The **Glyph** User-Interface Library

Bernard Sufrin*

Draft of June 18, 2024

Abstract

In this note 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]

The belief that has guided this work is that a rich set of interactive interface components can be composed using a small collection of combining forms, together with a suitable collection of elementary visual components – some reactive. So 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.

---

*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: it's the cost of straying outside the styling envelope that we want 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: a comprehensive kit for building app(lication)s for a large variety of devices and operating systems.

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

<div align="center">Row(Col(a, b), Col(a, b))</div>

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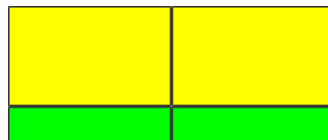
The height of a row is the maximum height of its components; its width is the sum of its components' widths; and a row is normally drawn with its components aligned along its top edge.

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 C.

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

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 Row and Col compositions are (almost) explainable by their names. Each has variants that explain the exact way in which components are aligned. Here are the Row·atTop(···) Row·centered(···) Row·atBottom(···) compositions of a pair of rectangles:[10]

Here are the Col·atLeft(···) Col·centered(···) Col·atRight(···) compositions of a different pair:[11]
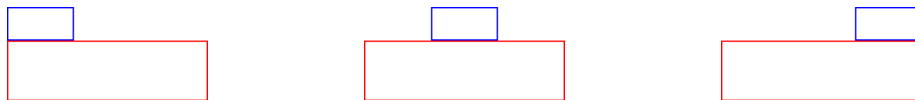
The row and column compositions described above yield glyphs that are "naturally" sized. Thus, for example, a row's width is the sum of its components' widths, and its height is the maximum of its components' heights. There are also fixed-size row (and column) compositions, whose row width and column height can be declared in advance, and that allow "expandable" spaces as components. Here we see a couple of examples:[12]

These were constructed by:

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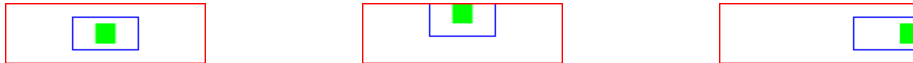
with

---

[10]Row, and Row.atTop mean the same.
[11]Col, and Col.atLeft mean the same.
[12]tab is an expandable space.

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

Finally, here are the Concentric·centered($\cdots$), Concentric·atTop($\cdots$) and dConcentric·atRight($\cdots$) compositions of a point and a pair of rectangles:[13]

OneOf: Temporal Alternations of Glyphs

The OneOf composition is used primarily in the implementation of dynamically "paged" interfaces. 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[14]

```
def bananas = Label("Bananas", fg=white, bg=blue)
val oneof: OneOf =
    OneOf(bg = Wide·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:

---
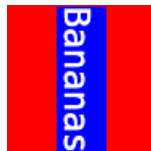
[13]Concentric, and Concentric.centered mean the same; and there are (of course) atLeft and atBottom variants of Concentric.

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

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.[15]

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

Grid and 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).*method*(...)

The main *method*s of such a Grid are:

```
Grid(width:  Int=0, height: Int=0)(glyphs: Glyph*): Composite
Table(width: Int=0, height: Int=0)(glyphs: Glyph*): Composite
grid(width:  Int=0, height: Int=0)(glyphs: Seq[Glyph]): Composite
table(width: Int=0, height: Int=0)(glyphs: Seq[Glyph]): Composite
```

The grid/Grid method organises glyphs as a sequence of rows of *identically-sized* cells, each large enough for any of the glyphs. When $width > 0, height \not> 0$ then *width* is the number of glyphs per row; when $width \not> 0, height > 0$ then *height* is the number of glyphs per column.[16]

The table/Table method with $width > 0, height \not> 0$ is similar, except that all cells in each column have the same width, while their height is the greatest of all those in the same row. When $width \not> 0, height > 0$ then *height* is the number of glyphs per column, and their width is the greatest of all those in the same column.

Figure 20 illustrates several ways of using the Grid methods.

---

[15]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 C for details.

[16]When neither condition is satisfied, then *width* is chosen as the ceiling of **sqrt**(*glyphs.length*)

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.[17]

```
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)
```

---

[17]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.

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.

```
Row.centered(Point(Wide.blue(cap=ROUND)),
             Skip(Wide.blue.strokeWidth),
             Rect(150f, 100f, Wide.blue(cap=ROUND))))
```


Glyph

eg1d

```
Row.centered(Point(Wide.blue),
             Skip(Wide.blue.strokeWidth),
             Point(Wide.green), Rect(150f, 100f, Wide.blue)
```


Glyph

eg1e

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.[18] 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.[19]

For example:

```
Row·centered(
    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

---

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

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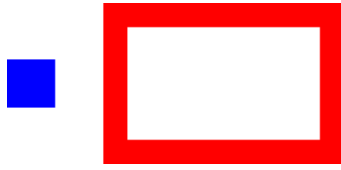
3 Glyph Transformers

Glyph transformers are used to derive glyphs from simpler glyphs. For example:

```
Framed(Wide.red)(eg1e)
```

```
Framed(Wide.red)(
  Framed(Wide.blue)(
    Concentric(
      FilledOval(120f, 80f, green), Point(Wide.red))))
```

It's important to understand that, as a matter of policy, transforming a glyph, g is an "algebraic" operation that results in a tree in which g is embedded: it no more makes a new copy of it than (for example) the successor *succ n* of a number n makes a new copy of n.[20]

Most of these "extrinsic" transformers are also delivered as "intrinsic" methods of glyphs. At the time of writing the intrinsic transformers are:

```scala
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 =
```
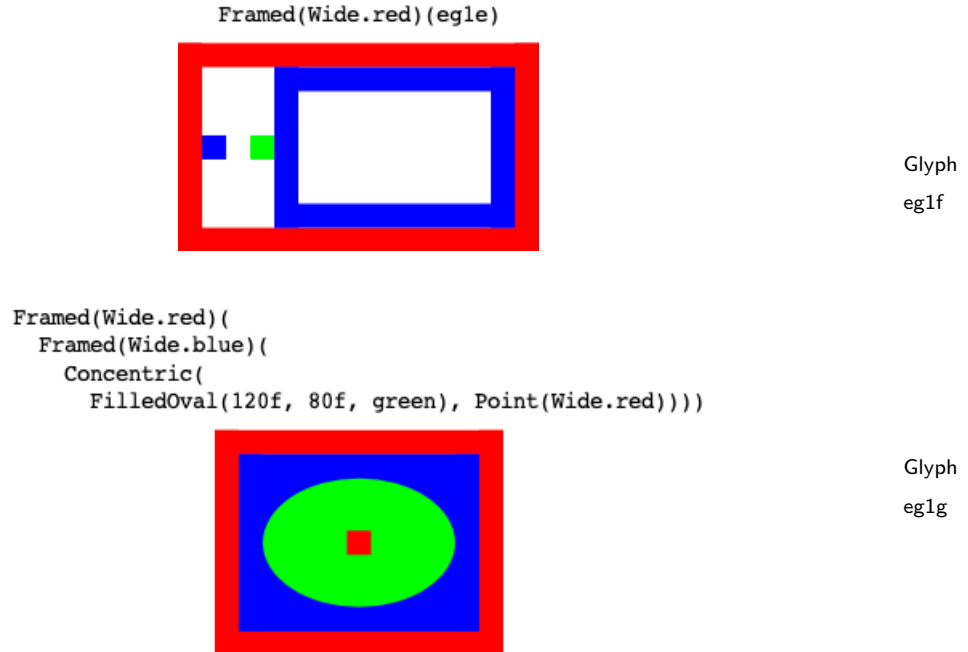
---

[20]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 mirrored(leftRight: Boolean, topBottom: Boolean, ···): Glyph =
def framed(fg: Brush = Glyphs·Framed·defaultFG,
           bg: Brush = Glyphs·Framed·defaultBG): Glyph
def edged(fg: Brush = Glyphs·Framed·defaultFG,
          bg: Brush = Glyphs·Framed·defaultBG): Glyph
def shaded(···
           enlarge: Scalar = 0·25f,
           delta: Scalar = 8f,
           down: Boolean=false): Glyph =
```

In the above signatures, the abbreviation $\cdots$ stands for the declaration that 'fg' and 'bg' parameters be inherited from "this" Glyph 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 1 shows a few examples of their use:
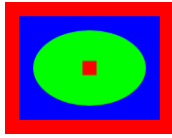
g
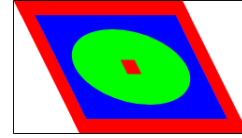
g·skewed(0·5,0)·framed()
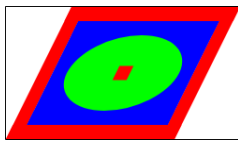
g·skewed(−0·5, 0)·framed()
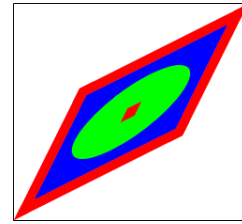
g·skewed(−0·5, −0·5)·framed()
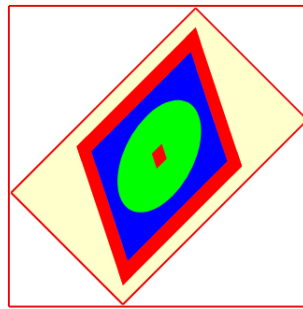

```
g · skewed ( 0 · 5 f ,  0 f ) · enlarged ( 27 ,  bg=yellow ( alpha=0 · 2 f ) )
              · framed ( redFrame )
              · turned ( 45 f )
              · mirrored ( leftRight=true , topBottom = false )
              · framed ( fg=redFrame )
```
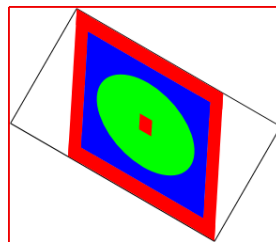

g·skewed(0·5f, 0·0f)·framed()·turned(30f)·framed(fg=redFrame)

Figure 1: 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.[21] 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.[22]

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))
```

---

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

[22] 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

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 both as a filled polygon and 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).[23]

```
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:

---

[23]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 $n \geq 2$.

The blueish : Brush is the same color as blue, but wider: it has "wobbly" edges specified by its pathEffect.

```
val blueish =
      blue ( width        = blue · strokeWidth ∗2 ,
            pathEffect = PathEffect · makeDiscrete (25 · 0f ,  4 · 0f ,  1))
```

Rendered in blueish, the stargons are:

All this works for stars of arbitrary odd arity: here are stars with $n = 17, 11, 5, 3$:
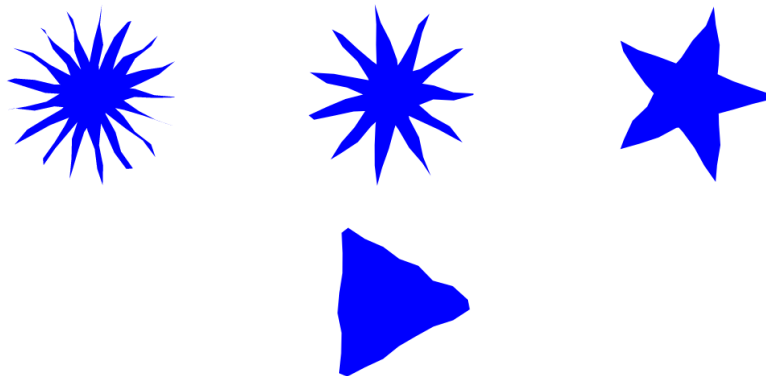
Using brushes with path effects to draw frames can give aesthetically interesting results. For example, here is:

```
Label ("Hello!",  font (32))
· enlarged (40)
· framed ( fg=red ( width=4f ,
                  pathEffect=PathEffect · makeDiscrete (7f ,  12 · 5f ,  15) ,
                  cap=BUTT )))
```

The three parameters of makeDiscrete are, in turn, the length of each "segment" into which the path drawn by the paint will be sliced, the limit of displacement of the endpoints of the segment from the path, and a seed for the generation of random displacements. 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,
                 pathEffect = PathEffect.makeDiscrete(5f, 100f, 15),
                 cap = ROUND)
)((0, 100), (200, 100)).enlarged(4).framed())
```
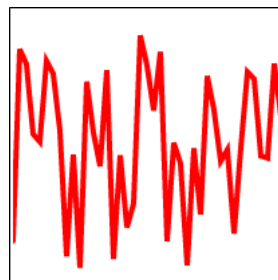
Many other path effects are available from **Skia** *via* **Skija**.

Blurred Paint Effects

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

$$\texttt{brush}\cdot\texttt{blurred(blur: Scalar, dx: Scalar=0f, dy: Scalar=0f): Brush}$$

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 ar framed(): notice that the latter is displaced by $(20, 20)$ within the natural bounding box indicated by the frame.
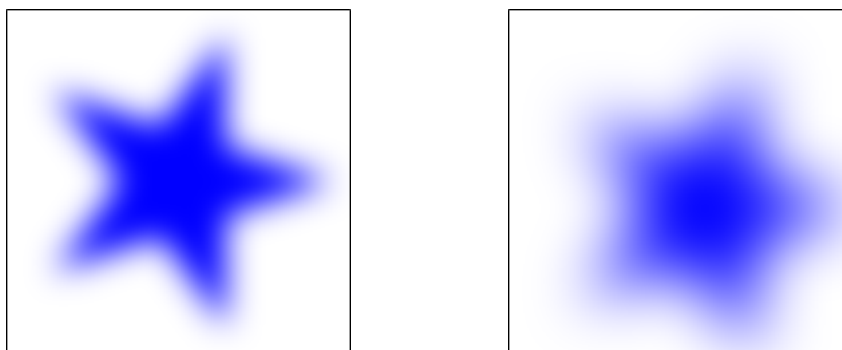
Blurred Frame Effects

Figure 21 shows the use of a blurred brush, as well as uses of the extrinsic glyph transformer

$$\texttt{BlurredFrame(blur: Scalar, spread: Scalar, dx: Scalar=0f, dy: Scalar=0f, } \cdots \texttt{)}$$

The styled blurred button-framing specification

$$\texttt{Decoration}\cdot\texttt{Blurred(blur: Scalar, spread: Scalar, dx: Scalar=0f, dy: Scalar=0f, } \cdots \texttt{)}$$

is illustrated in Figure 17.

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[24] 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
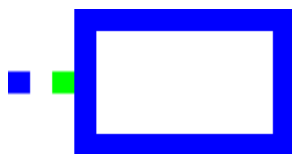
and the expression Row(bg=white)(egc, egc) denotes a glyph drawn as

---
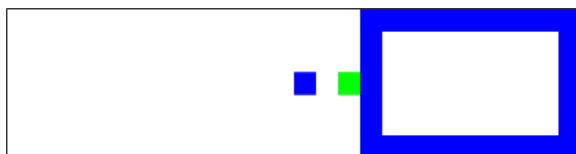
[24]*ie.* without having set location, parent.

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.[25]

When the glyph is copied before both uses all is well: Row(egc(), egc()) is drawn as[26]
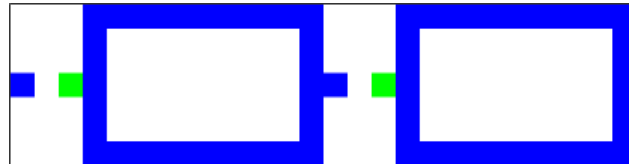
A.2 Brushes are mutable

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 was ever drawn with 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)

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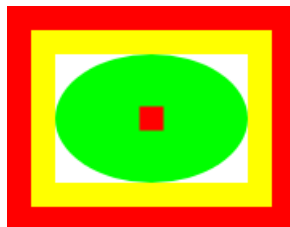
[25]That this can occur *without warning at compile time* is a defect in the design of the library to which there are a multiplicity of potential solutions: we are considering them at the time of writing. But for the moment we advise copying as a matter of course: it's not computationally very expensive.

[26]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

Here we discuss the low-level API for generating simple text glyphs. The Styled package provides more versatile methods for composing glyphs that consist of paragraphed text.

A Text is a glyph-builder, *ie.* a *factory* for text glyphs. The function text, defined below, constructs a glyph-builder from the given string at the specified size in (the italic Courier) typeFace.

```
val typeFace: Typeface =
    FontMgr·getDefault()
              ·matchFamilyStyle("Courier", FontStyle·ITALIC)

def text(s: String, size: Float): Text =
    Text(s, new Font(face, size))
```

The Texts text1, text2 are glyph builders that will eventually yield glyphs. The function em defines an *em*-width space.

```
val text1 = text("Á␣24pt␣Text", 24·0f)
val text2 = text("A␣12pt␣Text", 12·0f)
def em    = Skip(font(12f)·measureTextWidth("m"))
```

Texts provide the following methods that build glyphs:

```
·asGlyph:     Glyph
·atBaseline:  Glyph
```

Both yield glyphs with the same bounding box: but the second yields a glyph that is expected to be drawn only within top-aligned Row's.

Framed(grey)(text1.asGlyph(blue))

Framed(grey)(Row(text1.atBaseline(blue))

Drawing a sequence of atBaseline-generated glyphs in a *top-aligned* row positions them so that their baselines coincide, and this is the intended effect.
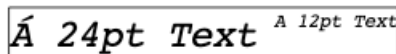
```
Framed(grey)(Row(text1.atBaseline, em, text2.atBaseline))
```

Á 24pt Text A 12pt Text
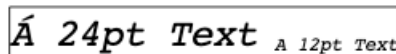
Deprecated uses of **asGlyph** and **atBaseline** glyphs

Using asGlyph-generated glyphs of different sizes in rows is a cheap and nasty way to simulate superscripts and subscripts and is not recommended for normal use.

```
Framed(grey)(Row(text1.asGlyph(), em, text2.asGlyph()))
```

Á 24pt Text A 12pt Text
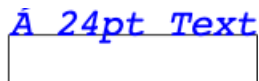
```
Framed(grey)(Row.atBottom(text1.asGlyph, em, text2.asGlyph)
```

Á 24pt Text A 12pt Text
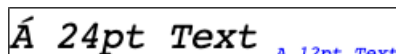
When an atBaseline-generated glyph does not have a Row as parent, it is drawn so that its baseline coincides with the top of its bounding rectangle: the result is never useful.

text1 · atBaseline (fg=blue)·framed(grey)

Á 24pt Text

Moreover, when atBaseline-generated glyphs of different heights are used in a bottom- or centre-aligned row the results are never useful.

```
Row· centered ( text1 · at Baseline ( fg=black ),
              text2 · at Baseline ( fg=blue )) · framed ( grey )}
```

Á 24pt Text A 12pt Text

27

C   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 C.4.

C.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.

C.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.

C.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.[27]

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.

C.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).

---

[27]**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.

C.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" that , 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.[28]

---

[28]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.

D 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.[29] 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.[30]

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)"
```

---

[29]More precisely, the bounding box of the glyph that represents the button on the screen.

[30]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 (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.

```scala
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.[31]

```scala
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.

---

[31]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.

```scala
    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.

```scala
    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.

```scala
    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.

```scala
 object ColourButton {
  val up:     Brush = Brushes.buttonForeground()
  val down:   Brush = Brushes.buttonDown()
  val hover:  Brush = Brushes.buttonHover()
  val bg:     Brush = Brushes.buttonBackground
  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).asGlyph(up, bg)
   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)
 }
```

E Examples

The best way of getting to grips with **Glyph** is to study the source of one or two of the medium-sized examples included with it: particularly DemonstrationNotebook, and CalculatorExample. 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.

E.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·tests
import   org·sufrin·glyph·_
import  Glyphs·Label

object Example1 extends Application  {
   val GUI:  Glyph = Label("A␣simple␣label")
   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 add an uncoloured 20 **ux**[32] 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")·enlarged(20)
```

The effect is discernible but not drastic.

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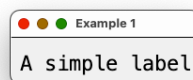
[32]Distance measurements are expressed in (possibly fractional) "logical units" (ux) – these sometimes correspond to the physical pixels on a screen, but on some high-resolution screens a **ux** may correspond to more than one pixel. Glyph manages the correspondence.

E.2 An Explicitly-styled GUI

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·tests
import org·sufrin·glyph·_
import Glyphs·_
import NaturalSize·{Row, Col}
import ReactiveGlyphs·TextButton

trait Example2Interface { ··· }

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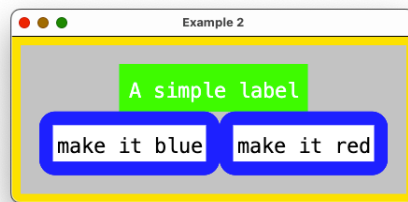

The interface is a centered column, on which there is a label above two side-by-side captioned buttons, each with a rounded blue frame around it. It is presented on a grey background with a yellow rim around it

```
trait Example2Interface {
  val blueish:       Brush = blue(cap=ROUND,  width=8)
  val labelColor: Brush = green()

  val GUI: Glyph = Col(bg=lightGrey)·centered(
    Label("A␣simple␣label", fg=white) enlarged(20, bg=labelColor),
    Row(TextButton("make␣it␣blue") { _ ⇒ labelColor color blue·color }
        ·framed(fg=blueish,  bg=blueish)
      ,
        TextButton("make␣it␣red") { _ ⇒ labelColor color red·color }
        ·framed(fg=blueish,  bg=blueish))
  )·enlarged(40f)·enlarged(20f,  bg=yellow)

}
```

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 labelColor that was used to paint the background around the label.

E.3 A GUI with Implicitly Styled Glyphs

This application was derived from example 2, and has an almost identical source code structure, save that the button and label definitions are imported from the styled package.

```
package org·sufrin·glyph·tests
import org·sufrin·glyph·Glyphs·_
import org·sufrin·glyph·NaturalSize·_
import org·sufrin·glyph·_

trait Example3Interface { ··· }

object Example3 extends Application  with Example3Interface {
  override def title: String = "Example␣3"
}
```

Its interface looks almost the same as the previous example, save that the button framing is a little more elegant, and the simple label is in a larger font, and placed on a coloured background
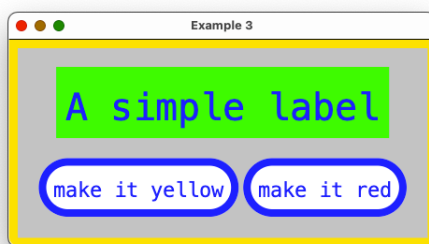
The Example3Interface is different: it uses *implicitly-styled* glyphs and leaves much of the heavy lifting concerning font sizes, and button frames to be defined implicitly by the *enclosing context.* Here that context is provided by importing the LocalStyle object:

```
trait Example3Interface {
  object LocalStyle extends Styles.Basic { ··· }
  import LocalStyle._
  import Spaces._
  import styled.TextButton
  import styled.TextLayout.TextLabel

  val labelColor: Brush = green()

  val GUI: Glyph = Col(bg=lightGrey).centered(
    TextLabel("A␣simple␣label") enlarged(20, bg=labelColor), ex,
    Row(TextButton("make␣it␣yellow") { _ ⇒ labelColor color yellow.color },
        TextButton("make␣it␣red")    { _ ⇒ labelColor color red.color })
  ).enlarged(40f).enlarged(20f, bg=yellow)

}
```

The LocalStyle object is an extension of Styles.Basic that specifies a variety
of styles by default. Here we adapt two of the defaults for use in styling
our labels and our buttons. They are declared as implicit, and the compiler
matches them to the implicit parameters of TextLabel and TextButton: their
names are unimportant: the implicit matching is done on their types.

- localLabels is almost identical to the basic labelStyle, except that its
  font is made (with the same face) at a different size.

- localButtons is almost identical to the basic buttonStyle, except that
  buttons are specified as being framed with round blue frames on a
  white background.

```
object LocalStyle extends Styles.Basic {
  import Styles.Decoration.Framed

  implicit val localLabels: Styles.GlyphStyle =
    labelStyle.copy(font=GlyphTypes.Font(face, 40))
  }

  implicit val localButtons: Styles.ButtonStyle =
    buttonStyle.copy(frame =
      Framed(fg = blue(width = 8, cap=SQUARE), bg = red))
```

Less rounded button frames can be delivered by adopting a higher curvature (lower radius factor), and the frames are square when no radius factor (or a factor of 0) is specified.

```scala
implicit val localButtons: Styles.ButtonStyle =
  buttonStyle.copy(
      frame = Framed(fg = blue(width = 8, cap=SQUARE), bg = red)
  )
```
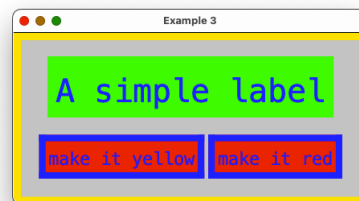

There are other kinds of button-framing, including Shaded, and Blurred

```scala
import Styles.Decoration.{Shaded, Blurred}

implicit val localButtons: Styles.ButtonStyle =
  buttonStyle.copy(frame = Shaded(fg = blue(width = 8), bg = red))

implicit val localButtons: Styles.ButtonStyle =
  buttonStyle.copy(frame = Blurred(blur=10f, spread=10f, fg = yellow))
```
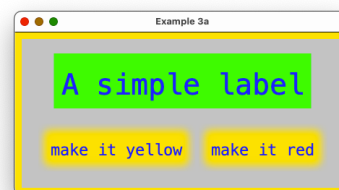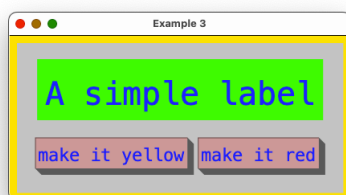

The important thing to understand is that all styles deliver buttons with the same functionality: this means that the visual style of an interface can straightforwardly be decided upon separately from its functionality.

E.4   A Primitive Calculator
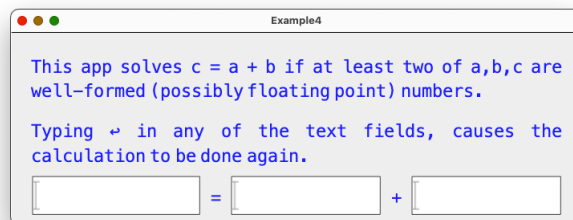
The application also uses styled components; but this time keeps to the basic standard.

```
package example4
import org·sufrin·glyph·NaturalSize·{Col, Row}
import org·sufrin·glyph·_
import styled·TextLayout·_

object Style extends Styles·Basic
import Style·_

trait Interface {
  ...
  val GUI: Glyph = Col·centered(
    help enlarged 25,
    Row·centered(c·framed(), TextLabel("␣=␣"),
                 a·framed(), TextLabel("␣+␣"), b·framed())
  ) enlarged(25)
}
```

Its Interface trait defines its help text, three identically-behaving text (input) fields a, b, c, and a couple of helpful methods.

```
trait Interface {
  val help = TextParagraphs(50, Justify)("···")

  def field(): TextField =
      TextField(size = 8, onEnter = { _ ⇒ calculemus() })

  val a, b, c = field()

  def calculemus() = ···

  val GUI: Glyph = ···
}
```

The most important of these methods is calculemus(). It is the core of the application: when invoked – by ENTER being typed in any of the text fields – it tries to convert each of the text fields into numbers, then calculates the third if at least two are defined.

```scala
def calculemus() =
 (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"
```

A slightly different definition of the behaviour of the text fields, allows a simple error report to be given if what's in a field doesn't look like a number when enter is pressed.

```scala
import windowdialogues·Dialogue·OK

def field(): TextField = TextField(size = 8, onEnter = {
    case s: String if s·toDoubleOption·isDefined ⇒ calculemus()
    case s: String ⇒
      OK(TextLabel(s"$s␣doesn't␣look␣like␣a␣number·"))
        ·InFront(help)
        ·start()
  })
```

F   Pages from DemonstrationNotebook

Figures 2 to 18 show some of the pages of the DemonstrationNotebook example. They are intended to illustrate some of the available tools in use, and their source code shows how the tools can be used.

The (top-level) notebook is structured as a sequence of pages with titles. Pages may themselves have nested notebooks, as exemplified by the "Etc*" page below.

The Layout feature of a notebook supports methods, such as rightButtons, leftButtons, ... whose invocation yields a GUI with buttons juxtaposed to a OneOf that has a glyph for each defined page.

```
trait DemonstrationPages
    · · ·

    val noteBook = new NoteBook
    val Page = noteBook · DefinePage

      Page("Help", "Help␣for␣the␣Demonstration␣Notebook") {···}
      Page("New", "Make␣a␣new␣or␣cloned␣GUI␣") {···}
      Page("Menus∗", "Window␣menus␣and␣dialogues") {···}
      Page("Transforms∗", "") {···}
      Page("Overlays∗", "Features␣implemented␣as␣···") {···}
      Page("Framing∗", "") {···}
      Page("Styles∗", "") {···}
      Page("Events∗", "") {···}
      Page("Fonts∗", "··") {···} (HelpStyle · labelStyle)
      Page("Etc∗", "") {
       val nested = new Notebook {}
       val Page = nested · DefinePage
       Page("Animation", "") {···}
       Page("Grid", "buts␣=␣8␣···␣") {···}
       Page("Blurred", "") {···}
       Page("Scroll", "Scrolling␣and␣Scaling␣with␣ViewPort") {···}
       Page("OneOf", "OneOf␣backgrounds") {···}
       Page("CheckBox", "Toggles␣and␣Checkboxes") {···}
       nested · Layout · rightButtons()
      }

    · · ·
```

Each of the lazy values defined below is bound to one particular notebook layout; and the top-level application logic chooses one of them as the GUI for the application. Figure 12 illustrates the "skewed buttons" presentation style.

```scala
    . . .
    lazy val asRNotebook = noteBook.Layout.rightButtons()
    lazy val asLNotebook = noteBook.Layout.leftButtons()
    lazy val asVNotebook = noteBook.Layout.rotatedButtons(3)
    lazy val asSNotebook = noteBook.Layout.skewedButtons(0.2f, 0f)
    lazy val asTNotebook =noteBook.Layout.topButtons()

} // end DemonstrationPages
```

The application as a whole is an instance of the Glyph Application trait – that provides definitions of GUI, Title, *etc.* Importantly the standard onClose method invokes a method that pops up a dialogue that asks for confirmation when the application window is closed.

```scala
object DemonstrationNotebook extends DemonstrationPages with Application {
  lazy val GUI =
      if (extraArgs contains "-notebook")    asRNotebook else
      if (extraArgs contains "-rnotebook")   asRNotebook else
      if (extraArgs contains "-lnotebook")   asLNotebook else
      if (extraArgs contains "-snotebook")   asSNotebook else
      if (extraArgs contains "-vnotebook")   asVNotebook else
      if (extraArgs contains "-tnotebook")   asTNotebook else
      if (extraArgs contains "-menu") asMenu else asRNotebook

  def title =
      s"""SmallTest␣-scale=$scaleFactor␣${extraArgs.mkString(", ")}"""

  override
  val defaultIconPath: Option[String] = Some ("./flag.png")

  override
  def onClose(window: Window): Unit = confirmCloseOn(GUI)(window)
}
```

Help for the Demonstration Notebook

This application demonstrates aspects of the Glyphs library by offering the choice of several demonstration GUIs. These are usually shown on the pages of a tabbed notebook, with tabs placed to the right. Several of the pages have pages nested within them: their names have ∗ by them.

Command−line arguments affect the notebook style (normally −notebook) and scale (normally 1.00).

    −notebook => tabs to the right (the default)
    −rnotebook => tabs to the right
    −lnotebook => tabs to the left
    −vnotebook => rotated tabs along the top
    −tnotebook => tabs along the top
    −snotebook => rotated and skewed tabs along the top

    −menu => individual popup windows selected from a menu bar

    −scale=d.dd the viewing scale is d.dd (1.00 by default)

The "New" page enables instantiation of a new GUI with a choice of tab style, viewing scale and starting screen.

The application evolved naturally during development because we saw that unit−testing was not going to be effective. It is not, and not intended to be, a comprehensive test; but if it works at all then a very substantial proportion of the toolkit must be functioning adequately.

Log Events

Just exit on close button

Help

New

Menus∗

Transforms∗

Overlays∗

Framing∗

Decoration∗

Events∗

Fonts∗

Etc∗

Figure 2: the Help page

44

Make a new or cloned GUI

The button below starts a completely new instance of the GUI. The checkboxes determine what tab layout and scale the new instance will have; as well as what screen (if there are many) it will be shown on at first.

There is no artificial limit to the number of instances that can be running at once within a single JVM, (though space constraints within the JVM will impose a natural limit).

[New instance]

−notebook✔	−scale=1.2✘	
−lnotebook✘	−scale=1.0✘	−screen=p✔
−snotebook✘	−scale=0.9✘	−screen=0✘
−vnotebook✘	−scale=0.8✘	−screen=1✘
−tnotebook✘	−scale=0.75✘	−screen=2✘
−menu✘	−scale=0.7✔	

Unlike a new instance, a cloned instance always shares some state with the current GUI –– for example the notebook page currently being viewed, and the state of checkboxes on a page. Changes made in a GUI, are reflected in its clones only when they next receive an event (such as entry of the mouse).

Because this is intended to be used only during development, the clone button needs to be enabled (using its adjacent checkbox) before it responds to presses. Tab layout, scale, and screen are specified as for new instances.

[Clone this instance] ✘

Help
New
Menus∗
Transforms∗
Overlays∗
Framing∗
Decoration∗
Events∗
Fonts∗
Etc∗

Figure 3: the New page

45

This page tests popup dialogues from which
multiple choices can be made. Each of the
buttons below pops up a CHOOSE popup with the
specified number of choices on it.

Pressing its close button pops the dialogue
down without making a choice (and yields
null to the popup's continuation).

One Two Three

Dialogues

Menus

Locators

Help

New

Menus*

Transforms*
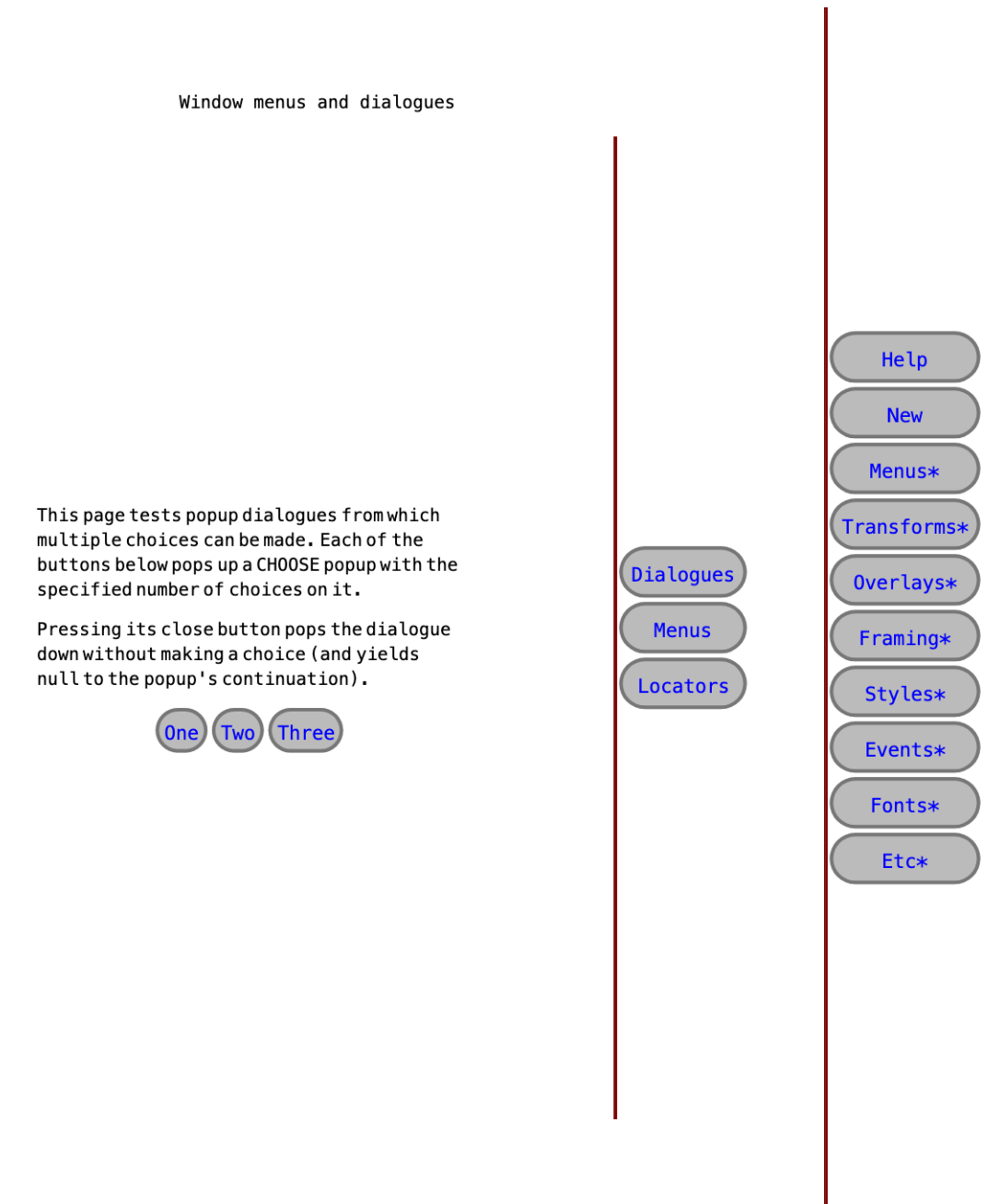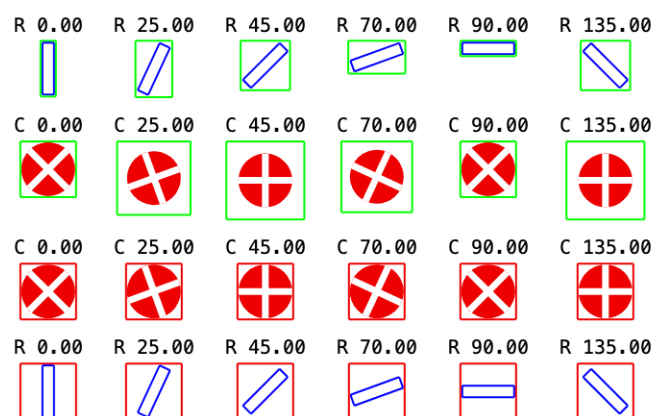
Overlays*

Framing*

Styles*

Events*

Fonts*

Etc*

Figure 4: the Dialogues subpage of Menus*

46

Turn transforms

The .turned transform with `tight=true` always yields a square bbox whose side is the larger of the two of the present glyph. For near-rotationally-symmetric glyphs this bbox fits more closely than the one yielded by `tight=false`.

Hereunder R denotes a rectangle, C denotes a circular glyph, and T denotes a triangle. Tight bounding boxes are shown in red, non-tight in green.

R 0.00   R 25.00   R 45.00   R 70.00   R 90.00   R 135.00

C 0.00   C 25.00   C 45.00   C 70.00   C 90.00   C 135.00

C 0.00   C 25.00   C 45.00   C 70.00   C 90.00   C 135.00

R 0.00   R 25.00   R 45.00   R 70.00   R 90.00   R 135.00

R/T turned d, for d in 0, −22.5, −45, −67.5, −90

tight=true   tight=true   non-tight
                                     bboxes with
                                     corresponding colours

Help

New

Menus∗

Turn   Transforms∗

Tight   Overlays∗

Skew   Framing∗

Mirror   Decoration∗

Events∗

Fonts∗

Etc∗
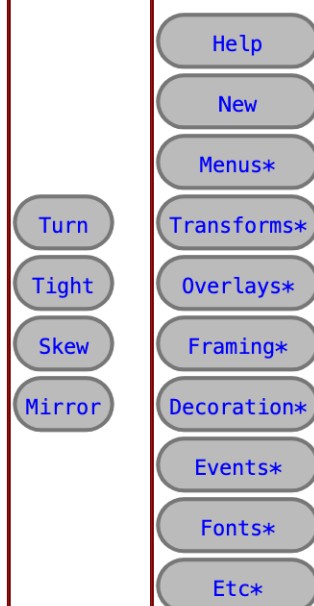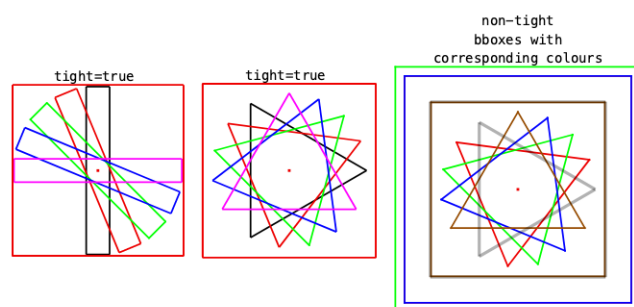
Figure 5: the Turn subpage of Transforms*
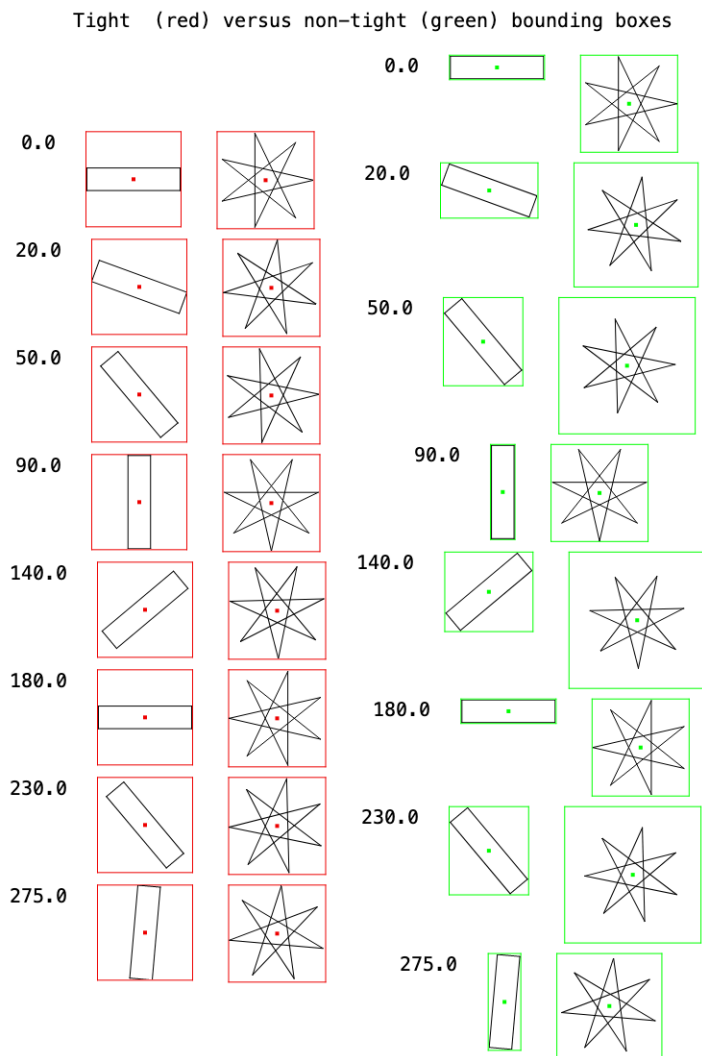
47

Figure 6: the Tight subpage of Transforms*
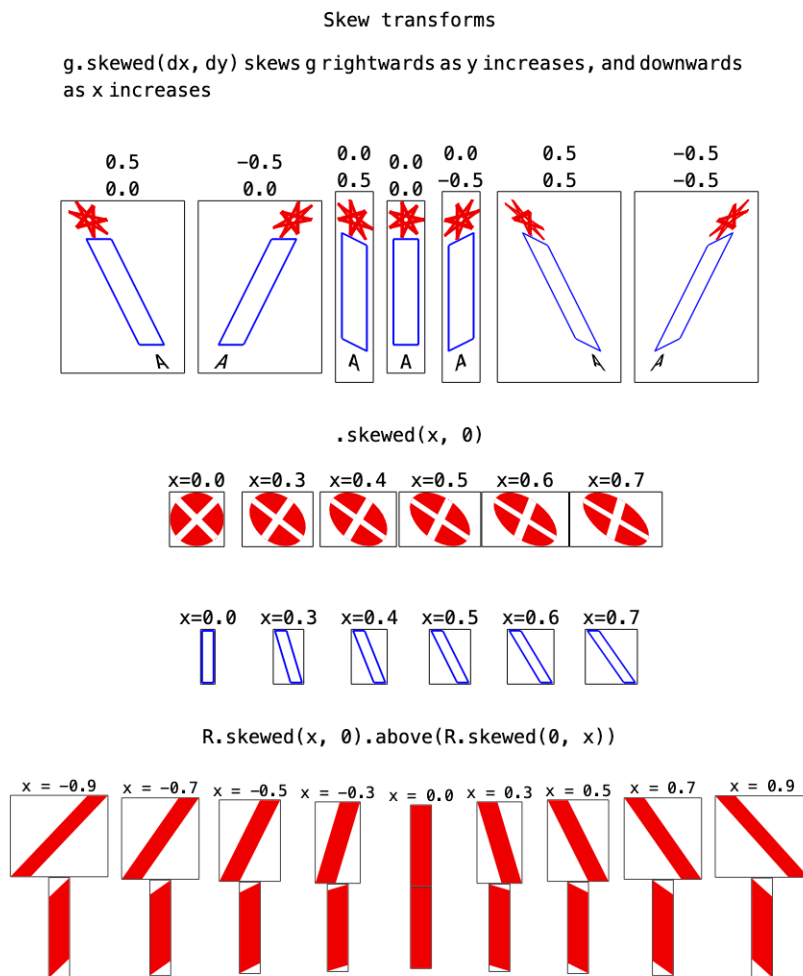
Skew transforms

g.skewed(dx, dy) skews g rightwards as y increases, and downwards as x increases

.skewed(x, 0)

R.skewed(x, 0).above(R.skewed(0, x))

Figure 7: the Skew subpage of Transforms*
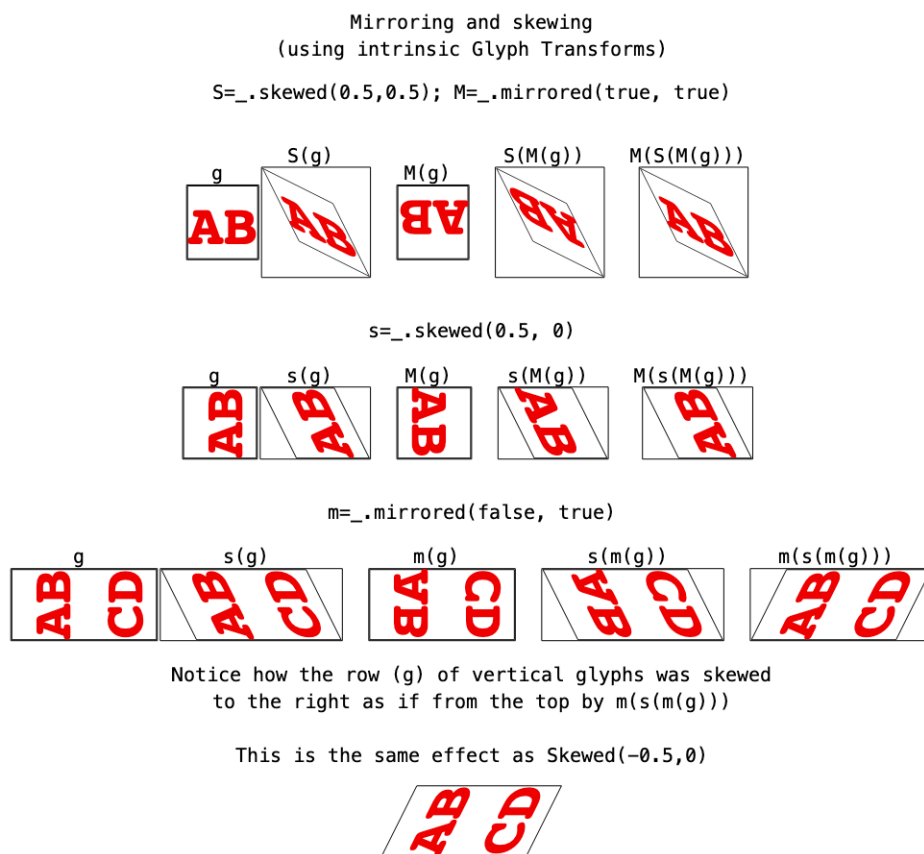
Figure 8: the Mirror subpage of Transforms*

Features implemented as overlay layers and annotations

This page tests annotation-style overlays, using the low-level annotation API.

The checkbox below enables/disables overlaying of a 10x10 grid on the whole of the current window, no matter what page/subpage is showing. The grid can be useful when exploring the dimensions of glyphs.

A checkbox always appears on the grid: pressing this disables the grid, which can only be re-enabled by the checkbox below.

Grid: ✖

The button below pops up an annotation overlay that points to the East wall of the window. The overlay stays up until its button is pressed, no matter what page/subpage of the app is showing.

Point to the East wall of the window

Dialogues

Menus

Locators

Annotation

Raw

Help

New

Menus∗

Transforms∗

Overlays∗

Framing∗

Decoration∗

Events∗

Fonts∗

Etc∗

Figure 9: the Annotation subpage of Overlays*
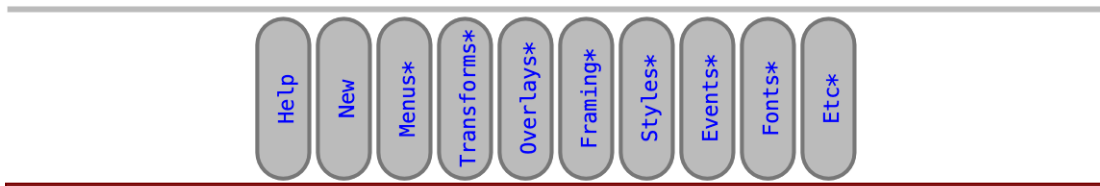
51

Features implemented as overlay layers and annotations

On this page we are testing modal dialogues implemented on overlays within the current window. You can exit the dialogue without making a choice by clicking on the topmost grey bar or typing ESC. You can shift the location of the dialogue by using the direction keys.

Each dialogue explains where it was intended to pop up (relative to the blue rectangle). The location of a dialogue that might be partly off screen are adjusted to make it (as far as possible) visible.

E/OKNO   W/OKNO   N/ABC

W/ABC   InFront/ABC   S/ABC

W/ABC[long]   InFront/ABC[long]   S/ABC[long]

Dialogues

Menus

Locators

Annotation

Raw

Figure 10: the Dialogues subpage of Overlays* (vertical tabs notebook)

52

Help New Menus* Transforms* Overlays* Framing* Styles* Events* Fonts* Etc*

Features implemented as overlay layers and annotations
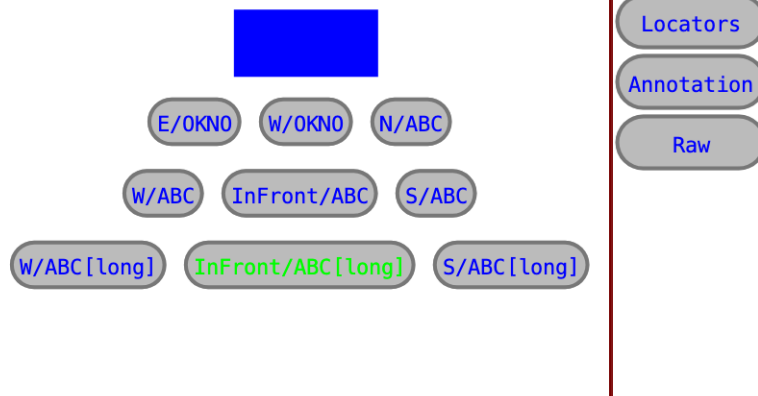
On this page we are testing modal dialogues implemented
on overlays within the current window. You can exit the
dialogue without making a choice by clicking on the
topmost grey bar or typing ESC. You can shift the
location of the dialogue by using the direction keys.

Each dialogue explains where it was intended to pop up
(relative to the blue rectangle). The location of a
dialogue that might be partly off screen are adjusted

Dialogues

InFront: This is a long pro-forma text
for a dialogue that I expect to be
justified properly in all the
dialogues.

An extraordinarily long button   a Bigger button that might cause jiggle   C

Raw

W/ABC   InFront/ABC   S/ABC

W/ABC[long]   InFront/ABC[long]   S/ABC[long]

Figure 11: a Popped-up dialogue (vertical tabs notebook)
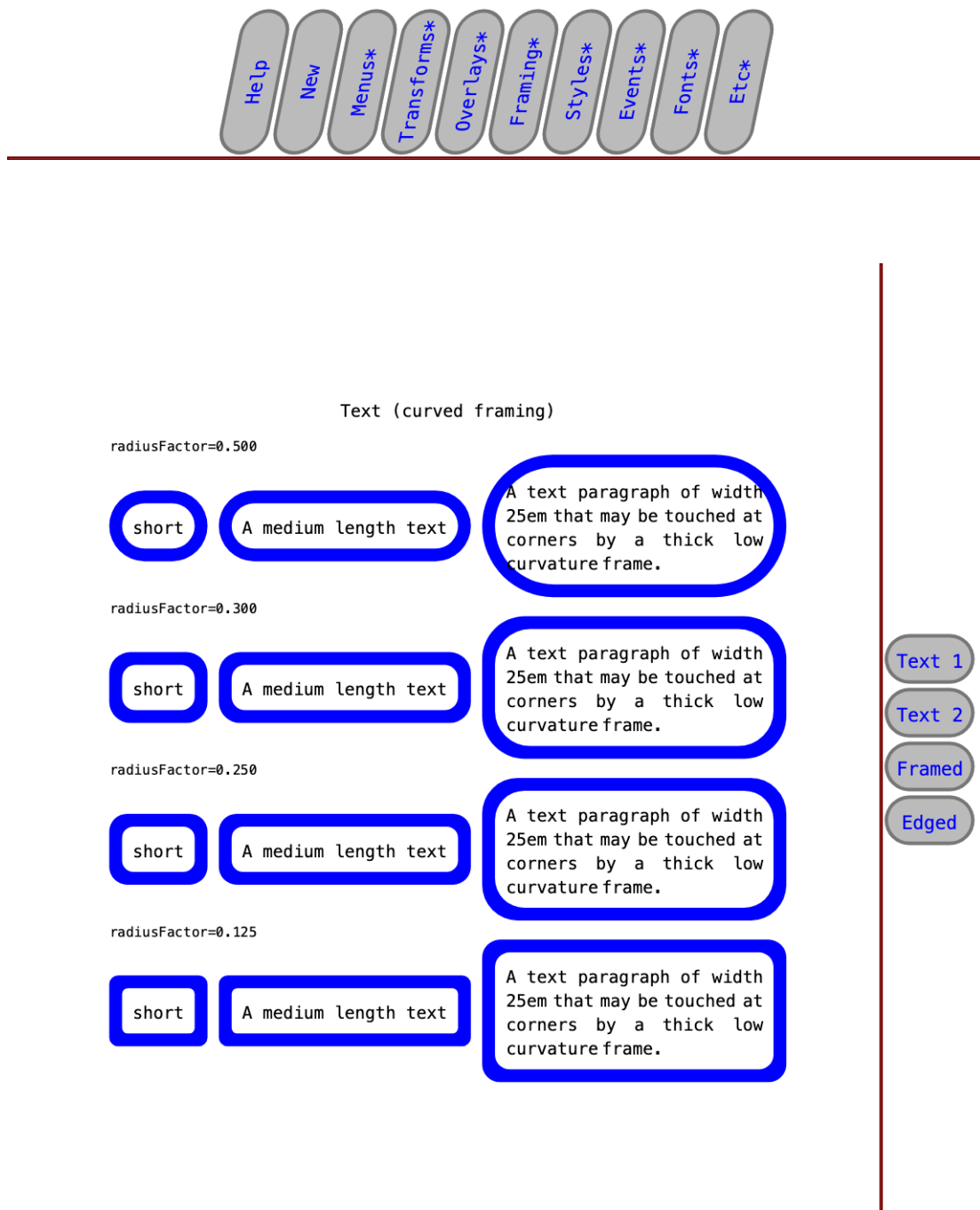
Figure 12: the Text1 subpage of Framing* (skewed tabs notebook)

Text (curved framing)

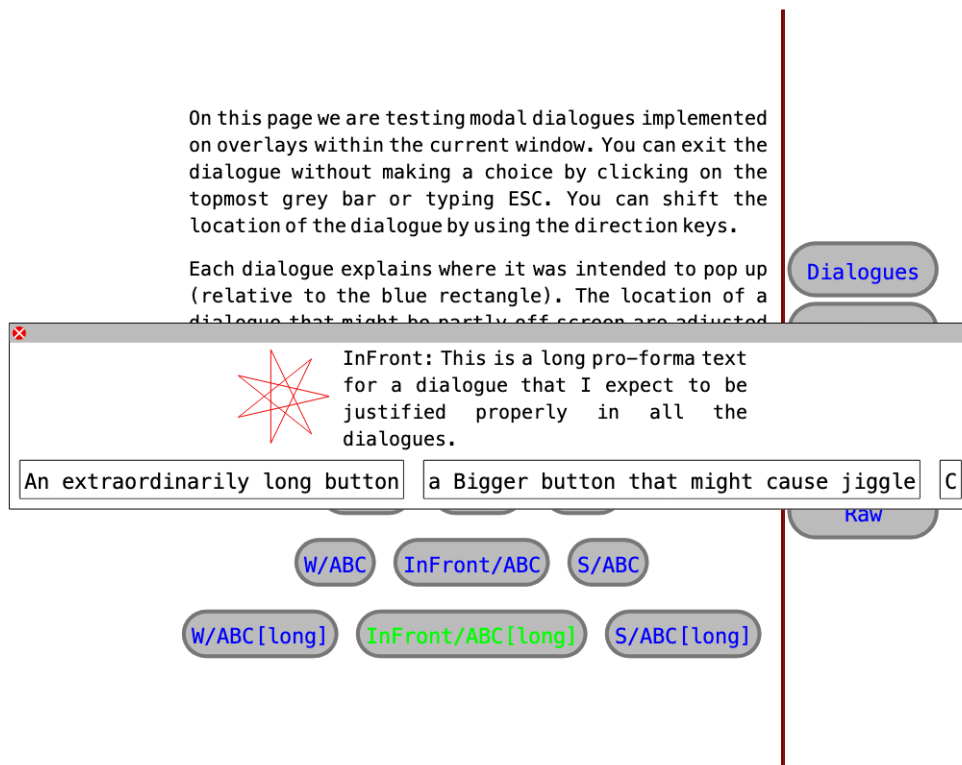radiusFactor=0.500

A medium length text

short

A tall text paragraph. It may be touched at corners by its frame, unless it has first been enlarged.

radiusFactor=0.300

A medium length text

short

A tall text paragraph. It may be touched at corners by its frame, unless it has first been enlarged.

radiusFactor=0.250

A medium length text

short

A tall text paragraph. It may be touched at corners by its frame, unless it has first been enlarged.

radiusFactor=0.125

A medium length text

short

A tall text paragraph. It may be touched at corners by its frame, unless it has first been enlarged.

Effects of enlargement by 25f before framing

radiusFactor=0.300

A medium length text

short

A tall text paragraph. It may be touched at corners by its frame, unless it has first been enlarged.

radiusFactor=0.125

A medium length text

short

A tall text paragraph. It may be touched at corners by its frame, unless it has first been enlarged.

Text 1

Text 2

Framed

Edged

Help

New

Menus*

Transforms*

Overlays*

Framing*

Decoration*

Events*

Fonts*
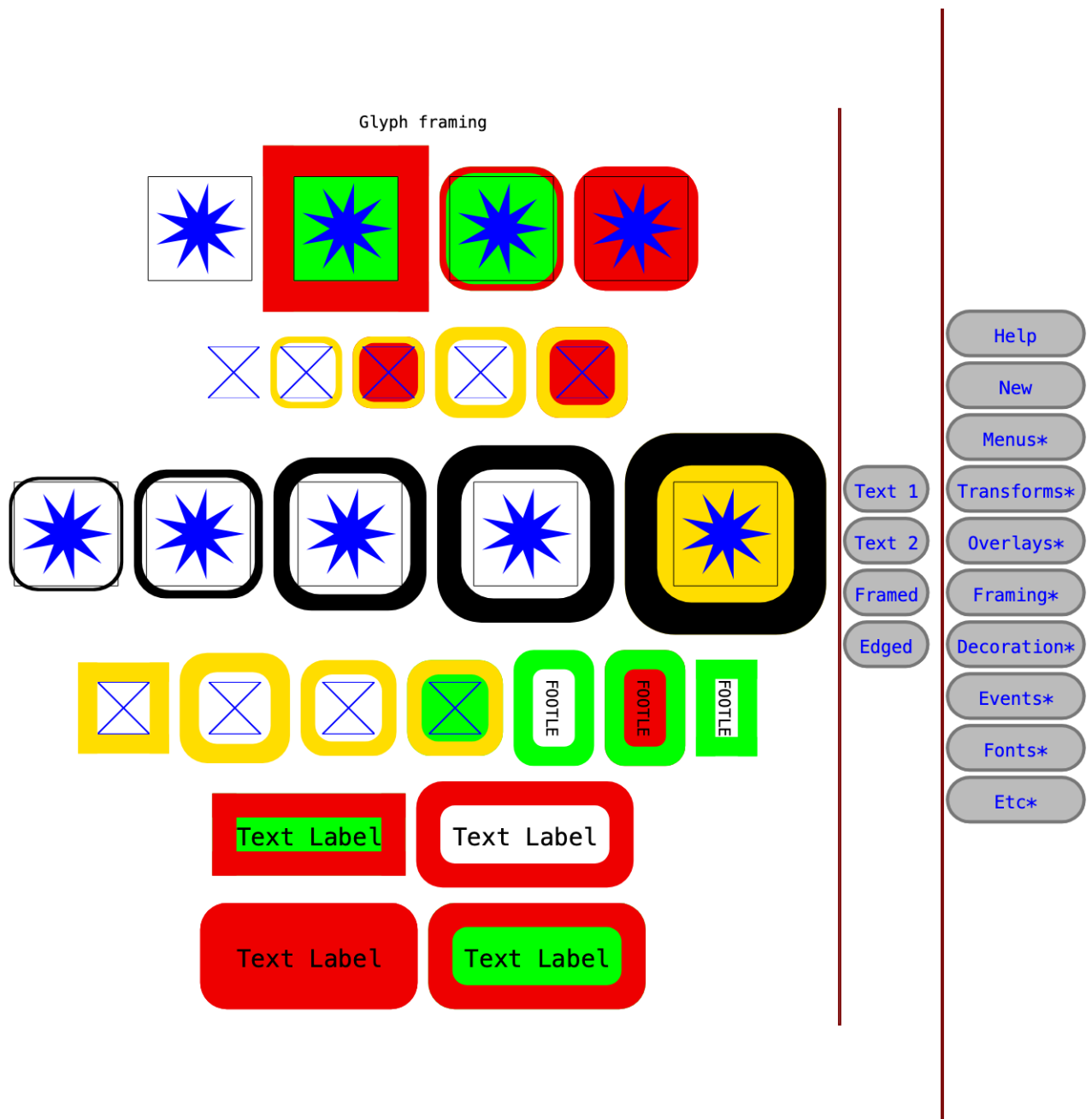
Etc*

Figure 13: the Text2 subpage of Framing*

55

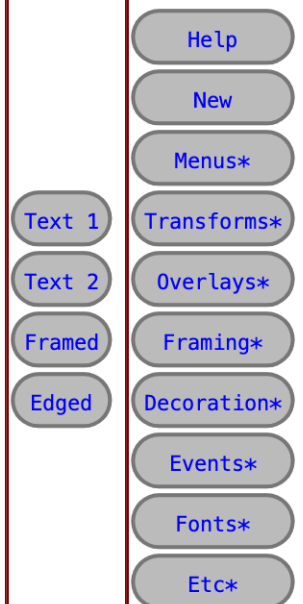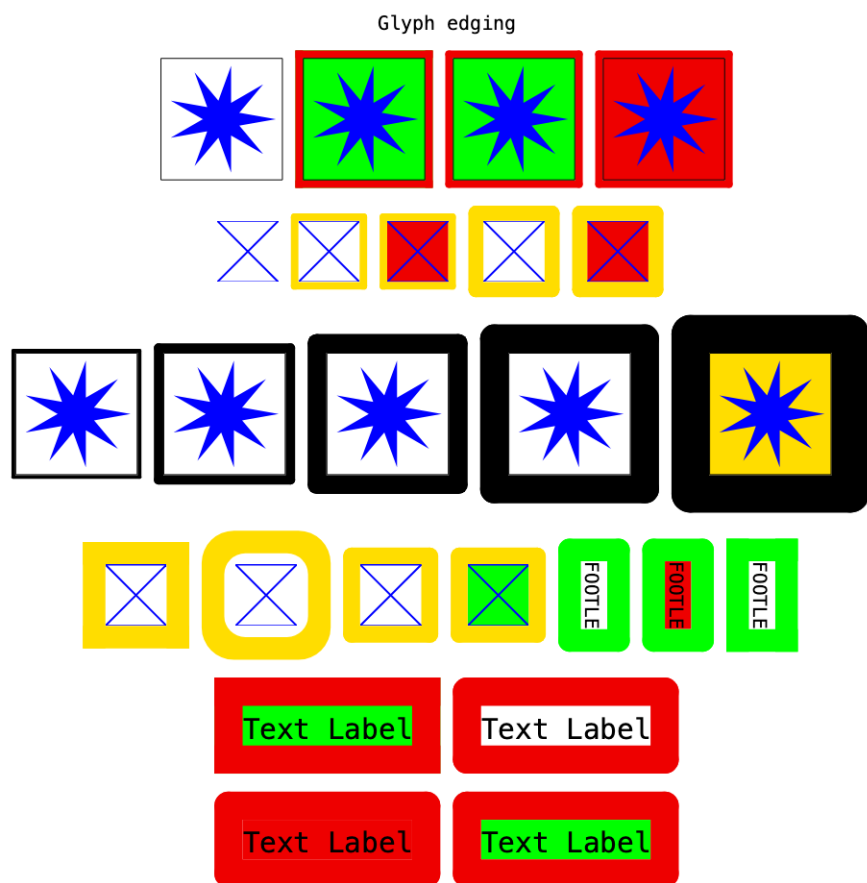Figure 14: the Framed subpage of Framing*

Figure 15: the Edged subpage of Framing*

```
The buttons here are all of the form
    styled.TextButton("..."){ _ => }(style)
where
    style =
        buttonStyle.copy(
            frame = Framed(fg = darkGrey(width=...),
                           bg = lightGrey,
                           enlarge = 0.25f,
                           radiusFactor = ...
    ))
```

(width=2, radiusFactor=0.5))

(width=4, radiusFactor=0.5)

(width=8, radiusFactor=0.5)

(width=10, radiusFactor=0.5)

(width=2, radiusFactor=0.25)

(width=4, radiusFactor=0.25)

(width=8, radiusFactor=0.25)

(width=10, radiusFactor=0.25)

Framed

Blurred

Shaded

Help

New

Menus*

Transforms*

Overlays*

Framing*

Styles*

Events*

Fonts*

Etc*

Figure 16: the Framed subpage of Styles*

```
The buttons here are all of the form
    styled.TextButton("..."){ _ => }(style)
where
    localStyle =
        buttonStyle.copy(up=buttonStyle.up.copy(fg=white))
    style =
        localStyle.copy(
            frame=
                Decoration.Blurred(...)
        ))
```

Framed

Blurred

Shaded

Blurred(fg=blue, blur=10f, spread=5f)

Blurred(fg=blue, blur=10f, spread=10f)

Blurred(fg=blue, blur=20f, spread=10f)

```
    style=buttonStyle.copy(frame=Decoration.Unframed)
```

This is an unframed button with an edge around it

Help

New

Menus∗

Transforms∗

Overlays∗

Framing∗

Styles∗

Events∗

Fonts∗

Etc∗

Figure 17: the Blurred subpage of Styles*

59

Mouse and keyboard events happening in the coloured frame
below are shown in the log beneath it. The most recent event
is also shown in the coloured frame.

Move events occur very frequently as the mouse traverses the
frame, and successive Moves in a sequence are usually for
physically close locations. Checking the box below
suppresses the second and subsequent Move reports in such a
sequence.

Shorten the log of Move sequences: ✔

00031 GlyphLeave

```
00000 GlyphEnter [keyboard grabbed])
00001 Move@(166.02295,30.622742)
00002 Move@(173.02295,12.622742)
00003 GlyphLeave
00004 GlyphEnter [keyboard grabbed])
00005 Move@(192.02295,62.62274)
00006 Move@(188.02295,94.62274)
00007 GlyphLeave
00008 GlyphEnter [keyboard grabbed])
00009 Move@(650.02295,14.622742)
00010 GlyphLeave
You are now eliding adjacent move events
00011 GlyphEnter [keyboard grabbed])
00012 Move@(584.02295,45.62274)
00015 GlyphLeave
00016 GlyphEnter [keyboard grabbed])
00017 Move@(514.02295,75.62274)
00031 GlyphLeave
```

Events

Windows

Help

New

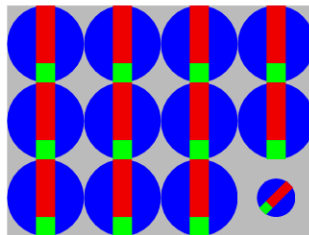Menus*

Transforms*

Overlays*

Framing*

Styles*

Events*

Fonts*

Etc*

Figure 18: the Events subpage of Events*

A grid of rotating buttons. Individual buttons are started/stopped by clicking on them; and can be started or stopped together with the Start all / Stop all toggle button. The speed of the last started/stopped button(s) can be adjusted with the Faster/Slower buttons.

Figure 19: the Animation subpage of Etc* (mid-animation)

buts = 8 blurred-frame buttons, data = 9 variable sized labels

Grid(fg=red(width=0), padx=20f, pady=2f).grid(width=5)(buts)

| Button 1 | Button 2 | Button 3 | Button 4 | Button 5 |
| Button 6 | Button 7 | Button 8 | | |

Grid(fg=red(width=0), padx=20f, pady=2f).grid(width=4)(buts)

| Button 1 | Button 2 | Button 3 | Button 4 |
| Button 5 | Button 6 | Button 7 | Button 8 |

Grid.grid(height=5)(buts) Grid.grid(height=4)(buts)

Button 1	Button 6
Button 2	Button 7
Button 3	Button 8
Button 4	
Button 5	

Button 1	Button 5
Button 2	Button 6
Button 3	Button 7
Button 4	Button 8

Grid(fg=red(width=0)).table(width=3)(data) -- variable height constant width 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)

Grid(fg=red(width=0)).table(height=3)(data) -- variable width constant 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)

Animation   Help
Grid   New
Blurred   Menus*
Scroll   Transforms*
OneOf   Overlays*
CheckBox   Framing*
             Styles*
             Events*
             Fonts*
             Etc*
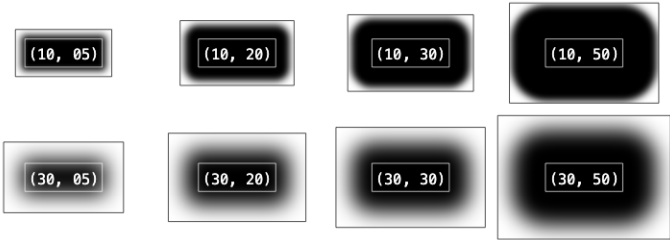
Figure 20: the Grid subpage of Etc*

Painting filled material with a blurred brush leads to the paint
coverage being enlarged by the blur of the brush

```
b30=black.blurred(30.00, 0.00, 0.00)
FilledRect(150, 30, fg=b30).enlargedBy(30f, 30f).framed()
FilledRect(150, 30, fg=b30).framed()

b15=blue.blurred(15.00, 0.00, 0.00)
filledStargon(5, fg=b15, C=64f, R=60f).framed()
```
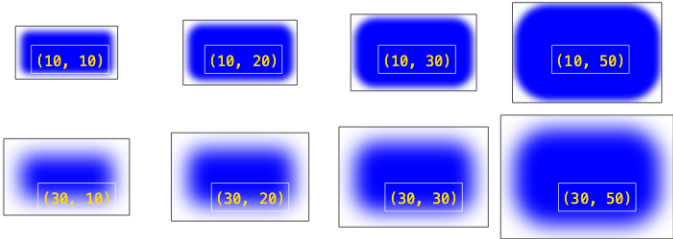
BlurredFrame(blur, spread)(...)

| (10, 05) | (10, 20) | (10, 30) | (10, 50) |

| (30, 05) | (30, 20) | (30, 30) | (30, 50) |

BlurredFrame(blur, spread, -blur/2f, -blur)(...)

| (10, 10) | (10, 20) | (10, 30) | (10, 50) |

| (30, 10) | (30, 20) | (30, 30) | (30, 50) |

Help

New

Animation   Menus∗

Grid   Transforms∗

Blurred   Overlays∗

Scroll   Framing∗

OneOf   Styles∗

CheckBox   Events∗

Fonts∗

Etc∗

Figure 21: the Blurred subpage of Etc*

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