))
            .framed(fg=Wide.blue, bg=Wide.blue)
            .framed(fg=Wide.red)
```



Glyph
eg1g

It's important to understand that, as a matter of policy, transforming a glyph, g is an "algebraic" operation that denotes a tree in which g is embedded: it no more makes a "new" copy of g than (for example) the successor *succ* n of a number n makes a "new" copy of n.[24]

Intrinsic glyph transformers include:

```
def scaled(scale: Scale): Glyph
def enlarged(delta: Scalar, ···): Glyph
def enlargedTo(w: Scalar, h: Scalar, ···): Glyph
def enlargedBy(w: Scalar, h: Scalar, ···): Glyph
def rotated(quadrants: Int, ···): Glyph
def turned(degrees: Scalar, circular: Boolean, ···): Glyph
def skewed(skewX: Scalar, skewY: Scalar, ···): Glyph
def mirrored(leftRight: Boolean, topBottom: Boolean, ···): Glyph
```

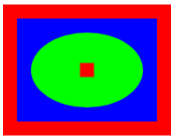
[24]In light of the first prototype library implementing glyphs mutably there is an argument against this policy because it could lead to inadvertent sharing, with effects such as those described in Appendix A.1. A future prototype is envisaged in which glyphs are no longer mutable, and the policy will be irrelevant.

```scala
def framed(fg: Brush = Glyphs.Framed.defaultFG,
           bg: Brush = Glyphs.Framed.defaultBG): Glyph
def roundFramed(fg: Brush = Glyphs.Framed.defaultFG,
                bg: Brush = Glyphs.Framed.defaultBG): Glyph
def edged(fg: Brush = Glyphs.Framed.defaultFG,
          bg: Brush = Glyphs.Framed.defaultBG): Glyph // same as framed
def shaded(enlarge: Scalar = 0.25f,
           delta: Scalar = 8f,
           down: Boolean=false, ...): Glyph
def beside(g: Glyph, align: VAlignment=Mid): Glyph
def above(g: Glyph, align: Alignment=Center): Glyph
```

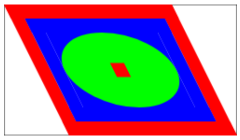
In the above signatures, the abbreviation $\cdots$ stands for the declaration that 'fg' and 'bg' parameters be inherited from the glyph being transformed, unless otherwise specified. For example, the full signature of mirrored is:

```scala
def mirrored(leftRight: Boolean, topBottom: Boolean,
             fg: Brush = thisGlyph.fg,
             bg: Brush = thisGlyph.bg): Glyph
```

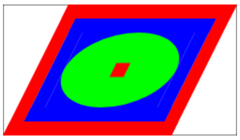
Figure 3 shows a few more examples of their use:



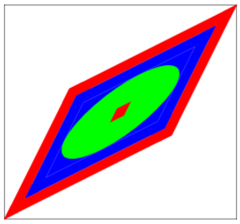
g



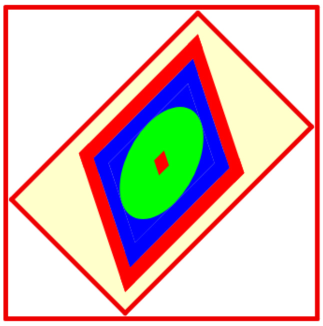
g·skewed(0·5,0)·framed()



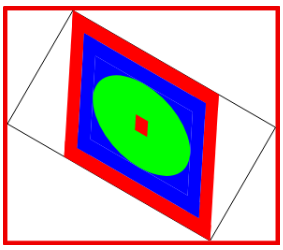
g·skewed(−0·5, 0)·framed()



g·skewed(−0·5, −0·5)·framed()



```
g·skewed(0·5f, 0f)·enlarged(27, bg=yellow(alpha=0·2f))
                  ·framed(redFrame)
                  ·turned(45f)
                  ·mirrored(leftRight=true, topBottom = false)
                  ·framed(fg=redFrame)
```



g·skewed(0·5f, 0·0f)·framed()·turned(30f)·framed(fg=redFrame)

Figure 4: Intrinsic glyph transformers in use

4 Polygonal Glyphs

4.1 Specification

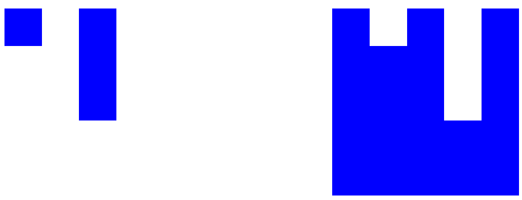
Polygons (open or closed) are specified by giving the coordinates of the relevant vertices in addition to the diagonal of the bounding box they will be drawn in.[25] They have several forms of constructor: all are features of Polygon or FilledPolygon . A *rough* guide to deciding on what part of the bounding box of a filled polygon gets filled is that a point is inside the polygon (therefore filled) if a line from it to some arbitrarily chosen far away point crosses one of the lines between adjacent vertices an odd number of times; it is outside if the number is even.[26]

Compare the two red FilledPolygons with vertices respectively given by

```
FilledPolygon(200, 200, fg = red)((0,   0), (200, 200),
                                  (200, 0), (0,   200))

FilledPolygon(200, 200, fg = red)((200, 200), (0, 0),
                                  (200, 0),   (0, 200))
```

Now compare the two blue FilledPolygons with vertices given by

```
FilledPolygon(100, 100, fg = blue)(
  (0,0), (20, 0), (20, 20), (40, 20), (40, 0),
  (60,0), (60, 60), (80, 60), (80, 0), (100,0))

FilledPolygon(100, 100, fg = blue)(
 ((100,0), (100, 100), (0, 100), (0,0),
  (20, 0), (20, 20), (40, 20), (40, 0),
  (60,0), (60, 60), (80, 60), (80, 0), (100,0))
```



[25] A closed polygon is one whose last vertex is the same as its first.

[26] The "far away point" should be chosen so that the line is not parallel to one of the edges. This is rarely difficult.

4.2 Paint Effects

Interesting effects are possible when paints are equipped with path-effects, or are blurred.

Path Effects

Using brushes with path effects to draw frames can give aesthetically interesting results. We provide intrinsic Brush-transforms for a couple of these, namely:

```
brush·sliced(segLength: Scalar, maxDisplacement: Scalar, ···): Brush
brush·dashed(on_0, off_0, on_1 , off_1 , ···): Brush
```

The former yields a brush whose paths are "sliced" into segments of the specified length; segment endpoints are displaced from the path by a random amount limited by the given maximum. If it appears, the third integer parameter is the seed for the generation of random numbers that determine the displacements.

The latter yields a brush that draws paths that are dotted/dashed with on (visible) segments of the given length(s), and off (transparent) segments of the given length.

In the following sequence of examples we compute the vertices of a 7-pointed regular star (a "stargon"), then construct a "wobbly" paintbrush (blueish) that is used to render the star: first as a filled polygon then as an open polygon.

First we define generators for filled and non-filled stargon glyphs of specific colours:

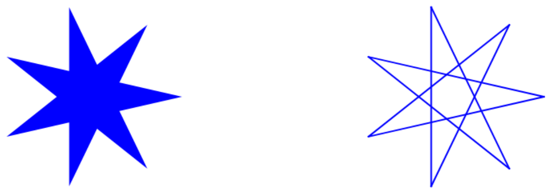
```
def filledStargon(n: Int, fg: Brush): Glyph =
    FilledPolygon·$(256, 256, fg, nothing)(regularStarPath(n))

def nonFilledStargon(n: Int, fg: Brush): Glyph =
    Polygon·$(256, 256, fg, nothing)(regularStarPath(n))
```

Both use a generator for the vertices of the paths used by the Polygon constructors to make star glyphs: R is the length of edges of the star, with centroid at (C, C).[27]

[27]Exercise: this algorithm can be understood by thinking of the points on the path as the successive places a tortoise will stop when at each stage it moves by R in the direction *heading*, then turns through *theta* – stopping when it has made $n + 1$ moves. Find out what happens if n is even, explain why, and suggest an alternative termination condition that yields stargons of *some* order for $n \geq 2$.

```
def regularStarPath(n: Int): Seq[(Scalar, Scalar)] = {
    val C: Scalar = 128.0f
    val R: Scalar = 115f
    val star = new ArrayBuffer[(Scalar, Scalar)]
    val theta = PI - PI/n
    star += (((C + R, C)))
    for {i ← 0 until n} {
      val a = theta * i
      star += (((C + R * Math.cos(a)).toFloat, (C + R * Math.sin(a)).toFloat))
    }
    star += ((C + R, C))
    star.toSeq
  }
```

Rendered straightforwardly with fg=blue the filled and open stargons are:

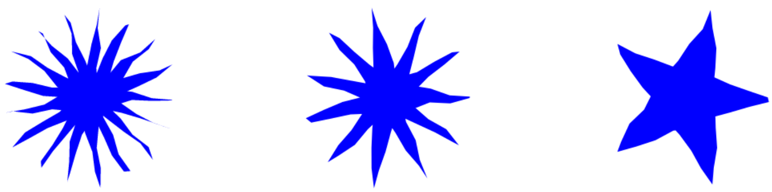


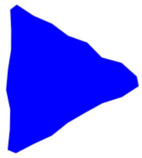
The blueish: Brush is the same color as blue, but wider: it has "wobbly" edges specified by a path effect described by sliced.

```
  val blueish =
        blue(width = blue.strokeWidth*2).sliced(25.0f, 4.0f, 1)
```

Rendered in blueish, the stargons are:

All this works for stars of arbitrary odd arity: here are stars with $n = 17, 11, 5, 3$:





But for $n = 4, 6, 8$, and other even numbers, the generator doesn't work.[28]

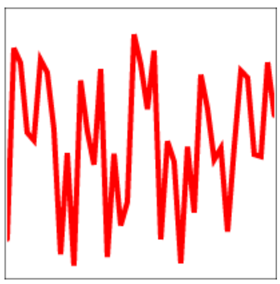
Framing with sliced and dashed paints

The sliced modifier can also be used to yield exotic-looking frames. For example:

```
Label("...").enlarged(20)
            .framed(fg = black(width = 2f).sliced(2.5f, 5f))
            .enlarged(10))
```

It's easier to see the effect of the displacement limit when the path is a straight line. Here's

```
Polygon(200, 200,
        fg = red(width = 4f, cap = ROUND).sliced(5f, 100f, 15)
)((0, 100), (200, 100)).enlarged(4).framed())
```



The dashed modifier gives rise to dashed/dotted lines, for example:

[28]Try them, then see if you can explain why.

```
Label("···").enlarged(20)
            .framed(fg = black(width = 2f).dashed(15f, 5f))
            .enlarged(10))
```

Many other effects are available from **Skia** *via* **Skija**.

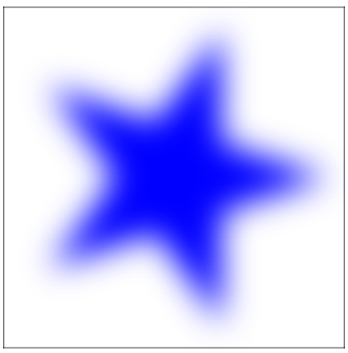
Blurred Paint

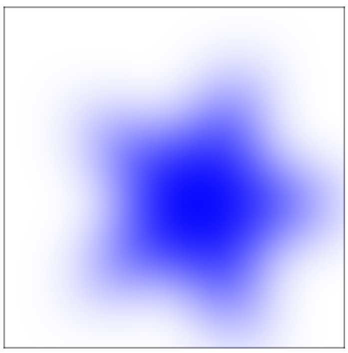
The intrinsic Brush-transformer blurred yields a brush that blurs all the filled glyphs that it paints. When brush: Brush

```
brush.blurred(blur: Scalar, dx: Scalar=0f, dy: Scalar=0f): Brush
```

yields a blurred brush of the same colour as brush. When used on *filled* glyphs, it blurs their outline, and can shift the blurred outline relative to their origin (by (dx, dy)).

Below we show the effects of painting a filled star with blue.blurred(24f) and with blue.blurred(48f, 20f, 20f). Both are framed(): notice that the latter is displaced by $(20, 20)$ within the natural bounding box indicated by the frame.





A Mutability of Glyphs and Brushes

A.1 Glyphs are mutable

In the present prototype implementation it is *essential to understand that a glyph is mutable*: in particular that its location and parent features are changed as it is incorporated in its glyph tree by its parent glyph.

This happens exactly once per glyph, and in order to use a glyph more than once it necessary to copy it. Fortunately this is straightforward: each form of glyph has a copy method that yields a fresh[29] structurally identical ("deep") copy. The copy method has fg, bg arguments that specify the foreground and background brushes of the copy; and these are defaulted to those of the glyph being copied.

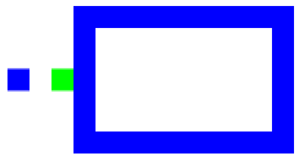
As a convenience, the Glyph may be "applied" with the same result:

```
def copy (fg: Brush=this.fg, bg: Brush=this.bg): Glyph
def apply(fg: Brush=this.fg, bg: Brush=this.bg): Glyph =
    this.copy(fg, bg)
```

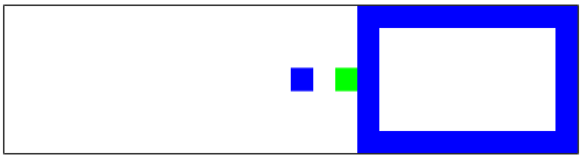
When a glyph is used twice without copying, the results are rarely what was intended. For example, here we show the outcome of using a glyph value a second time in a tree without copying. When egc is defined by

```
val egc: Glyph =
        Row.centered(Point(Wide.blue),
                     Skip(Wide.blue.strokeWidth),
                     Point(Wide.green),
                     Rect(150f, 100f, Wide.blue))
```

it denotes a glyph drawn as



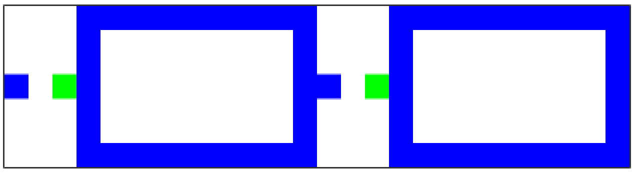
and the expression Row(bg=white)(egc, egc) denotes a glyph drawn as



[29] *ie.* without having set location, parent.

What is happening is that egc's location relative to its parent in the glyph tree is set twice by the Row compositor; and the second setting is the one used during drawing.[30]

When the glyph is copied before both uses all is well: Row(egc(), egc()) is drawn as[31]



A.2 Brushes are mutable

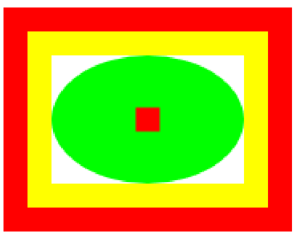
In the present prototype implementation the attributes of brushes can be changed dynamically, and the effects of these changes *apply retrospectively to every glyph that ever incorporated them.* Although this feature is not intended to be used frequently, it can occasionally be useful: for example using a brush on part of a GUI that indicates state by changing its colour.

Brushes have "chained" methods that can be used to change their attributes; these have the same names as the attributes, for example:

```
def strokeWidth(i: Float):        Brush = { ···; this }
def color(i: Int):                 Brush = { ···; this }
def cap(cap: PaintStrokeCap):      Brush = { ···; this }
```

For example, here's what eg1g looks like in a context where the colour of the wide blue Brush has been changed retrospectively to yellow by the command:

```
Wide·blue·color(0xFFFFFF00)
```



[30] That this can occur *without warning at compile time* is a defect in the design of the library to which there are a multiplicity of potential solutions: we are considering them at the time of writing. But for the moment we advise copying as a matter of course: it's not computationally very expensive.

[31] In fact if it's not going to be used again only one copy is needed.

The mutation methods return the glyph from which they are invoked, so to change colour, width, and cap in the same command one could write:

```
Wide.blue.color(0xFFFFFF00).strokeWidth(50).cap(ROUND)
```

or, recalling that Scala methods can also be used as infix operators, one could have written:

```
Wide.blue color 0xFFFFFF00 strokeWidth 50 cap ROUND
```

It is probably unwise to change a stroke width dynamically in such a way as to increase the size of a bounding box beyond its original size, for that would invalidate an important assumption made during glyph composition, namely that the sizes of component glyphs' bounding boxes do not increase after composition. This assumption is invalid for glyphs such as those built by `.framed(...)` whose bounding boxes are partly determined by the width of brushes used in them. Decreasing a stroke width dynamically does not violate this assumption.

B Text Glyphs and Fonts

Here we discuss the API for generating simple text glyphs. The glyphXML package provides versatile methods for composing composite glyphs that may include paragraphed text.

The factory for simple text glyphs is:

```
Text(text:      String,
     font:      Font,
     fg:        Brush   = defaultFG,
     bg:        Brush   = defaultBG,
     transient: Boolean = false)
```

It yields a glyph representing the string text in the font font. The foreground of the text is coloured with the brush fg brush, and its background with the brush bg.[32]

The factory for fonts is FontFamily. It has several methods, but the one we shall use here is:

```
FontFamily(family: String, style: String, size: Scalar): Font
```

It yields a Font made from the given font family, in the named style, at the given (point) size.[33]

In the examples that follow we have defined:

```
val Courier       = FontFamily("Courier", "normal", 24)
val CourierItalic = FontFamily("Courier", "italic", 24)
val Menlo         = FontFamily("Menlo", "normal", 20)
val MenloBold     = FontFamily("Menlo", "bold", 16)
val MenloTiny     = FontFamily("Menlo", "normal", 12)
```

Text("A text", Courier).framed(redFrame)

Text("A text", MenloBold).framed(redFrame)

[32]The detailed implementations of non-transient texts are represented uniquely, and this can save a good deal of space.

[33]The detailed implementations of fonts are represented uniquely; and although the Font type is opaque, a unique identifier indicating how a font that was constructed by FontFamily is available as: FontFamily.fontID(font).

```
Text("A text", CourierItalic).framed(redFrame)
```

A text

When texts are placed in a row whose alignment is BaseLine, their baselines will coincide.

```
Row(align=Baseline)(...)
```

As you can see **baselines coincide** even in tiny fonts

But they don't coincide in a row whose alignment is Top or Mid, for example:

```
Row(align=Top)(...)
```

As you can see **baselines do not coincide** especially in tiny fonts

C The glyphXML notation

The glyphXML notation is a dialect of **xml**, whose elements are translated to glyphs. It was originally designed to support the composition of textual glyphs, but has developed beyond that in ways that will be seen from perusal of the many example programs in which it is used.[34]

The following straightforward example starts by declaring a common attribute for <p> elements, then defines three glyphs.

```
import glyphXML.Language._
implicit val style: StyleSheet = StyleSheet()
val declare: Glyph = <ATTRIBUTES key="tag:p" enlarged="20px"/>

val gxml1: Glyph =
    <p width="20em" align="left" frame="red/2" rotated="2">
      Hello world
    </p>

val gxml2: Glyph =
    <p width="20em" align="left" frame="black/3-5-5">
       Hello world! Is this the very best you can do?
    </p>

val gxml3: Glyph =
  <div width="27em" align="justify">
    <p>
      The <b>brain</b> in <i>pain</i> feels with_out
      ex_ception as if it is going
      down an uninhibitedly con_vol_uted drain.
    </p>
    <p>
      But when not in pain, it is some_what sensitive to <bi>rain.</bi>
    </p>
    <p textForeground="red">
       Whatever else you do, try to remember that not_with_stand_ing that
       <b>glyphXML</b> may some_what resemble HTML, it is a completely
       different lang_uage.
    </p>
  </div>
```

[34] Its convenience in use depends largely on the fact that in Scala 2 programs **xml** syntax can be included directly; and to a lesser extent on the use of implicit functions to transform **glyphXML** elements to glyphs when such elements appear in a context requiring a glyph. We have not yet introduced StyleSheets, so although style(sheet) specifications play an important role in the materialization of glyphXML expressions as Glyphs, we will not explore that role in detail here.

Notice the various forms of decoration/framing that are specified for these two examples.

gxml1: Glyph

Hello world

gxml2: Glyph

Hello world! Is this the very best you can do?

Notice that the hyphenation in this example appears at discretionary points: indicated by the presence of underscores.

gxml3: Glyph

The **brain** in *pain* feels with-out exception as if it is going down an uninhibitedly con-voluted drain.

But when not in pain, it is somewhat sensitive to ***rain.***

Whatever else you do, try to remember that notwithstand-ing that **glyphXML** may some-what resemble HTML, it is a completely different lang-uage.

D The Focus Protocol

The windows of an application's GUI each have two foci: each is either undefined or associated with a reactive glyph.

- The window's keyboardFocus is the reactive glyph that is expected to handle keyboard events within that window. Reactive glyphs usually co-operate to manage it.
- The window's mouseFocus is the reactive glyph that is expected to handle mouse events, such as those arising from movements, button presses and releases, and from mouse wheel events. Such events may also originate on a trackpad, touchscreen, *etc*.

The (mouse) focus protocol is designed to ensure that events arising from the manipulation of the mouse (or other pointing device) get directed to an appropriate reactive glyph. The essence of the protocol is described below. The specification of "try to locate a reactive glyph..." is at D.4.

D.1 MouseMove events

(a) If mouseFocus is defined, then

  i If the mouse location is within the focussed glyph, direct the event to that glyph.

  ii If mouse location is not within the focussed glyph, then direct a GlyphLeave event to that glyph, and set mouseFocus to None.

(b) If mouseFocus is not defined, then try to locate a reactive glyph that contains the mouse location.

  i If there is such a reactive glyph, direct a GlyphEnter event to it, and set mouseFocus to it.

  ii If there is no such reactive glyph, ignore the event.

D.2 MouseButton events

(a) If mouseFocus is defined, then

i If the mouse location is within the focussed glyph, direct the event to that glyph.

ii If mouse location is not within the focussed glyph, then just set mouseFocus to None.

(b) If mouseFocus is not defined, then try to locate a reactive glyph that contains the mouse location.

i If there is such a reactive glyph, direct a GlyphEnter event to it, and set mouseFocus to it.

ii If there is no such reactive glyph, ignore the event.

D.3 MouseScroll events

(a) If mouseFocus is defined, then direct the event to the focussed glyph.

(b) If mouseFocus is not defined, then ignore the event.[35]

In effect the protocol defined above means that a reactive glyph becomes "aware" that it has the focus when the mouse moves into it; and becomes "aware" that it no longer has the focus when the mouse moves out of it. Although it would be straightforward to change the protocol so that fewer events are ignored, we believe that in many cases the response of a glyph to an event ignored in the protocol would be to ... ignore it.

D.4 Locating Reactive Glyphs

Recall that the glyphs comprising a GUI all have their own (0,0)-origin coordinate system, and that each glyph is located relative to its parent in the glyph tree as its parent is being laid out. This approach makes it straightforward to implement geometric transforms on glyphs, such as the rotations, scalings, and skewings described briefly earlier – a transformed glyph is displayed by applying the transform before displaying the untransformed glyph.

The search for a reactive glyph that contains a location is conducted by first searching the glyph tree for the glyph that most closely contains that location, then finding its nearest reactive parent in the tree (usually, but not always, the same glyph).

[35] **We are currently considering adding a third locus, namely scrollFocus.**

The search for the glyph that most closely contains a location is conducted top-down in the glyph tree. Composite glyphs that don't themselves contain the location are not searched, but those that do contain the location are searched for more specific components. For the moment we consider that this algorithm is adequately efficient, but if necessary glyph trees could be indexed straightforwardly.

On the other hand, experience has shown that the algorithm is not efficient enough to use during mouse motion to decide whether the mouse location is still within the focussed glyph. This is because mouse motion events are generated at high frequency as the mouse traverses a window. So instead of using the algorithm, we annotate each reactive glyph with the inverse of the (constant) transform that was (last) used to display it, then use this inverse to map the (absolute) mouse location back to the coordinate system of the reactive glyph itself.

D.5 Locating Glyphs in the presence of Overlays

Menus and dialogues are managed by the module overlaydialogue.Dialogue which provides a collection (possibly empty) of *overlays* per window: each of these is specified as a glyph tree with a few additional properties.

The overlays are organised as a stack of GUI "layers" drawn topmost-last: each appearing on top of its predecessor in the stack, and all appearing on top of the main GUI tree. There is also a collection of named "decorations", each defined by a glyph tree. These are drawn in no particular order after the main GUI and the overlay stack.

The algorithm to locate a glyph that contains the mouse is designed to find a currently-visible glyph containing the mouse in:

i the topmost layer of the stack, or in

ii a decoration, or in

iii a currently-visible glyph in the application's main GUI tree.

When the topmost layer of the stack is "modal" (*ie* represents a menu or a modal dialogue), then only (i) and (ii) above are considered.

The net effect is that glyphs in the main interface that are *completely hidden* by the topmost layer of the stack will not be selected during a mouse-focus transfer. Normally, as far as a button is concerned, if you can't see it *all* then you can't press it.[36]

[36] In the exceptional situation of *loose hiding* being enabled for the topmost overlay then if you can see some of a button then you can press it.

E Anatomy of a Simple Reactive Glyph

Here we give an account of the structure of the reactive glyph class Coloured-Button. The appearance of such buttons is specified by a single glyph. The foreground (or the background) colour of the glyph changes when the mouse hovers over it, and when a button is pressed (but not yet released) within it.If its background flag is true, then it's the background colour of the button that is changed.

It inherits the features of a GenericButton that deal with the details of mouse-motion and button-clicks. We shall discuss these later: the main thing to understand now is that the state of an active, non-disabled button is captured by:

```
var hovered:  Boolean
var pressed:  Boolean
```

The former is true if and only if the (mouse) pointer is within the button.[37] The latter is true if and only if the button has been pressed, but not yet released, within the bounding box of the button. When a button is released within the button, its react method is invoked.[38]

A button can be programmatically disabled or made inactive:

```
var disabled: Boolean
var inactive: Boolean
```

The first part of the definition is straightforward: the constructor takes the glyph used to specify the button's appearance when neither hovered nor pressed. The brushes down and hover specify the foreground colour of the glyph when hovered and pressed, and when just hovered. If the colour of the background is to be changed on state changes, then background is set.

```
class ColourButton(
  appearance:      Glyph,
  down:            Brush,
  hover:           Brush,
  val background:  Boolean,
  val react:       Reaction) extends GenericButton {

  override def toString: String =
           s"ColourButton($up, $down, $hover, $background)"
```

[37]More precisely, the bounding box of the glyph that represents the button on the screen.
[38]with a parameter, sometimes ignored, that captures the current state of the keyboard modifiers, the exact button pressed, etc.

Because we cannot rely on the fg (bg) brush of the appearance not being shared anywhere else in the GUI tree, we want to avoid changing that brush. So we construct a *copy* (glyph) of the appearance glyph, with fg (bg) set to currentBrush – a copy of the appearance's relevant brush. We intend to use glyph when drawing the button; its foreground (or background) will be painted using the copied brush, and the appropriate features of that brush will be copied (from one of up, down, hover) according to the current state of the button.

```
val up: Brush =
    if (background) appearance.bg else appearance.fg

val currentBrush: Brush = up.copy()

val glyph: Glyph          =
    if (background)
       appearance(bg=currentBrush)
    else
       appearance(fg=currentBrush)

def setCurrentBrush(b: Brush): Unit = {
   currentFG.color(b.color).width(b.strokeWidth)
 }
```

The draw method shows the current state of the button by painting it with the appropriate brush using the appropriate opacity (alpha). It captures the current geometric transform that will be used to render its glyph.[39]

```
def draw(surface: Surface): Unit = {
    val (brush, alpha) =
        if (disabled) (up, alphaDisabled) else
        if (inactive) (up, alphaUp) else
        (pressed, hovered) match {
          case (true, true)  ⇒ (down,  alphaDown)
          case (false, true) ⇒ (hover, alphaHover)
          case (_, _)        ⇒ (up,    alphaUp)
        }
    surface.withAlpha(diagonal, alpha) {
      setCurrentBrush(brush)
      glyph.draw(surface)
      surface.declareCurrentTransform(this)
    }
}
```

The following definitions can be overridden if necessary, but have proven satisfactory in practice.

[39]The latter is used to speed up the tracking of mouse movements. If the button is inactive or disabled, it won't be used, but capturing it does no harm.

```
def alphaDisabled: Int = 0x70;   def alphaUp:     Int = 0xFF
def alphaDown:     Int = 0xFF;   def alphaHover:  Int = 0xF0
```

The actual glyph that will be shown must be properly installed in the GUI tree by making the button glyph its parent.

```
locally { glyph.parent = this }
```

The rest of the button glyph description is completely standard: it implements the remaining glyph features by forwarding to its "embedded" glyph.

```
override def diagonal: Vec = glyph.diagonal

override def glyphContaining(p: Vec): Option[Hit] =
                glyph.glyphContaining(p)

override def contains(p: Vec): Boolean =
                glyph.contains(p)

val fg: Brush = glyph.fg
val bg: Brush = glyph.bg

def copy(fg: Brush=this.fg, bg: Brush=this.bg): Glyph =
    new ColourButton(appearance(fg.copy(), bg.copy()), down, hover, react)

}
```

As usual we define a companion object to deliver methods that support the convenient construction of useful ColourButtons. The first one we show here provides a text-labelled button using the various defaults provided by Brushes. The defaults can be overridden at construction time.[40]

```
object ColourButton {
 import Brushes.{buttonForeground⇒up, ButtonDown   ⇒down,
                 buttonBackground⇒bg, buttonHover  ⇒hover }
 def apply(text:  String,
           up: Brush=up, down: Brush=down, hover: Brush=hover,
           bg: Brush=bg, background: Boolean = true)
          (react: Reaction): ReactiveGlyph =
 { val glyph: Glyph = Brushes.buttonText(text)
   new ColourButton(glyph, down, hover, background, react)
 }

 def apply(glyph:Glyph,down:Brush,hover:Brush,background:Boolean)
          (react: Reaction): ReactiveGlyph =
     new ColourButton(glyph, down, hover, background, react)
}
```

[40]The Brushes defaults are all defined as variables, so can be changed as a program starts. This provides a very rudimentary alternative to the styling features shown later.

F Examples

The very best way of getting to grips with **Glyph** is to study the source of one or two of the examples included in the tests package: particularly the glyph sampler code from tests.demonstrationBook, and the smaller tests.CalculatorExample. Additional examples worth looking at are ResizeableWindowTest and ResizeAndSplitTest. They demonstrate sophisticated use of glyphXML and the Resizeable container to build resizeable GUIs,

The following few examples are considerably smaller, and not particularly useful save as a getting-started guide. The use of implicit style parameters in examples 3 and 4 means that the reader should be somewhat familiar with the notion of implicits in Scala 2.

F.1 A Passive GUI

This is an entirely passive application. Its "interface" is an (unstyled) text label. The abstract class Application provides the link between the interface and the outside world, needing only a definition of GUI to set up and populate its single main window.

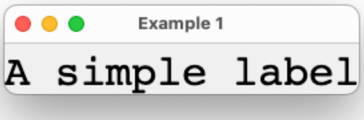
```
package org.sufrin.glyph
package tests

import unstyled.static.Label

object Example1 extends Application  {
  val font = FontFamily("Courier").makeFont(size=32)
  val GUI: Glyph = Label("A␣simple␣label", font)
  override def title: String = "Example␣1"
}
```

On my computer, the text of the label seems a bit close to the inner edge of the window: so our first modification to the program will be to enlarge it all round by an uncoloured rim of width 20**ux**.[41] border around the label. The colours and font of unstyled labels are given default values in the definition of Label.

```
val GUI: Glyph = Label("A␣simple␣label", font).enlarged(20)
```

[41]Distance measurements are expressed in (possibly fractional) "logical units" (ux) – these sometimes correspond to the physical pixels on a screen, but on some high-resolution screens a **ux** may correspond to more than one pixel. **Glyph** manages the correspondence.

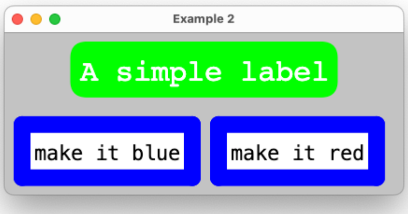
The effect is discernible but not drastic.



F.2 A GUI with explicitly-styled components

This application's interface is defined as GUI in the trait Example2Interface that is mixed-in with Application to form the main program.

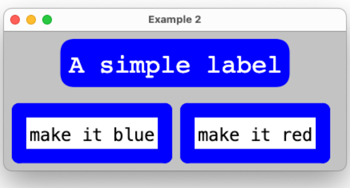
```
package org.sufrin.glyph
package tests

import unstyled.static._
import NaturalSize.{Col, Row}
import unstyled.reactive.TextButton
import Brushes._
import GlyphTypes.Font

trait Example2Interface { ... }

object Example2 extends Application  with Example2Interface {
  override def title: String = "Example␣2"
}
```

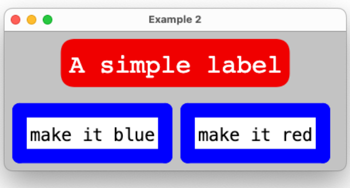


The interface is a centered column on a light grey background on which there is a label separated by a transparent spacer from a row of two captioned buttons: each framed using a rounded blue brush. Interglyph spacing on the row is 10**ux**. The label's background is initially painted with a rounded green brush; and the buttons change its colour.

```
trait Example2Interface {
  val buttonFrame: Brush =
        blue(cap=ROUND, width=18)
  val labelBackground: Brush =
        green().rounded(18)
  val font: Font =
        FontFamily("Courier").makeFont(size=32)
  val spacer =
      Rect(0, 20, fg=nothing)

  val GUI: Glyph = Col(align=Center, bg=lightGrey) (
    Label("A␣simple␣label", font,
          fg=white,
          bg=labelBackground) enlarged(20),
    spacer,
    Row(skip=10)(
        TextButton("make␣it␣blue")
          { _ ⇒ labelBackground.color(blue.color) }
          .framed(buttonFrame),

        TextButton("make␣it␣red")
          { _ ⇒ labelBackground.color(red.color) }
          .framed(buttonFrame)
    )
  ).enlarged(20)
}
```

A TextButton's default response to the mouse cursor entering it is to turn its caption green; when the cursor is pressed in this state the caption turns red, and if the cursor is released when the caption is red, then the button's reaction method is invoked. Here, each button's reaction changes the colour labelBackground that was used to paint the label's background.





F.3 A GUI with implicitly-styled components

This application was derived from example 2, and has an almost identical source code structure, except that the interface trait

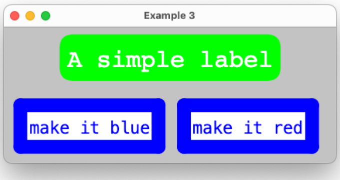
1. leaves the detailed appearance ("styling") of the components it uses to be specified implicitly by a StyleSheet named style which is to be determined later; and

2. imports its Label and TextButton components from the styled package, in which all components are defined with styling set.

```
package org.sufrin.glyph
package tests
import unstyled.static.Rect
import NaturalSize.{Col, Row}
import Brushes._
import styled.TextButton
import styled.Label

trait Example3Interface {
  implicit val style: StyleSheet

  val spacer = Rect(0, 20, fg=nothing)

  def GUI: Glyph = Col(align=Center, bg=lightGrey) (
    Label("A simple label") enlarged(20),
    spacer,
    Row(skip=10)(
      TextButton("make it blue")
      { _ => style.labelBackgroundBrush.color(blue.color) },

      TextButton("make it red")
      { _ => style.labelBackgroundBrush.color(red.color) }
    )
  ).enlarged(20)

}
```

Here the details of the appearance of the interface have been abstracted into the single *implicit* style value – whose definition has been delegated to a main program.[42]

[42]Did you notice that the GUI: Glyph is defined as as a parameterless method? Defining it as a val would lead to the various components dependent on style starting to be evaluated during the construction of an Example3Interface object before style is materialized – a subtle

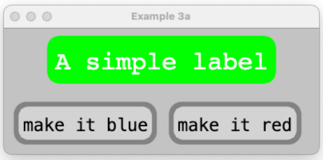


Our first main program defines style so that the interface looks exactly the same as that of Example2. This differs from the default style in only a few respects.

```
object Example3 extends Application with Example3Interface  {
    override def title: String = "Example␣3"
    implicit val style: StyleSheet = StyleSheet(
      labelBackgroundBrush  = green()·rounded(18),
      labelForegroundBrush  = white,
      labelFontFamily       = FontFamily("Courier"),
      labelFontSize         = 32,
      buttonDecoration = styles·decoration·Edged(blue(cap=ROUND, width=18))
    )
}
```

Of course all styles deliver buttons (and other glyphs) with the same functionality, so *the visual style of an interface can straightforwardly be decided upon separately from its functionality.*

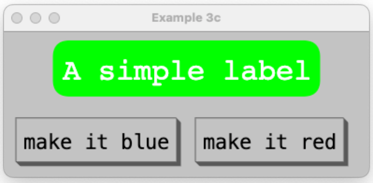
The following appearances differ only insofar as their interfaces were built with styles specifying different button appearances.




```
    buttonBackgroundBrush = grey2,
    buttonForegroundBrush = black,
    buttonDecoration =
      styles·decoration·Framed( fg=darkGrey(cap=ROUND, width=6)
                              , bg=grey2
                              , radiusFactor = 0·3f)
```

error whose occurence depends on the order of construction of the fields of trait instances. An alternative would be to define it as a lazy val. The calculator example of F.4 gives a straightforward simple alternative to extending Application with an interface trait.

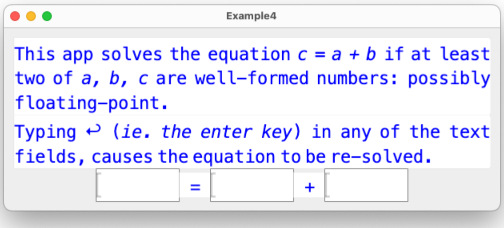

```
buttonForegroundBrush = red,
buttonDecoration =
  styles·decoration·Blurred(fg=yellow, bg=nothing, 16, 5)
```




```
buttonForegroundBrush = black,
buttonDecoration =
  styles·decoration·Shaded(fg=darkGrey, bg=nothing, 16, 5)
```

F.4 A Primitive Calculator

Here we construct a (very) primitive calculator that also uses styled components. The interface is now defined as a class with a style parameter. The text on its window is specified in glyphXML, a notation reminiscent of (but different from and incompatible with) HTML.[43]




```
package org.sufrin.glyph
package tests
import NaturalSize.{Col, Row}
import styled.overlaydialogues.Dialogue.OK
import styled._

object Example4 extends Application {
  val GUI: Glyph = new Example4Interface(StyleSheet()).GUI
  override def title: String = "Example4"
}

class Example4Interface(sheet: StyleSheet) {
  implicit val style: StyleSheet = sheet
  import glyphXML.Language._

  val help: Glyph =
    <div width="40em" align="justify">
      <p>
      This app solves the equation <i>c = a + b</i>
      if at least two of <i>a, b, c</i>
      are well-formed numbers: possibly floating-point.
      </p>
      <p>
        Typing <tt></tt> (<i>ie. the enter key</i> in any of
        the text fields, causes the equation to be re-solved
        if possible.
      </p>
    </div>
```

[43]This time the interface is specified as a class that takes a style sheet as a parameter, thereby avoiding the need to defer construction of the GUI until the style materializes.

The interface embodies three text fields, all of which have the same appearance and behaviour. When the Enter key is pressed within them they invoke calculemus() if the text of the field looks like a number; otherwise they pop up a dialogue objecting to it.

```
  val a, b, c = textField()
  val fields = List(a, b, c)

  def textField(): TextField = TextField(
     size = 8,
     onEnter = {
       case s: String if s.toDoubleOption.isDefined ⇒ calculemus()
       case s: String ⇒
         OK( objectTo(s) ).InFront(help).start()
     }
   )

   val GUI: Glyph = Col(align=Center)(
    help enlarged 25,
    Row.centered(c.framed(), TextLabel("␣=␣"),
                 a.framed(), TextLabel("␣+␣"), b.framed())
  ) enlarged(25)

  def calculenus(): Unit = ···
}
```

The calculemus() method is the core of the application. It tries to convert each of the text fields into numbers, then calculates the third if at least two are defined.

```
def calculemus(): Unit =
 (c.text.toDoubleOption, a.text.toDoubleOption, b.text.toDoubleOption)
  match {
     case (None, Some(av), Some(bv)) ⇒ c.text = format(av+bv)
     case (Some(cv), Some(av), None) ⇒ b.text = format(cv−av)
     case (Some(cv), None, Some(bv)) ⇒ a.text = format(cv−bv)
     case (Some(cv), Some(av), Some(bv)) ⇒
          if (cv == av+bv) {} else c.format = text(av+bv)
     case _ ⇒
   }

def  format(d: Double): String = f"$d%.5g"
```

G Images of GUIs

Figures 5 to 16 show images of a few pages of the tests.demonstrationBook.Pages example. They illustrate some of the available tools in use, and their source code shows how the tools can be used. Figures 17 and 18 show a dynamically configurable GUI.

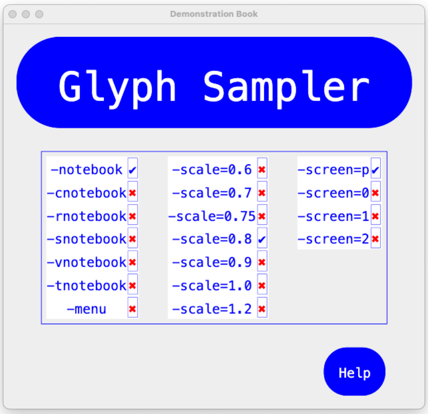


Figure 5: the Splash page with checkboxes to select stylistic features

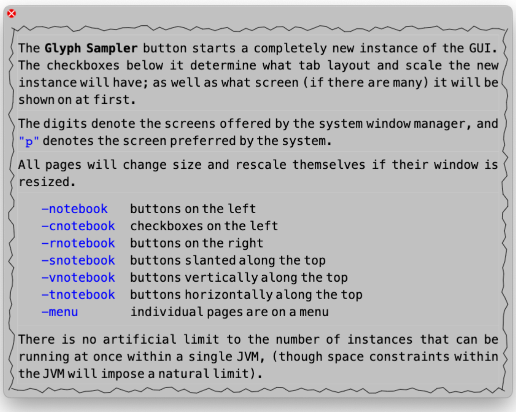


Figure 6: Help for the Splash page

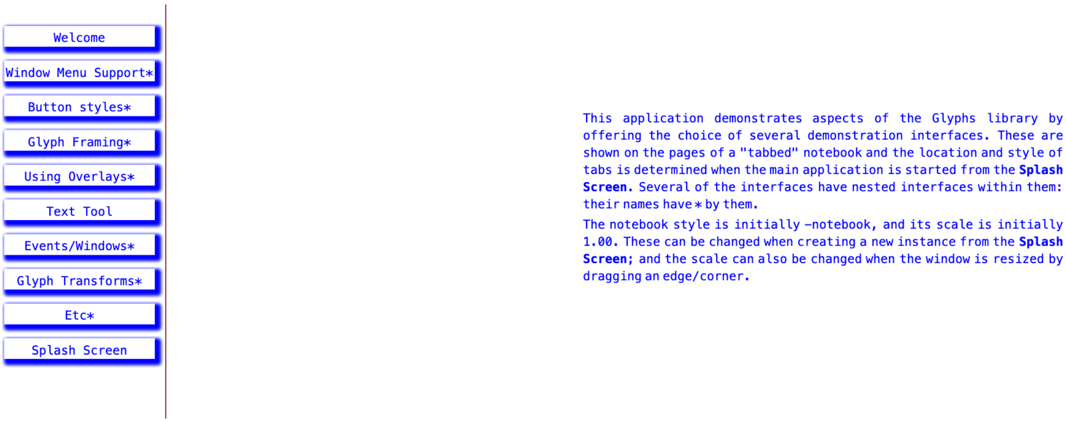


Figure 7: Notebook style welcome page, with sidebar page-selector

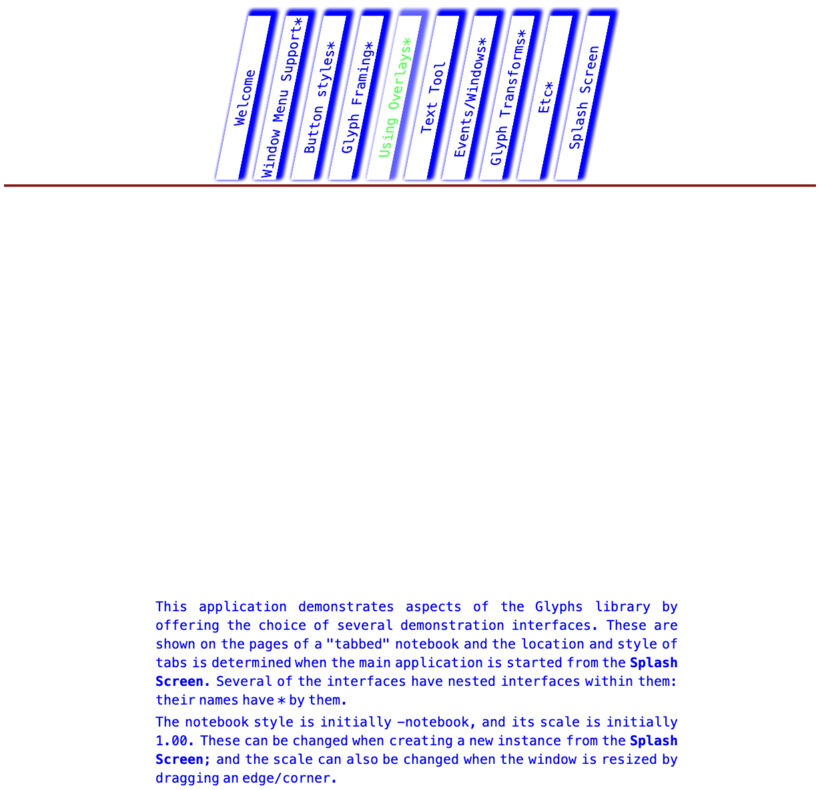


Figure 8: a different notebook style page-selector

Figure 9: the Menu style page-selector

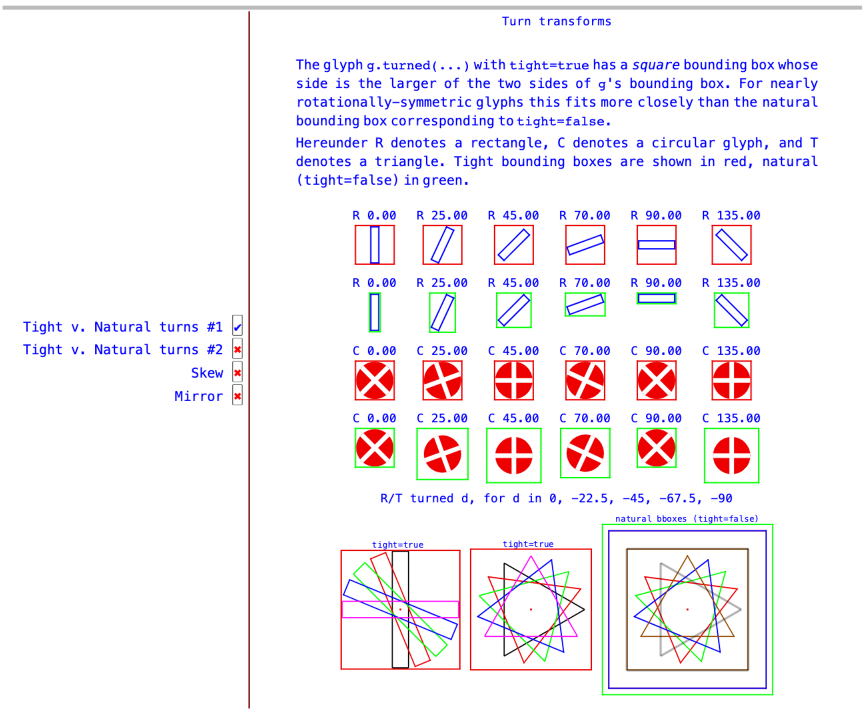


Figure 10: Turn transformations

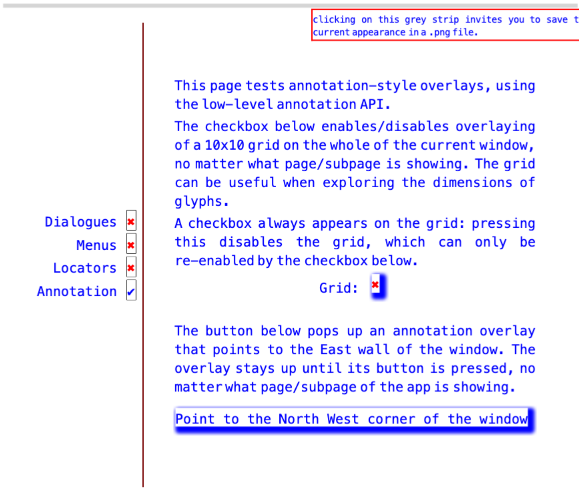


Figure 11: Annotations page

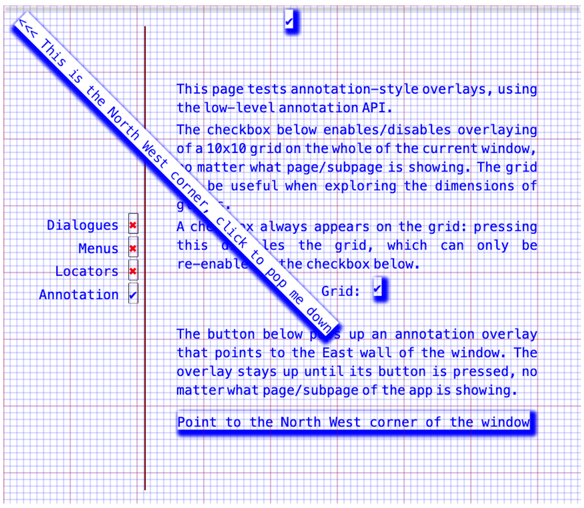


Figure 12: Annotations page with grid and corner enabled

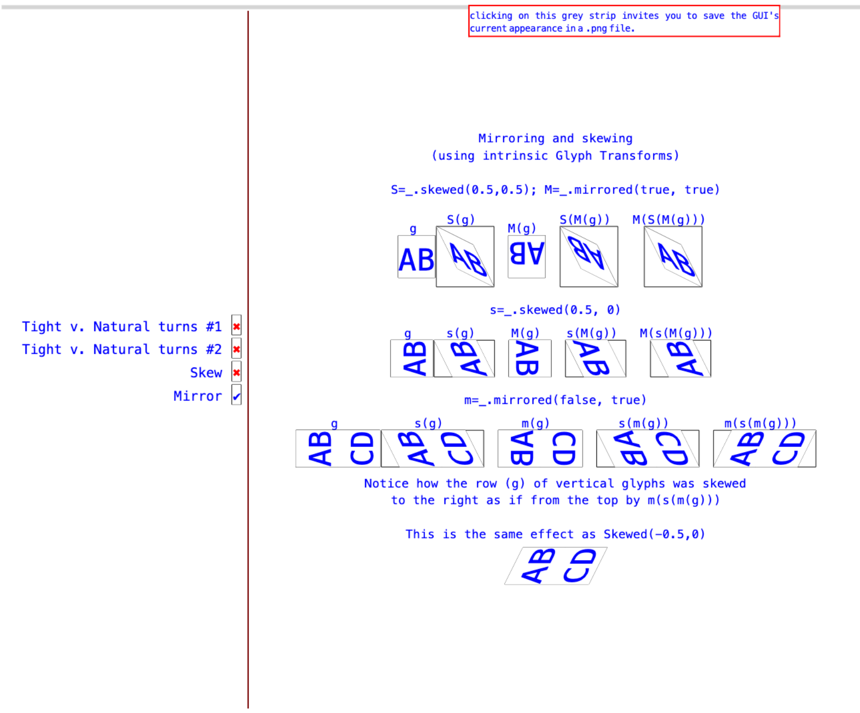


Figure 13: Mirror transformations

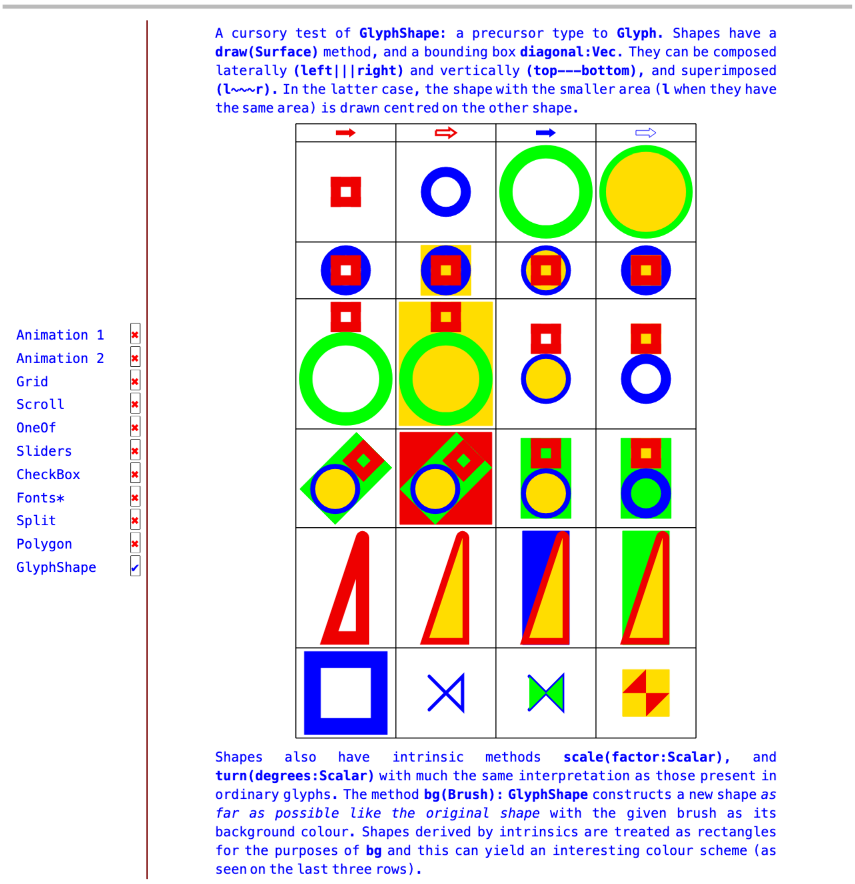


Figure 14: Exploration of the GlyphShape API

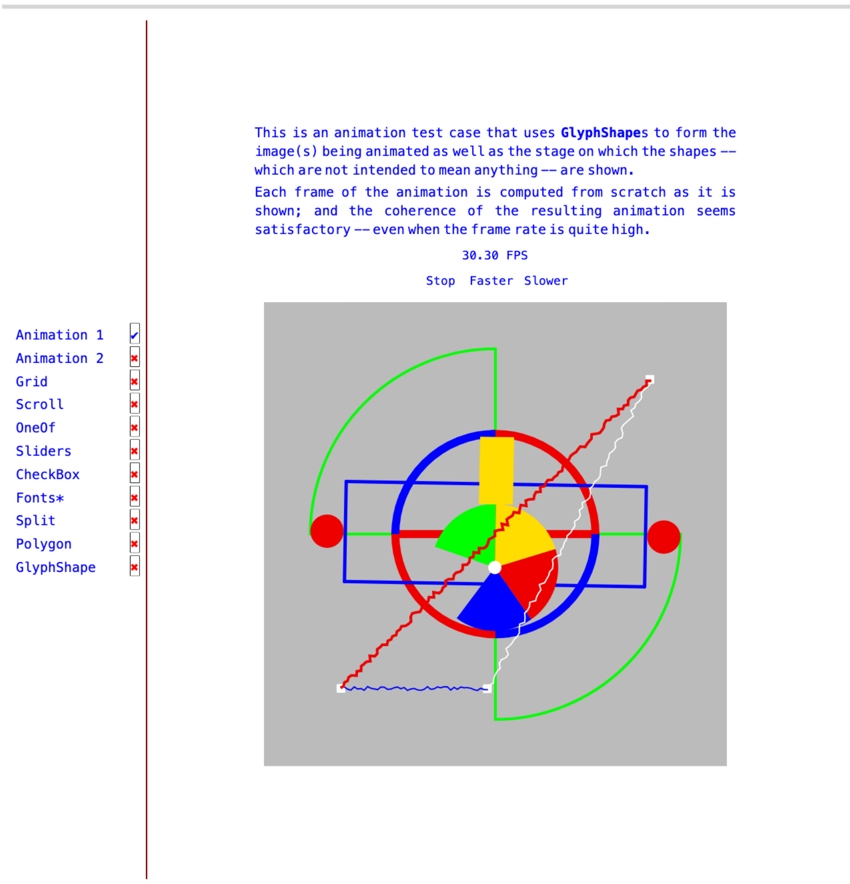


Figure 15: An animation in progress

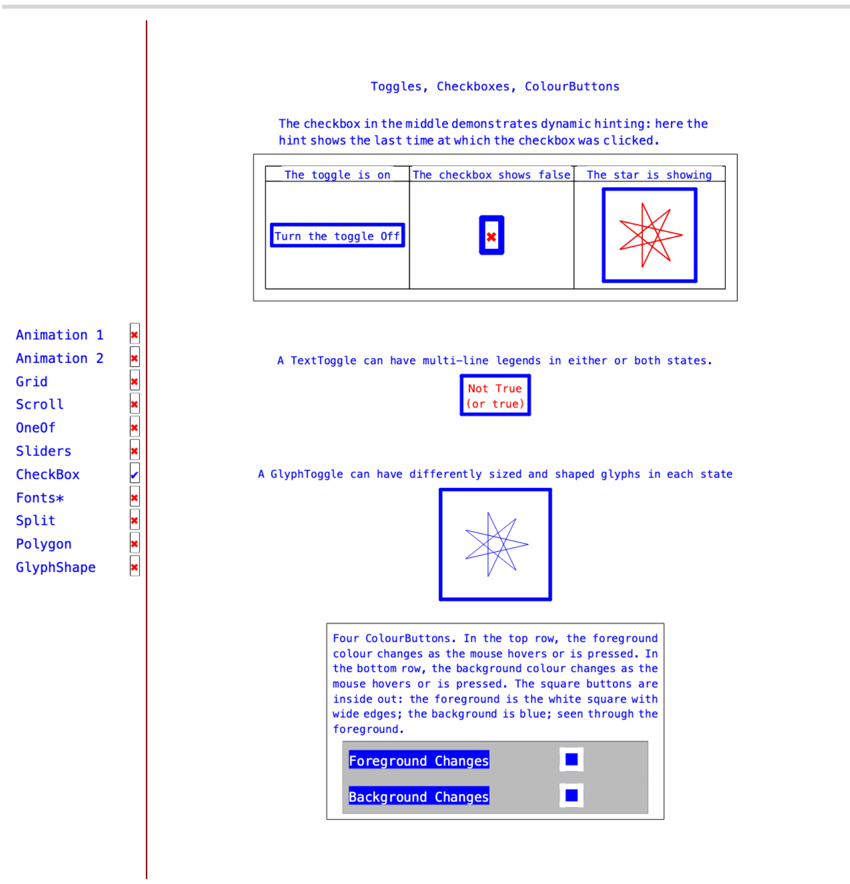


Figure 16: Varieties of checkbox

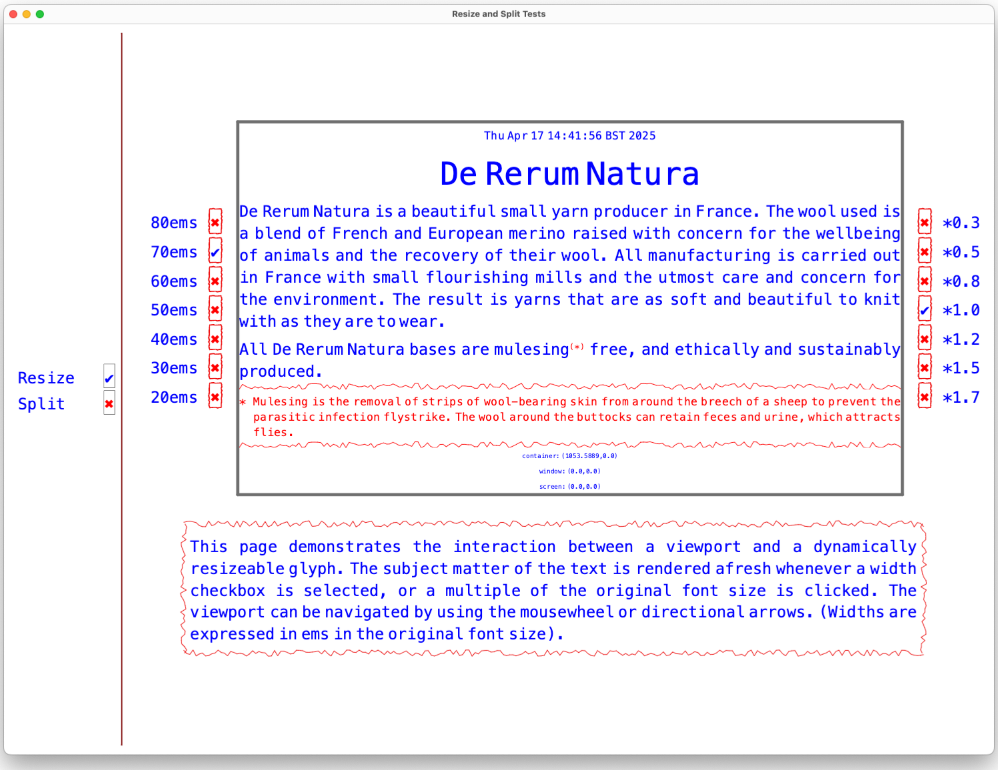


Figure 17: Viewport onto a dynamically configurable glyph

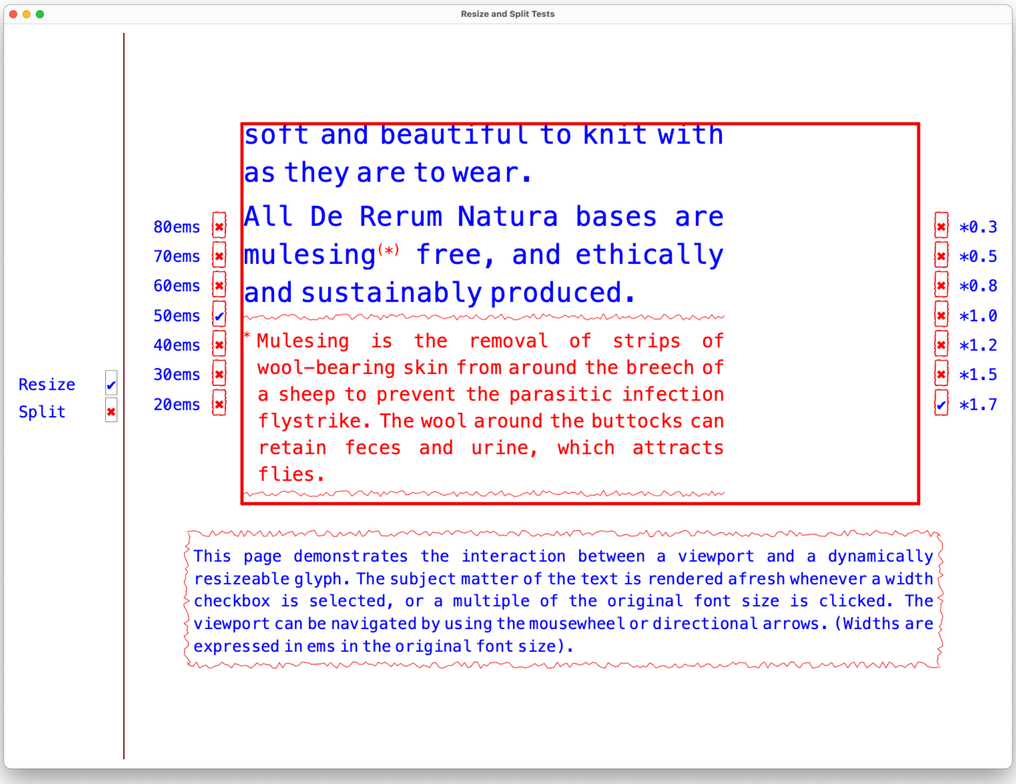


Figure 18: Viewport onto a dynamically configurable glyph (different configuration)

G.1 Source that gave rise to Figures 17 and 18

```
val ResizeTest: Glyph = {
  // A textual glyph whose dimensions and style are both specified by context.
  // Its width is 0.95 * the width of the container it will occupy.
  def theText(implicit context: StyleSheet): Glyph =
   <div width="0.5*container.width" align="justify">
    // time the glyph was last constructed
    <p align="center" fontScale="0.7">{s"${new␣java.util.Date()}"}</p>
    // title
    <p align="center" fontScale="2">De Rerum Natura</p>
    <p>
      De Rerum Natura is a beau_tiful small yarn producer in France. The
      wool used is a blend of French and Euro_pean merino raised with
      con_cern for the well_being of an_imals and the rec_ov_ery of their
      wool.  All man_ufact_uring is carr_ied out in France with small
      flour_ishing mills and the ut_most care and con_cern for the
      env_iron_ment.  The res_ult is yarns that are as soft and beau_tiful
      to knit with as they are to wear.
    </p>
    <p>
      All De Rerum Natura bases are mule_sing<row><span
      textForeground="red" fontScale="0.5">(*)</span></row> free, and
      eth_ical_ly and sus_tain_ably produced.
    </p>
    <fill width="1*container.width" foreground="red/1~5~5"  height="0.7ex"/>
    <p fontScale="0.7" labelForeground="red" textForeground="red" hang="*␣">
      Mulesing is the removal of strips of wool-bearing skin from around
      the breech of a sheep to prevent the parasitic infection flystrike.
      The wool around the buttocks can retain feces and urine, which
      attracts flies.
    </p>
    <fill width="1*container.width" foreground="red/1~5~5"  height="0.7ex"/>
    <div align="center" fontSkip="0ex">
    <p fontScale="0.4" >{s"container:␣${context.containerDiagonal}"}</p>
    <p fontScale="0.4" >{s"window:␣${context.windowDiagonal}"}</p>
    <p fontScale="0.4" >{s"screen:␣${context.screenDiagonal}"}</p>
    </div>
  </div>
```

```
  // resizeable glyph whose content is a function of a stylesheet
  lazy val resizeable =
      styled.Resizeable { case context: StyleSheet => theText(context) }

  val viewPort =
      unstyled.dynamic.ViewPort(resizeable,
                                fg=redFrame,
                                initialPortDiagonal=resizeable.diagonal)

  val widths = List(80, 70, 60, 50, 40, 30, 20)
  val radioWidths =
    styled.RadioCheckBoxes(widths.map{ems => s"${ems}ems"}, prefer = "70ems"){
      case None =>
      case Some(widthIndex)  => // index of the selected checkbox
        viewPort.reset()
        // Choose a column width in ems, then rebuild resizeable
        resizeable.atSize(Vec(widths(widthIndex)*style.emWidth, 0f))
    }

  val scales = List(0.3f, 0.5f, 0.8f, 1.0f, 1.2f, 1.5f, 1.7f)
  val radioScales =
    styled.RadioCheckBoxes(scales.map{ scale => s"*$scale"}, prefer="*1.0") {
      case None =>
      case Some(scaleIndex) => // index of the selected checkbox
        viewPort.reset()
        // Choose a scaled text font, then rebuild resizeable
        resizeable.currentStyle =
           resizeable.currentStyle.copy(
              textFontSize=protoStyle.textFontSize*scales(scaleIndex))
        resizeable.atSize(resizeable.currentStyle.containerDiagonal)
    }

  Col(align=Center)(
      FixedSize.Row(width=80ems, align=Mid)(
        Col(align=Center)(radioWidths.glyphButtons()),
        style.hFill(), viewPort, style.hFill(),
        Col(align=Center)(radioScales.glyphButtons(align=Left)),
      ),
      style.vFill(3),
      <p width="80ems" align="justify">
        This page demonstrates the interaction between a viewport and a
        dynamically resizeable glyph.  The subject matter of the text is
        rendered afresh whenever a width checkbox is selected, or a multiple
        of the original font size is clicked.  The viewport can be navigated
        by using the mousewheel or directional arrows. (Widths are expressed
        in ems in the original font size).
      </p> enlarged 20 framed(red(width=1).sliced(5,5)) enlarged 10
  )
}
```

H What next?

If you have found these example interfaces interesting and would like to study one or two larger working examples that use many features of **Glyph** then tests.demonstrationBook.Pages is a good place to start.

The package styled defines a plethora of styled glyphs, both reactive and passive; and the styled.overlaydialogues and styleded.windowdialogues packages show how to use the **Glyph**'s basic features to build popup dialogues and menus.

The package glyphXML was a latecomer to this party. It integrates the scala.xml notation embedded (in Scala 2) with styles, and makes it fairly straightforward to mix styled texts with glyphs in a way that may be helpful. It is something of a work in progress, but has been used throughout much of the code of tests.demonstrationBook.Pages.

The goal of using GUIs is to control useful applications, and as well as the usual futures/promises machinery provided in Java and Scala, we have designed new ways of using **Glyph** interfaces to control applications that need not be running in the same thread or process or even the same virtual machine as the interface itself. These depend on channel-based communication, as implemented in our microCSO DSL and its (somewhat more extensive) predecessor ThreadCSO. The latter provides cross-network communication straightforwardly through its .net package.

In due course we will provide examples of using **Glyph** with these DSLs to build larger-scale application programs.

I Acknowledgements

This work would not have been possible were it not for the open source **Skia**, **Skija**, and **JWM** projects.

I am grateful to Sasha Walker*, for her patience in waiting for me to develop the initial working prototype of this library, and for her tolerance when listening to explanations of my implementation of the focus protocol.

Dominic Catizone* made a remark that helped me solve a problem with designing the geometry of (non-quadrant) rotations of bounding boxes.

*Of Magdalen College, and the Department of Computer Science, Oxford University