69

Trestle Tutorial

Mark S. Manasse and Greg Nelson

May 1, 1992

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth, This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers,

Robert W. Taylor, Director

Trestle Tutorial

Mark S. Manasse and Greg Nelson

May 1, 1992

©Digital Equipment Corporation 1992.

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

Abstract

This is a tutorial introduction to programming with Trestle, a Modula-3 window system toolkit currently implemented over the X window system. We assume that you have some experience as a user of window systems, but no previous experience programming with X or other window systems. To run Trestle, you need a copy of SRC Modula-3 and an X server.

The tutorial begins with examples of programming using built-in Trestle interactors, and continues by showing you how to build your own interactors: both leaf interactors and interactors that contain their own sub-windows and modify their behavior.

The source code presented in the tutorial is shipped as part of the Modula-3 release from SRC, in the package `trestletutorial`. At SRC, you can fetch a copy of this by typing

```
cp -r /proj/m3/pkg/trestletutorial .
```

in your home directory. At other sites, you'll have to ask the people who installed SRC Modula-3 where they put the package sources. You will probably want to have a copy of the Trestle Reference Manual (SRC Report 68) nearby as you work through the tutorial.

The first few examples in the tutorial are programs; their source code is reproduced in subdirectories named after that program. The later examples are new classes of interactors For these, the subdirectories are named after the interactor, and contain both `src` and `test` subdirectories. The `src` directories contain the source code for the interface and implementation of the new interactor, and the `test` directory contains a simple program to exercise the interactor.

Contents

1 Hello Trestle 1

2 Split windows 2

3 Points, Rectangles, and Regions 4

4 Painting 7

5 Handling events: the Spot program 10

6 Tracking the mouse 13

7 The Fifteen Puzzle 19

8 Cards 23

9 About locking level 30

10 Asynchronous painting 32

11 Selections and event-time 37

12 A simple filter 43

13 An event-time filter 47

14 A proper split 49

15 The painting method 57

16 Conclusion 61

17 Solutions 61

18 Acknowledgments 66

References 67

Index 69

Trestle is a Modula-3 window system toolkit. Trestle is organized as a collection of interfaces structured around a central abstract type: a virtual bitmap terminal or VBT, which represents a share of the workstation's screen, keyboard, and mouse—a thing comparable to the viewers, windows, or widgets of other systems.

This report contains a series of example programs that introduce the properties of the `VBT` abstraction gradually. The programs are short and easy, and a number of exercises have been included. Programming with windows has a reputation for being difficult; we hope to show that it can be great fun. We assume some familiarity with Modula-3, but we will explain any unusual features of the language as we encounter them. If you have read a few chapters of one of the books on Modula-3 (e.g., [4] or [1]), working the exercises in this tutorial can be a good way to teach yourself the language. It would also be helpful to have a copy of the *Trestle Reference Manual* [3] close at hand.

This report can also serve as an introduction to object-oriented programming. Objects and inheritance are used almost constantly in programming user interfaces, because there are so many windows that are similar to one another but not quite the same. Method overriding is the basic tool for creating new window classes that are slight variations on existing classes. For example, a `ButtonVBT.T` has more than two dozen methods, but only four of them are supplied by the `ButtonVBT` module. The others are inherited from `ButtonVBT.T's` supertype, `Filter.T`. In turn, `ButtonVBT.T` has many subtypes corresponding to the different kinds of buttons that can appear in user interfaces.

For a more complete treatment, see the *Trestle Reference Manual* [3]. It would be handy to have a copy of this if you want to solve the exercises.

The later programs in the report illustrate some fairly advanced techniques for dealing with concurrent threads. Trestle was originally designed as a research project. One of the goals of the project was to determine how much a multiprocessor could speed up a window system. As a result, Trestle's locking is aggressively fine-grained. (See "A performance analysis of a multiprocessor window system" [2].)

1 Hello Trestle

To use Trestle, you need a copy of SRC Modula-3 and an X server for your system. If you have these, you may want to compile and run the example programs as you read the tutorial.

The first example program is in the file `Hello.m3`:

```
MODULE Hello EXPORTS Main;
IMPORT TextVBT, Trestle;

VAR v := TextVBT.New("Hello Trestle"); BEGIN
  Trestle.Install(v);
  Trestle.AwaitDelete(v)
END Hello.
```

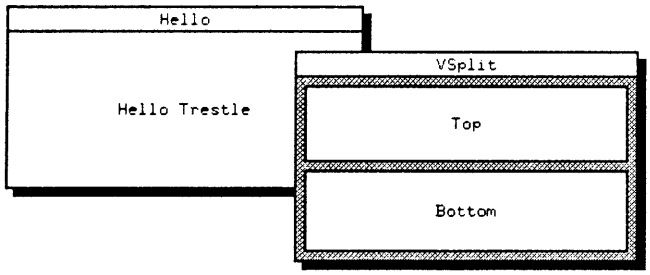

Figure 1: The first example program creates a window that displays the text `Hello Trestle.` The second shows how to split a window into two with an adjustable dividing bar. The title bars, outer borders, and black shadows are produced by the window manager, and will vary from system to system.

A `TextVBT` is a class of `VBT` that displays a text string. The example program creates a `TextVBT,` installs it on the screen, and waits for the user to delete the window. Figure 1 shows what the window looks like after resizing. When the user deletes the window with the window manager, `Trestle.AwaitDelete` will return and the program will exit. If the window manager provides no way to delete windows, the window will stay installed until the user kills or exits the program.

If you are using SRC Modula-3, you will probably want to use an *M3 makefile* to control the compilation and linking of your programs. For example, suppose that the source for `Hello` is in the file `Hello.m3` and the file named `m3makefile` in the same directory contains

```
import_lib (m3ui)
import_lib (m3X11R4)
import_lib (X11)
implementation (Hello)
program (Hello)
```

Then you can compile and link your program simply by typing `"m3make"` in that directory. You should use a separate directory and M3 makefile for each program that you build.

2 Split windows

`VBTs` are generally organized into a tree structure, with the root `VBT` representing the top-level application window. The internal nodes are called *split* `VBTs` or *parent* `VBTs:` they divide their display area between one or more child `VBTs` according to

some layout depending on the class of split. At the leaves of the tree are `VBTs` that contain no subwindows.

A typical application consists of a number of leaf `VBTs` whose behavior is specific to that application, together with other leaf `VBTs` that provide buttons, scrollbars, and other *interactors,* all held together by a tree of splits that define the geometric layout.

One of the most important splits is the `HVSplit`, in which the children are stacked horizontally or vertically. The following program demonstrates a vertical `HVSplit` whose division of space is adjustable by the user:

```
MODULE VSplit EXPORTS Main;
IMPORT Trestle, TextVBT, BorderedVBT, HVSplit, Axis,
  HVBar, Pixmap;

VAR v :=
  BorderedVBT.New(
    HVSplit.Cons(Axis.T.Ver,
      BorderedVBT.New(TextVBT.New("Top")),
      HVBar.New(size := 3.0, txt := Pixmap.Gray),
      BorderedVBT.New(TextVBT.New("Bottom"))),
    size := 3.0,
    txt := Pixmap.Gray);

BEGIN
  Trestle.Install(v);
  Trestle.AwaitDelete(v)
END VSplit.
```

The call `HVSplit.Cons(Axis.T.Ver, ch1, ch2, ...)` creates a vertical split with children `ch1, ch2, ...` in top-to-bottom order. (`Axis.T.Ver` and `Axis.T.Hor` are the Trestle names for the vertical and horizontal axes. The first argument to `HVSplit.Cons` determines whether to create a horizontal or vertical split.)

A *filter* is a split with one child. The call `BorderedVBT.New(ch, size:= s, txt : = t)` creates a filter in which the parent paints a border around the child `ch;` the border is `s` millimeters wide and is painted with the texture `t.` (A *pixmap* is a rectangular array of pixels; a *texture* is an infinite pattern of pixels obtained by tiling the plane with a pixmap. A *bitmap* is a pixmap in which the pixels are of depth one.) In the `VSplit` program, the outer call to `BorderedVBT.New` produces a wide gray border the matches the adjusting bar; the two inner calls default the size and texture arguments, producing a thin solid border of the user's default foreground color. Figure 1 shows what the window looks like.

The call `HVBar.New(size := s, txt := t)` produces an adjusting bar for an `HVSplit;` the bar is `s` millimeters wide and is painted with the texture `t.` The user can drag the bar with the mouse to adjust the sizes of the other children.

To support users who have more than one type of display on their desks, the screentype of a `VBT` is not constant, but changes as the user moves the `VBT` from screen

to screen. Trestle supplies resources like fonts, pixmaps, cursor shapes, and painting operation codes in both screen-dependent and screen-independent forms. The standard Trestle splits and interactors all use screen-independent resources and take dimensions specified in millimeters, so that they will look about the same when they move from screen to screen. For example, the value `Pixmap.Gray` in the program above is a screen-independent pixmap: it varies with the screentype to produce a uniform effect on all screens. Screen-dependent resources are provided for sophisticated applications that depend on features available only on a particular screen—for example, color map animation.

Exercise 1. The incomplete program `Monster` shown below creates a depth 8 tree of alternating adjustable horizontal and vertical splits (Figure 2). The 256 leaves of the tree are labeled with the numbers 0 through 255. The heart of the program is a recursive procedure that constructs a subtree of the monstrous split. The base case of the procedure just returns a bordered `TextVBT`. The recursive case of the procedure is left for you to write as an exercise. (If you have SRC Modula-3, you will find the incomplete program in the tutorial directory; just edit it and type `"m3make".`)

Hint: The array `Axis.Other[]` exchanges `Axis.T.Hor` and `Axis.T.Ver`.

```
MODULE Monster EXPORTS Main;
IMPORT TextVBT, Trestle, HVSplit, Axis, HVBar, Fmt,
  VBT, BorderedVBT, Pixmap;

PROCEDURE New(lo, hi: INTEGER; hv: Axis. T) : VBT.T =
  BEGIN
    IF hi - lo = 1 THEN
      RETURN
        BorderedVBT.New(TextVBT.New(Fmt.Int(lo)))
    ELSE
       (* You fill in this part *)
    END
  END New;

VAR v : = BorderedVBT.New(
            New(0, 256, Axis.T.Hor),
            size := HVBar.DefaultSize,
            txt := Pixmap.Gray);

BEGIN
  Trestle.Install(v);
  Trestle.AwaitDelete(v)
END Monster.
```

3 Points, Rectangles, and Regions

The interfaces `Point`, `Rect`, and `Region` define dozens of useful operations on integer lattice points and sets thereof. We won‘t present complete listings of the

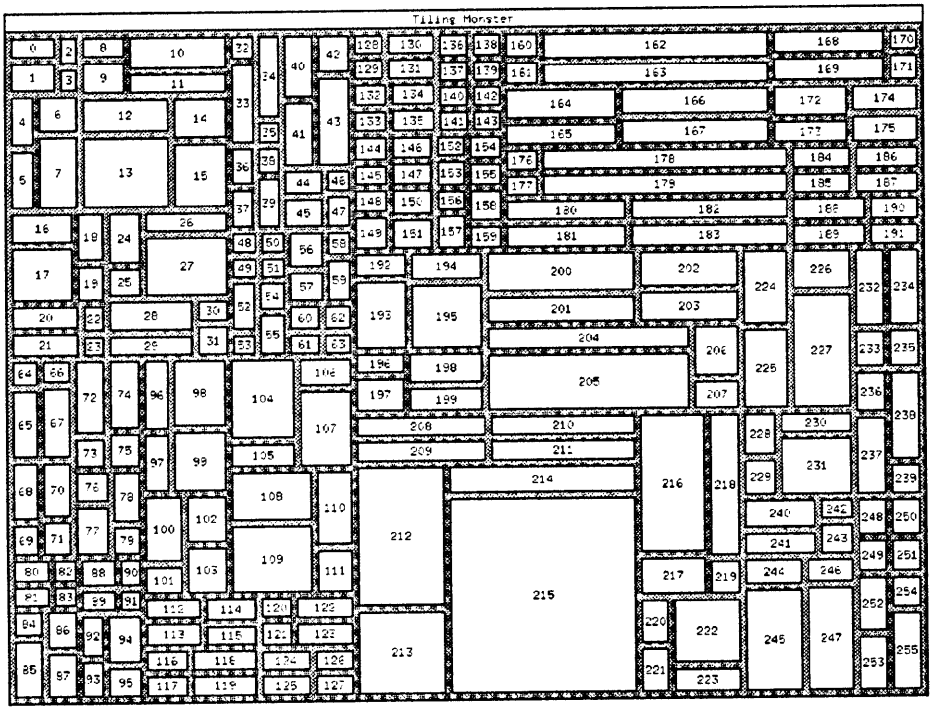

Figure 2: A nightmare if you don't like tiling windows (or even if you do). The Tiling Monster is a tree of adjustable binary splits eight levels deep.

interfaces, since they are long and consist mostly of procedures whose functions are obvious from their names, but we will briefly introduce the basic data types and most common routines.

A `Point.T` is a two-dimensional point with integer coordinates; the horizontal and vertical coordinates of a point `pt` are named `pt.h` and `pt.v`. The procedures `Point.Add` and `Point.Sub` provide component-wise addition and subtraction on points.

A `Rect.T` is a set of points lying in a rectangle whose sides are parallel to the coordinate axes. The directions on the screen are named after the compass points, with north at the top. A point `pt` lies in a rectangle `rect` if

- `pt.h` is in `[rect.west .. rect.east - 1]`
- `pt.v` is in `[rect.north .. rect.south - 1]`

Notice that `h` increases west to east; `v` increases north to south (for which we offer our apologies to Descartes).

Here are some useful operations on rectangles:

`Rect.NorthWest(rect), Rect.NorthEast(rect)`
`Rect.SouthWest(rect), Rect.SouthEast(rect)`
The four vertices of `rect`.

`Rect.Middle(rect)`
The center point of `rect`.

`Rect.IsEmpty(rect)`
Tests whether `rect` *is empty.*

`Rect.Member(pt, rect)`
Tests whether the point `pt` *lies in the rectangle* `rect`.

`Rect.Overlap(rect1, rect2)`
Tests whether the two rectangles have any point in common.

`Rect.FromPoint(pt)`
The rectangle containing only the point `pt`.

`Rect.Meet(rect1, rect2)`
The intersection of the two rectangles.

`Rect.Add(rect, pt)`
The translation of `rect` *by the point* `pt` *regarded as a vector.*

`Rect.Empty, Rect.Full`
The empty rectangle and the largest representable rectangle.

A `Region.T` represents an arbitrary set of integer lattice points, compactly encoded as a set of disjoint rectangles. Here are some useful operations on regions:

`Region.IsEmpty(rgn)`
Tests whether `rgn` *is empty.*

`Region.Member(pt, rgn)`
Tests whether point `pt` *lies in the region* `rgn`.

`Region.Add(rgn, pt)`
The translation of `rgn` *by the point* `pt` *regarded as a vector.*

`Region.JoinRect(rect, rgn)`

The set of points that are in `rect` *or* `rgn`*.*

`Region.Difference(rgn1, rgn2)`
The set of points that are in `rgn1` *and not in* `rgn2`*.*

`Region.FromRect(rect)`
The set of points in the rectangle `rect`*.*

`Region.Empty, Region.Full`
The empty region and the largest representable region.

For example, here is a procedure that we will find useful later: it creates a circular region of a given radius, centered at the origin:

```
PROCEDURE Circle(r: REAL) : Region. T =
  VAR res : = Region.Empty;
  BEGIN
    FOR h := FLOOR(-r) TO CEILING (r) DO
      FOR v := FLOOR(-r) TO CEILING (r) DO
        IF h * h + v * v <= FLOOR(r * r) THEN
          WITH rect = Rect.FromPoint(Point.T{h, v}) DO
            res := Region.JoinRect(rect, res)
          END
        END
      END
    END;
    RETURN res
  END Circle;
```

This loop simply tests each relevant integer lattice point for membership in the circle and adds it to the region if appropriate. The procedure won't win any prizes for efficiency, but it works. The final region is compactly represented, even though it was built up from singleton rectangles, because the operations in the `Region` interface always compact their results.

4 Painting

Changing the visible contents of a Trestle window's screen is called *painting*. In general, every `VBT` painting procedure is determined by

- a destination, which is a set of screen pixels;
- a source, which is conceptually an infinite array of pixels, together with a rule for associating source and destination pixels;

- an operation `op`, which is a function that takes a destination and source pixel and produces a destination pixel.

The effect of a painting procedure is to set `d := op(d, s)` for each destination pixel `d` and corresponding source pixel `s`.

Thus, in general, the final value of a destination pixel depends on its initial value and the value of the corresponding source pixel. But many painting operations ignore the source pixel; such an operation is called a *tint*. If `op` is a tint, we just write `op(d)` instead of `op(d, s)`. For example, the two most basic painting operations are the tints `PaintOp.Bg` and `PaintOp.Fg`, defined by

```
PaintOp.Bg(d)  = the screen's background pixel
PaintOp.Fg(d)  = the screen's foreground pixel
```

They ignore both the source and the initial value of their destination: painting with `Bg` sets the destination pixels to the background pixel value, while painting with `Fg` sets them to the foreground pixel value. The background and foreground pixels vary from screentype to screentype; you can think of `Bg` as white and `Fg` as black (unless you prefer video-reversed screens).

Another useful tint is `PaintOp.Swap`, which exchanges the screen's foreground and background pixel. If it is applied to any other pixel the result depends on the screentype, but for any pixel `d` of any screentype it satisfies

```
Swap(Swap(d)) = d
Swap(d) # d
```

Swap can be used on general screens the way `XOR` is used on bitmap screens.

Trestle also supplies the tint `PaintOp.Transparent`, defined by

```
Transparent(d) = d
```

for any pixel `d`. `Transparent` may seem useless at first—but the number zero also seems useless until you need it.

There are a variety of other tints, of which we mention one:

`PaintOp.FromRGB(r, g, b)`

Return a tint that sets a pixel to the color whose mixture of red, green and blue is given by `(r, g, b)`.

The tint will work on both color-mapped and true-color displays. There are other arguments that control what the tint will do on black-and-white displays, but we won't explain them here.

So much for tints. Next in importance are those painting operations that are built from pairs of tints. For example, the operation `BgFg` is defined by

```
PaintOp.BgFg(d, 0)  = the screen's background pixel
PaintOp.BgFg(d, 1)  = the screen's foreground pixel
```

`BgFg` should be used with a source that is one-bit deep, such as a font; the effect is to copy the source to the destination, interpreting `0` as background and `1` as foreground.

Similarly, we have `TransparentFg`, defined by

```
PaintOp.TransparentFg(d, 0) = d
PaintOp.TransparentFg(d, 1) = the screen's foreground pixel
```

`TransparentFg` is also used with sources that are one-bit deep; the effect is to copy the source to the destination, leaving the bits that correspond to `0's` unchanged and setting the bits that correspond to `1's` to the foreground pixel.

`BgFg` and `TransparentFg` are examples of the following general rule: if `X` and `Y` are two of the tints `Bg, Fg, Swap,` and `Transparent,` then Trestle supplies the painting operation `XY` defined by `XY(d, 0) = X(d)` and `XY(d, 1) = Y(d).`

Finally, we mention one painting operation that is neither a tint nor formed from a pair of tints:

```
PaintOp.Copy(d, s) = s
```

`Copy` should be used only when the source pixels are of the same type as the destination pixels; for example, when copying from one part of the screen to another.

Trestle has painting procedures that operate on rectangles, regions, stroked and filled paths, and other exotic shapes, but for now we will just describe the following three:

```
VBT.PaintTint(v, clip, op)
```

Set `d := op(d)` for each pixel `d` in the screen of the `VBT v` that lies in the rectangle `clip`. The operation must be a tint.

```
VBT.PaintRegion(v, rgn, op)
```

Set `d := op(d)` for each pixel `d` in the screen of the `VBT v` that lies in the region `rgn`. The operation must be a tint.

```
VBT.Scroll(v, clip, delta, op := PaintOp.Copy)
```

Translate a rectangle of `v's` screen by `delta` and use it as a source for the operation `op` applied to each pixel in clip.

That is, `VBT.Scroll` sets `d := op(d, s)` for each pair of pixels `d, s` such that `d` lies in the given clipping rectangle, `d` and `s` both lie in `v's` screen, and the displacement from `s` to `d` is the vector `delta.` If `op` is defaulted to `Copy,` the effect is to translate a rectangle of pixels from `v's` screen by `delta,` overwriting the contents of the `clip` rectangle. The operation must apply to source pixels and destination pixels of the same type.

5 Handling events: the Spot program

A `VBT` is an object whose methods define its response to user events. For example, if the user reshapes a window, the system will call the window's `reshape` method; if the user exposes some part of the window, the system will call the window's `repaint` method, and if the user clicks the mouse over the window, the system will call the window's `mouse` method.

As an example, we will write a program called `Spot` that displays a single spot on the screen (see Figure 3). The user can move the spot to a new position by clicking with the mouse. When the window is reshaped, the spot moves to the middle of the new screen.

A `VBT.Leaf` is a `VBT` that responds to events by ignoring them. The `Spot` program defines a subtype of `VBT.Leaf` in which the `mouse`, `repaint`, and `reshape` methods are overridden with procedures that behave as follows:

- On a mouse click, move the spot to the position of the click.
- On an exposure, repaint the white background and the spot.
- On a reshape, move the spot to the center of the new screen and repaint.

The subtype also has a data field to record the region currently occupied by the spot. In our first version of the program, the spot will be a circle 10.5 pixels in diameter. So far the program is:

```
MODULE Spot EXPORTS Main;

IMPORT VBT, Trestle, Region, Rect, Point, PaintOp;

TYPE
  SpotVBT = VBT.Leaf OBJECT
    spot : Region.T
  OVERRIDES
    mouse := Mouse;
    repaint := Repaint;
    reshape := Reshape
  END;

VAR
  v := NEW(SpotVBT, spot := Circle (10.5)) ;

  (* Definitions of Circle, Repaint,
     Reshape, and Mouse. *)

BEGIN
  Trestle.Install(v);
  Trestle.AwaitDelete(v)
END Spot.
```

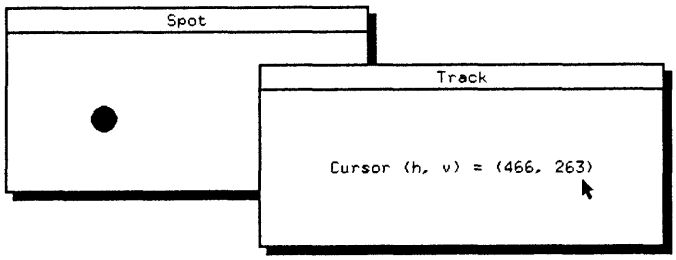

Figure 3: Clicking in the Spot window moves the spot; rolling the cursor in the Track window updates the displayed coordinates continuously.

We described the `Circle` procedure in the section on regions, so all that remains is to describe the procedures `Mouse`, `Repaint`, and `Reshape`.

Trestle calls a window's `repaint` method whenever a region of the window has been exposed. The `repaint` method for the `Spot` window is:

```
PROCEDURE Repaint(v: SpotVBT; READONLY rgn: Region.T) =
  BEGIN
    VBT.PaintRegion(v, rgn, PaintOp.Bg);
    VBT.PaintRegion(v,
      Region.Meet(v.spot, rgn), PaintOp.Fg)
  END Repaint;
```

The argument `rgn` is the region to be repainted.

The rectangular extent of a window is called its *domain.* Trestle calls a window's `reshape` method whenever its domain changes. The `reshape` method for the `Spot` window is:

```
PROCEDURE Reshape(v: SpotVBT;
    READONLY cd: VBT.ReshapeRec) =
  VAR delta :=
    Point.Sub(
      Rect.Middle(cd.new),
      Rect.Middle(v.spot.r));
  BEGIN
    v.spot := Region.Add(v.spot, delta);
    Repaint(v, Region.Full)
  END Reshape;
```

The `cd` argument to the method contains several fields, but the only one that matters here is `cd.new`, which is the new domain of the window. To move the spot to the center of the new domain, the method simply translates the spot region by the vector `delta` from the current center of the spot to the center of the new domain. Then it repaints the entire window by passing `Region.Full` to the `repaint` method.

When a window moves it may be faster to copy the pixels from the old domain to the new domain instead of recomputing the new pixels from scratch. Therefore, the `cd` argument to the reshape method includes a rectangle `cd.saved`, which is the portion of the old domain of the window that Trestle has preserved. The `reshape` method can copy the pixels in this rectangle to the appropriate portion of the new domain, using `VBT.Scroll`. This optimization is important for large areas whose contents are expensive to recompute, but for the `Spot` program the saved pixels aren't worth bothering about.

Exercise 2. Change the `reshape` method to preserve the position of the spot relative to the south-west comer of the window, instead of moving it to the middle. (Hint: the record `cd` contains a field `cd.prev`, which is the previous domain of the VBT.)

Trestle calls the mouse method of a window whenever the user clicks the mouse over the window. The `mouse` method for the `Spot` window is:

```
PROCEDURE Mouse(v: SpotVBT; READONLY cd: VBT.MouseRec) =
  VAR delta: Point.T;
  BEGIN
    IF cd.clickType = VBT.ClickType.FirstDown THEN
      delta :=
        Point.Sub(cd.cp.pt, Rect.Middle(v.spot.r));
      v.spot := Region.Add(v.spot, delta);
      Repaint(v, Region.Full)
    END
  END Mouse;
```

The `cd` argument to `Mouse` contains several fields.

The field `cd.clickType` is `FirstDown` if the button went down when no other buttons were down, `OtherDown` if it went down when some other button(s) were already down, `LastUp` if it went up when all other buttons were up, and `OtherUp` if it went up when some other button(s) were still down. The `Spot` program responds only to clicks of type `FirstDown`.

The field `cd.cp` is the cursor position where the mouse button went down. A cursor position has various fields, but the only one that is of interest here is the field `cp.pt`, which is the position of the cursor as a `Point.T`.

Although we have not used it in the spot program, every `VBT v` has a method `v.rescreen` that the system calls whenever `v's` screentype changes. The method is passed a record `cd` of type `VBT.RescreenRec` containing several fields; `cd.st` is the new screentype. Whenever `v's` screentype changes, `v's` domain is automatically reshaped to the empty rectangle `Rect.Empty`: the `reshape` method is however not called; reshaping to empty is implicit in being rescreened. Typically the next event will be a reshape to a non-empty rectangle on the new screen.

Exercise 3. Modify the program so that its spot will be about eight millimeters in diameter, regardless of the screen resolution. (Hint: `VBT.MMToPixels(v, mm, ax)` returns the number of pixels (as a `REAL`) that corresponds to `mm` millimeters on

`v's` screentype in the axis `ax.)` For the purpose of this exercise, you can assume that the screentype has the same resolution in the horizontal and vertical axes.

6 Tracking the mouse

Our next example program is called `Track,` because it installs a window that tracks the cursor. When the cursor is anywhere over the window, the coordinates of the cursor are displayed in the center of the window. When the cursor is outside the window, the text "Cursor gone" is displayed.

The track window behaves exactly like a `TextVBT` with respect to repaints, reshapes, and mouse clicks; it behaves differently only when the system delivers a cursor position. Therefore it inherits most of its methods from `TextVBT,` overriding only the `position` method and declaring one new method of its own, for initialization:

```
MODULE Track EXPORTS Main;

IMPORT VBT, Trestle, TextVBT, Fmt;

TYPE
  TrackVBT = TextVBT.T OBJECT
  METHODS
    init(): TrackVBT := Init
  OVERRIDES
    position := Position
  END;

  (* Definitions of Position and Init *)

VAR v := NEW(TrackVBT).init(); BEGIN
  Trestle.Install(v);
  Trestle.AwaitDelete(v)
END Track.
```

The `TrackVBT` must initialize its supertype `TextVBT,` since a newly-allocated `TextVBT` must be initialized with some text string before it can be used. (In the previous example program, the `SpotVBT` did not initialize its supertype `VBT.Leaf,` since a newly-allocated `VBT.Leaf` is ready to use.)

Every `VBT` class that needs initialization after allocation provides an `init` method for doing so. The arguments to the method depend on the class. By convention, the `init` method also returns the `VBT` after initializing it. The `init` method for a subtype is responsible for calling the `init` method of its supertype, if this is necessary. Here is the `init` method for a `TrackVBT:`

```
PROCEDURE Init(v: TrackVBT): TrackVBT =
  BEGIN
    EVAL TextVBT.T.init(v, "Cursor gone");
```

```
    RETURN v
  END Init;
```

The call `TextVBT.T.init(v, t)` will initialize a newly-allocated `TextVBT v` to display the text `t`. (It would also be legal to simply return the result from `TextVBT.T.init`, which will be `v`, but it would cost an unnecessary implicit `NARROW`.)

In general, if a `VBT` class `Cl.T` has an associated procedure `Cl.New`, then by convention the call `Cl.New(args)` means the same thing as `NEW(Cl.T).init(args)`. So if you want to, you can write `NEW(BorderedVBT.T).init(ch)` instead of `BorderedVBT.New(ch)`, and similarly for `TextVBT`, `HVBar`, and all the other `VBT` classes.

All that remains is to specify `TrackVBT's position` method.

To track the cursor, you specify a region called a *cage*. Generally the region should contain the current cursor position. Trestle waits for the cursor to leave the cage, and reports this event by calling your `position` method, which sets a new cage containing the new position, and so it goes.

If you're not interested in tracking the cursor at all, set the cage to the special value `VBT.EverywhereCage`.

If you want to know when the cursor leaves your window, set the cage to be the window's domain.

If you want to know when it comes back, set the cage to be the special cage `VBT.GoneCage`, which contains all positions outside the window, including the artificial position `gone`.

If you're interested in any motion of the cursor, however tiny, set the cage to be the single point containing the current cursor position: then the next motion of the mouse will generate a position code.

Here is the position procedure for the `Track` program:

```
PROCEDURE Position(v: TrackVBT;
    READONLY cd: VBT.PositionRec) =
  BEGIN
    IF cd.cp.gone THEN
      TextVBT.Put(v, "Cursor gone");
    ELSE
      TextVBT.Put(v,
        Fmt.F("Cursor(h, v) = (%s, %s)"
          Fmt.Int(cd.cp.pt.h), Fmt.Int(cd.cp.pt.v)));
    END;
    VBT.SetCage(v, VBT.CageFromPosition(cd.cp))
  END Position;
```

The only field of `cd` that matters in this program is `cd.cp`, which is a cursor position. If `cd.cp.gone` is true, then the system is reporting that the cursor has left the window. In this case, the program changes the text of the `TextVBT` to be "Cursor gone", using

the procedure `TextVBT.Put`, and waits for the cursor to return to the window by setting its cage to `VBT.GoneCage`. `VBT.CageFromPosition(cp)` returns a cage that contains only the position `cp` —that is, `GoneCage` if `cp` is gone, and a one-point rectangular cage otherwise.

If `cd.cp.gone` is false, then the system is reporting that the cursor is at position `cd.cp.pt`. In this case the program constructs the text string representing the position of the cursor (using procedures from the standard Modula-3 `Fmt` interface), uses `TextVBT.Put` to display the string, and sets a one-point cage around this cursor position, so that the system will report the next cursor motion.

Finally, here is an exercise to try your hand at.

Exercise 4. The incomplete program `Draw` listed below is a simple drawing program that allows you to draw line segments by pressing a mouse button at the start point, dragging the mouse, and releasing it at the endpoint. During dragging, the end point of the segment follows the cursor, pulling the segment like a rubber band. The body of the position procedure is left blank for you to complete. Figure 4 shows what the window looks like.

In order to handle repaints, the program keeps track of the line segments in a variable `path` of type `Path.T`. Here is what you need to know about paths for this exercise:

`Path.Reset(path)`

Set `path` to be empty.

`Path.MoveTo(path, p); Path.LineTo(path, q)`

Add the segment `(p, q)` to `path`.

`Path.Translate(path, delta)`

Return the result of adding `delta` to all vertices of all segments in `path`.

`VBT.Stroke(v, clip, path, op)`

Apply the tint `op` to each pixel of `v` that lies in the given clipping rectangle and on some segment of the given path.

`VBT.Line(v, clip, p, q, op)`

Like `VBT.Stroke` for a path containing only the segment `(p, q)`.

The program also uses push buttons:

`ButtonVBT.New(ch, proc)`

Return a filter that looks like its child `ch`, but when the user clicks on it, the action procedure `proc` will be called.

The action procedure is passed the button itself and the `MouseRec` for the mouse click on the button. Finally,

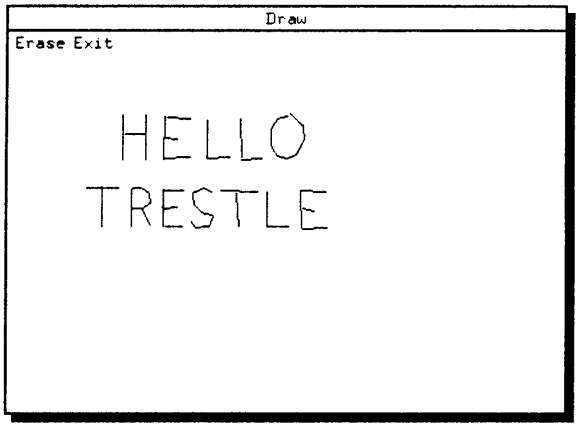

Figure 4: A simple drawing program that illustrates mouse tracking, buttons, and drawing PostScript-like paths.

`ButtonVBT.MenuBar(`v_1`, ... ,` v_n`)`

Return a split with children v_1, ..., v_n, *left-justified and separated by small horizontal gaps.*

One of the buttons of the program exits the program. It uses

`Trestle.Delete(v)`

Delete the installed window `v`.

The program also uses the `UNUSED` and `FATAL` pragmas of SRC Modula-3. These suppress the warnings that would otherwise be given for unused parameters and unhandled exceptions. A `FATAL` pragma is legal wherever a declaration is legal, and has the same scope that the declaration would have; an `UNUSED` pragma is legal immediately before a formal parameter declaration. Our earlier example programs could be improved by declaring `TrestleComm.Failure` to be fatal.

```
MODULE Draw EXPORTS Main;

IMPORT VBT, Trestle, Point, Rect, Path, ButtonVBT,
  PaintOp, Region, HVSplit, TextVBT, Axis, TrestleComm;

FROM VBT IMPORT ClickType;

TYPE DrawVBT = VBT.Leaf OBJECT
    path: Path.T;
    drawing := FALSE;
    p, q: Point.T;
  OVERRIDES
    repaint  := Repaint;
    reshape := Reshape;
```

```
    mouse := Mouse;
    position := Position
  END;
```

The `drawing` flag is set while the user is rubber banding the segment `(p, q)`.

```
PROCEDURE Repaint (v: DrawVBT; READONLY rgn: Region.T) =
  BEGIN
    VBT.PaintRegion(v, rgn, Op := PaintOp.Bg);
    VBT.Stroke(v, rgn.r, v.path, op := PaintOp.Fg);
  END Repaint;

PROCEDURE Reshape(v: DrawVBT;
    READONLY cd: VBT.ReshapeRec) =
  <*FATAL Path.Malformed*>
  BEGIN
    v.path := Path.Translate(v.path,
      Point.Sub(
        Rect.Middle(cd.new),
        Rect.Middle(cd.prev)));
    v.drawing := FALSE;
    Repaint(v, Region.Full)
  END Reshape;
```

The following procedure inverts each pixel on the line from `v.p` to `v.q`:

```
PROCEDURE XorPQ(v: DrawVBT) =
  BEGIN
    VBT.Line(v, Rect.Full, v.p, v.q, Op := PaintOp.Swap)
  END XorPQ;

PROCEDURE Mouse(v: DrawVBT; READONLY cd: VBT.MouseRec) =
  BEGIN
    IF cd.clickType = ClickType.FirstDown THEN
      v.drawing := TRUE;
      v.p := cd.cp.pt;
      v.q := v.p;
      XorPQ(v);
      VBT.SetCage(v, VBT.CageFromPosition(cd.cp))
    ELSIF v.drawing AND cd.clickType = ClickType.LastUp
    THEN
      Path.MoveTo(v.path, v.p);
      Path.LineTo(v.path, v.q);
      VBT.Line(v, Rect.Full, v.p, v.q);
      v.drawing := FALSE
    ELSIF v.drawing THEN (* Chord cancel *)
      XorPQ(v);
      v.drawing := FALSE
```

```
      END
    END Mouse;
```

The first call to `XorPQ(v)` may be a no-op since `v.p` is equal to `v.q` at that point and therefore the segment is empty; but the program as written will work even for wide lines where the segment from a point to itself might be a non-empty dot with a radius equal to the line width.

```
PROCEDURE Position(v: DrawVBT;
    READONLY cd: VBT.PositionRec) =
  BEGIN
      (* You fill in this part *)
  END Position;
```

The basic function of the position method is to implement "rubber-banding" of the line being drawn. To do this it should undraw the line to the previous cursor position and redraw the line to the new cursor position.

```
PROCEDURE DoErase(<*UNUSED*>b: ButtonVBT.T;
  <*UNUSED*>READONLY cd:VBT.MouseRec) =
  BEGIN
    Path.Reset(drawVBT.path);
    drawVBT.drawing := FALSE;
    Repaint(drawVBT, Region.Full)
  END DoErase;

PROCEDURE DoExit(<*UNUSED*>b: ButtonVBT.T;
  <*UNUSED*>READONLY cd: VBT.MouseRec) =
  BEGIN
    Trestle.Delete(main);
  END DoExit;

VAR
  drawVBT := NEW(DrawVBT, path := NEW(Path.T));
  menuBar := ButtonVBT.MenuBar(
    ButtonVBT.New(TextVBT.New("Erase"), DoErase),
    ButtonVBT.New(TextVBT.New("Exit"), DoExit));
  main := HVSplit.Cons(Axis.T.Ver, menuBar, drawVBT,
    adjustable := FALSE);
  <*FATAL TrestleComm.Failure*>
BEGIN
  Trestle.Install(main);
  Trestle.AwaitDelete(main)
END Draw.
```

Notice that the `mouse` method implements the *chord cancel* convention: an unexpected chord on the mouse will cancel the drawing operation.

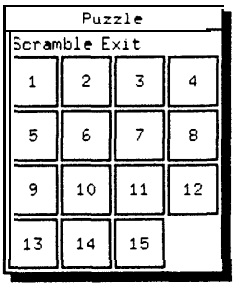

Figure 5: A computer version of the Fifteen Puzzle illustrates many operations on split windows, including *reparenting*.

7 The Fifteen Puzzle

Trestle has many kinds of splits. In addition to `HVSplits`, there are `ZSplits` (overlapping windows), `PackSplits` (in which the children are packed into rows like the words in a paragraph), and `TSplits` (in which the parent gives its screen to one child at a time). The `Split` interface provides operations that apply to `splits` in general, such as deleting, replacing, and enumerating children, To introduce the `Split` interface, we will program Sam Loyd's famous Fifteen Puzzle (see Figure 5).

The puzzle requires fifteen numbered cells to be sorted in order by sliding them around within a four by four frame. In our computerized version, clicking on a cell adjacent to the empty space will slide it into the space. There is also a button that scrambles the puzzle into a random solvable position.

```
MODULE Puzzle EXPORTS Main;

IMPORT Trestle, VBT, BorderedVBT, HVSplit, TextureVBT,
  Fmt, RigidVBT, PaintOp, Split, ButtonVBT, Axis,
  TextVBT, Random, TrestleComm;
```

Here is the type declaration for a `Cell`:

```
TYPE Cell = BorderedVBT.T OBJECT
  METHODS
    init(ch: VBT.T): Cell := Init
  OVERRIDES
    mouse := Mouse
  END;
```

`NEW(Cell).init(ch)` produces a `VBT` that looks like `ch`, but its shape is a rigid square and its mouse method moves the cell around in the puzzle. For the numbered cells, `ch` will be a black-bordered `TextVBT`; for the empty cell, `ch` will be a white `TextureVBT` child. (A `TextureVBT` is a `VBT` that displays a fixed texture.)

The cells are packed into rows using a horizontal `HVSplit`, and the rows are stacked on top of one another using a vertical `HVSplit`:

```
VAR puzzle := HVSplit.Cons(Axis.T.Ver,
  HVSplit.Cons(Axis.T.Hor,
    New(1), New(2), New(3), New(4)),
  HVSplit.Cons(Axis.T.Hor,
    New(5), New(6), New(7), New(8)),
  HVSplit.Cons(Axis.T.Hor,
    New(9), New(10), New(11), New(12)) ,
  HVSplit.Cons(Axis.T.Hor,
    New(13), New(14), New(15), New (16)));

  space: Cell;
   (* The cell representing the empty space *)

  cell: ARRAY [1..15] OF Cell;
   (* "cell[i]" is the cell numbered "i" *)

PROCEDURE New(n: INTEGER) : Cell =
  BEGIN
    IF n = 16 THEN
      space :=
        NEW(Cell).init(TextureVBT.New(PaintOp.Bg));
      RETURN space
    ELSE
      cell[n] :=
        NEW(Cell).init(
          BorderedVBT.New(TextVBT.New(Fmt.Int(n))));
      RETURN cell[n]
    END
  END New;
```

Here is the initialization procedure for creating a new cell:

```
PROCEDURE Init(c: Cell; ch: VBT.T): Cell =
  BEGIN
    EVAL BorderedVBT.T.init(c,
      RigidVBT.FromHV(ch, 10.0, 10.0),
      op := PaintOp.Bg);
    RETURN c
  END Init;
```

`BorderedVBT.T.init(c, ch, op := PaintOp.Bg)` initializes `c` to be a `BorderedVBT` with child `ch` and a thin border that will be painted with the background color. These thin white borders keep the cells from touching one another.

`RigidVBT.FromHV(ch, n, m)` returns a filter that looks and behaves like its child `ch`, except that its preferred size range is `n` by `m` millimeters. Here is how this

works: Every `VBT` has a shape method that determines its preferred size range. For example, the `shape` method for a `BorderedVBT` calls its child's `shape` method and then adds the border size. The `shape` method for a `RigidVBT` returns values supplied when the filter is created, ignoring its child's preferred size range.

We could have saved a filter level by overriding the `shape` method of `Cell` instead of using `RigidVBT`, but such parsimony would be out of place in a program like this one.

Since the cells move around as the user works on the puzzle, we need a procedure for finding a cell's current row and column:

```
PROCEDURE GetRowCol(c: Cell;
    VAR (*out*) row, col: INTEGER) =
  VAR
    parent: HVSplit.T := VBT.Parent(c);
    grandparent: HVSplit.T := VBT.Parent(parent);
  <*FATAL Split.NotAChild*>
  BEGIN
    col := Split.Index(parent, c);
    row := Split.Index(grandparent, parent)
  END GetRowCol;
```

The children of any split are ordered; `Split.Index(p, ch)` returns the number of children of split `p` that precede its child `ch`. Horizontal `HVSplits` are ordered left-to-right; vertical `HVSplits` are ordered top-to-bottom. Therefore, in `Puzzle`, the column of a cell is its index in the parent `HSplit`, and the row of a cell is its parent's index in the grandparent `VSplit`.

When the user clicks on a cell that is next to the space, the cell swaps itself with the space:

```
PROCEDURE Mouse(v: Cell; READONLY cd: VBT.MouseRec) =
  VAR vRow, vCol, spRow, spCol: INTEGER;
  BEGIN
    IF cd.clickType = VBT.ClickType.FirstDown THEN
      GetRowCol(v, vRow, vCol);
      GetRowCol(space, spRow, spCol);
      IF vRow = spRow AND ABS(vCol - spCol) = 1
         OR vCol = spCol AND ABS(vRow - spRow) = 1 THEN
        Swap(v, space)
      END
    END
  END Mouse;
```

Swapping is possible because Trestle's splits and filters allow children to be inserted and deleted dynamically. The procedure `Split.Replace(v, ch, newch)` will replace the child `ch` of `v` with the new child `newch`. The old child `ch` is placed in a

detached state, where it can be inserted into some other split if necessary. Two cells can be swapped using a dummy child and three replacements:

```
PROCEDURE Swap(v, w: VBT.T) =
  VAR temp := NEW(VBT.Leaf);
    <*FATAL Split.NotAChild*>
  BEGIN
    Split.Replace(VBT.Parent(v), v, temp);
    Split.Replace(VBT.Parent(w), w, v);
    Split.Replace(VBT.Parent(temp), temp, w)
  END Swap;
```

`Swap` could also have been implemented using the procedures `Split.Delete` and `HVSplit.Insert`, but it would have been messier.

One tricky bit of coding remains, which is the procedure that scrambles the puzzle:

```
PROCEDURE DoScramble(<*UNUSED*> v: ButtonVBT.T;
    <*UNUSED*> READONLY cd: VBT.MouseRec) =
  VAR j, parity: INTEGER;
  BEGIN
    parity := 0;
    FOR i := 1 TO 13 DO
      j := Random.Subrange(Random.Default, i, 15);
      (* Set j to a random element of [i..15] *)
      IF i # j THEN
        Swap(cell[i], cell[j]);
        INC(parity)
      END
    END;
    IF parity MOD 2 = 1 THEN
      Swap(cell[14], cell[15])
    END
  END DoScramble;
```

Exercise 5. Explain how `DoScramble` selects a random solvable position. (This exercise is more about permutations than about Trestle, but you might enjoy it anyway.)

You may be wondering at this point what the window looks like when the user clicks the `Scramble` button. If every call to `Split.Replace` updated the screen, there would be an unpleasant flurry of painting that would show many intermediate states as well as the final one. Trestle avoids this defect by implementing a policy of *lazy redisplay*. This means that Trestle allows the screen to become temporarily inconsistent, and fixes it only when the window configuration has stabilized. Here is the machinery that makes this work:

- Every `VBT` has a `redisplay` method. The call `v.redisplay()` is responsible for updating `v's` screen if it has become inconsistent.

- If an operation on a `VBT` makes its screen out-of-date, the operation *marks* the `VBT`. For example, `Split.Replace` marks the split as it swaps the new child for the old one.
- After every user event, Trestle calls the `redisplay` method of every marked `VBT`, simultaneously clearing its mark.

The `Spot` and `Track` programs didn't need to supply a `redisplay` method, because they never allowed the screen to become inconsistent.

The built-in splits work hard to redisplay economically. For example, if two children of an `HVSplit` with the same size are swapped, and then the split is redisplayed, then `HVSplit` reshapes only these two children; the other children won't know that anything happened. The `DoScramble` procedure might move a cell several times, but because of lazy redisplay the cell will only be displayed in its final position.

All that remains of the `Puzzle` program is to declare the procedure for exiting the puzzle, construct the main window, and install it:

```
PROCEDURE DoExit(<*UNUSED*> self: ButtonVBT.T;
    <*UNUSED*> READONLY cd: VBT.MouseRec) =
  BEGIN Trestle.Delete(main) END DoExit;

VAR menuBar :=
  ButtonVBT.MenuBar(
    ButtonVBT.New(TextVBT.New("Scramble"), DoScramble),
    ButtonVBT.New(TextVBT.New("Exit"), DoExit));

  main := HVSplit.Cons(Axis.T.Ver, menuBar, puzzle);

<*FATAL TrestleComm.Failure*>
BEGIN
  Trestle.Install(main);
  Trestle.AwaitDelete(main)
END Puzzle.
```

8 Cards

Our next example program illustrates `ZSplits`, which are parent windows that display overlapping child windows. The program has a pulldown menu that allows you to create subwindows that look like little colored cards, and drag them around with the mouse (see Figure 6).

The program uses Trestle's `HighlightVBT` to highlight the outline of a card as the user drags it. This filter uses `PaintOp.Swap` to complement the pixels of a rectangular outline on the screen. The highlight can be moved efficiently, without repainting the child, since the original pixel values can be restored with a second application of `PaintOp.Swap`. Solid rectangles can be highlighted by setting the highlight width to be very large.

Trestle automatically inserts a highlight filter above each top-level installed window so that any of its subwindows can use highlighting. For example, the feedback from the adjusting bars in the VSplit and Tiling Monster programs relied on the automatically-installed highlight filter.

The `Cards` program also illustrates how to build a pull-down menu and associate it with an anchor button.

Here is a list of the new procedures used by the `Cards` program:

`ZSplit.Lift(ch)`

Lift the child `ch` to the top of its `ZSplit` parent.

`ZSplit.InsertAt(z, ch, pt)`

Insert the child `ch` into the `ZSplit z` with its northwest corner at `pt`, giving the child its preferred shape.

`ZSplit.Move(z, ch, rect)`

Change the domain of the child `ch` of the `ZSplit z` to the rectangle `rect`.

`HighlightVBT.SetRect(v, rect)`

Change the highlighted rectangle of the first highlight filter above the `VBT v` to be `rect`. This can be used to set the highlight for the first time, move the highlight, or take down the highlight (if `rect = Rect.Empty`). A third argument sets the width of the highlight; it defaults to a thin line.

`MenuBtnVBT.TextItem(txt, p)`

Create a `ButtonVBT.T` with child `TextVBT.New(txt)` and action procedure `p`, suitable for including in a pop-up or pulldown menu. The item will be highlighted when the user rolls the mouse into it, and activated if he releases the mouse button over it.

`AnchorBtnVBT.New(ch, m)`

Create a button that looks like `ch` and that pops up the menu `m` in the first `ZSplit` ancestor of `ch` when the user clicks on it. The menu will be positioned so that its northwest corner coincides with the southwest comer of `ch`. Generally the menu is a vertical split of buttons, but in general it can be any `VBT`.

`Split.AddChild(v, ch)`

Insert `ch` as a new last child of `v`, which can be any kind of split.

And here is the listing of the program:

```
MODULE Cards EXPORTS Main;

IMPORT PaintOp, RigidVBT, TextureVBT, BorderedVBT, VBT,
```

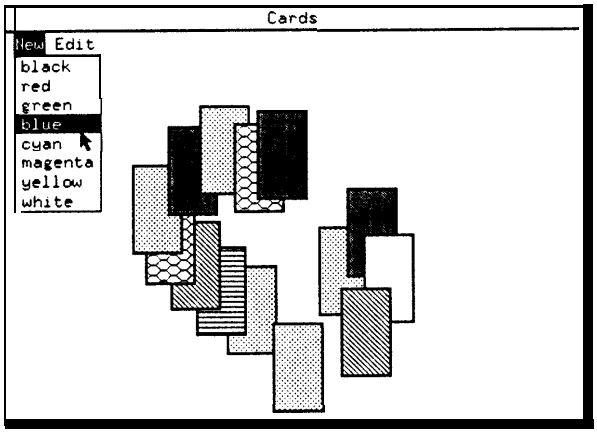

Figure 6: This program allows the user to create small colored cards and drag them into overlapping arrangements. A gray highlight rectangle follows a card as it is dragged. (Textures have been substituted for colors to produce a black-and-white figure.)

```
ZSplit, Rect, Point, Trestle, Filter, HighlightVBT,
ButtonVBT, HVSplit, AnchorBtnVBT, TextVBT, Axis, Text,
Split, MenuBtnVBT, TrestleComm, Random;
```

The cards are rigid bordered `TextureVBTs` with custom `mouse` and `position` methods to implement the dragging behavior:

```
TYPE Card = BorderedVBT.T OBJECT
  METHODS
    init(r, g, b: REAL): Card := Init
  OVERRIDES
    mouse := Mouse;
    position := Position
  END;

PROCEDURE Init(card: Card; r, g, b: REAL) : Card =
  VAR ch :=
    RigidVBT.FromHV(
      TextureVBT.New(
        PaintOp.FromRGB(r, g, b)),
      20.0, 32.3);
  BEGIN
    EVAL BorderedVBT.T.init(card, ch);
    RETURN card
  END Init;
```

The parent of the card windows is a global `ZSplit.T`, extended with additional data fields to represent the state associated with the dragging:

```
TYPE Parent = ZSplit.T OBJECT
    draggee: Card := NIL;
    rect  := Rect.Empty;
    pt:  Point.T
  END;

VAR zSplit :=
  NEW(Parent).init(TextureVBT.New(PaintOp.Bg));
```

Since `parent` does not declare an `init` method, it inherits the method from `ZSplit.ZSplit.init` takes a single argument, which is a `VBT` to use as a *background* child. This child is below all the other children, and by default has the same shape as the parent. It is not necessary to declare a separate `init` method for a `Parent`, since `draggee` and `rect` can be initialized to their default field values, and the initial value of `pt` doesn't matter. But calling `ZSplit.New` wouldn't work to initialize `zSplit`, since it would allocate a `ZSplit.T`, not a `Parent`.

The `mouse` method for the `Cards` program begins dragging on a down click, moves the child on the corresponding up click, and cancels on any chord:

```
PROCEDURE Mouse(ch: Card; READONLY cd: VBT.MouseRec) =
  VAR p: Parent := VBT.Parent(ch);
  BEGIN
    IF cd.clickType = VBT.ClickType.FirstDown THEN
      p.draggee := ch;
      p.rect  := VBT.Domain(ch);
      p.pt  := cd.cp.pt;
      ZSplit.Lift(ch);
      HighlightVBT.SetRect(p, p.rect);
      VBT.SetCage(ch, VBT.CageFromPosition(cd.cp, TRUE))
    ELSIF p.draggee = ch THEN
      IF cd.clickType = VBT.ClickType.LastUp THEN
        ZSplit.Move(ch,  p.rect)
      END;
      HighlightVBT.SetRect(p,  Rect.Empty);
      p.draggee := NIL;
      VBT.SetCage(ch,  VBT.EverywhereCage)
    END
  END Mouse;

PROCEDURE Position(ch: Card;
    READONLY cd: VBT.PositionRec) =
  VAR p: Parent  :=  VBT.Parent(ch);
  BEGIN
    IF p.draggee # ch THEN
```

```
      VBT.SetCage(ch, VBT.EverywhereCage)
    ELSE
      IF NOT cd.cp.offScreen AND
         Rect.Member(cd.cp.pt, VBT.Domain(p))
      THEN
        p.rect :=
          Rect.Add(p.rect, Point.Sub(cd.cp.pt, p.pt));
        p.pt := cd.cp.pt;
        HighlightVBT.SetRect(p, p.rect)
      END;
      VBT.SetCage(ch, VBT.CageFromPosition(cd.cp, TRUE))
    END
  END Position;
```

The procedures illustrate two subtle points about the delivery of mouse clicks and cursor positions. First, whenever Trestle delivers a mouse click of type `FirstDown` to a window, it designates this window as the *mouse focus.* All subsequent mouse button transitions will be delivered to the mouse focus, regardless of where they occur, up to and including the next transition of type `LastUp`. If it weren't for this rule the program wouldn't work: the final up transition could be delivered to the background or to another card, either of which would ignore it.

Second, `VBT.CageFromPosition` takes a boolean argument that controls whether to track the cursor outside the domain of the `VBT`. The `Draw` program defaulted this argument to `FALSE`, causing all positions outside the window to be treated alike, so that rubber banding froze when the cursor left the window. This would not work for the `Cards` program, since the card has to track the cursor outside its domain. It therefore sets the boolean argument to `TRUE`.

The ability to track outside your domain is useful, but not without its dangers. Suppose the user drags a card clear outside the top-level window and releases the button. Without special precautions, the card would move itself outside its parent's domain and become invisible to the user. To avoid this, the position procedure doesn't move the highlight rectangle if the position it receives is outside the parent's domain.

The position procedure tests for the possibility `cd.cp.offscreen` which indicates a position on a different screen from the window receiving the position. Since different screens might or might not have different coordinate systems, this test is necessary to prevent the possibility that the card would track when the user rolls the cursor around a different screen.

Next we consider the button action procedure that inserts a new colored card:

```
TYPE
  ClrRec = RECORD r, g, b: REAL; name: TEXT END;

CONST
  Clr = ARRAY OF ClrRec {
    ClrRec{0.0, 0.0, 0.0, "black"},
    ClrRec{1.0, 0.0, 0.0, "red"},
```

```
    ClrRec{0.0, 1.0, 0.0, "green"},
    ClrRec{0.0, 0.0, 1.0, "blue"},
    ClrRec{0.0, 1.0, 1.0, "cyan"},
    ClrRec{1.0, 0.0, 1.0, "magenta"},
    ClrRec{1.0, 1.0, 0.0, "yellow"},
    ClrRec{1.0, 1.0, 1.0, "white"}};

PROCEDURE DoNewChild(b: ButtonVBT.T;
    <*UNUSED*> READONLY cd: VBT.MouseRec) =
  VAR
    colorName: TEXT;
    card: Card;
    p: Point.T;
    dom: Rect.T;
  BEGIN
    colorName := TextVBT.Get(Filter.Child(b));
    FOR i := FIRST(Clr) TO LAST(Clr) DO
      IF Text.Equal(Clr[i].name, colorName) THEN
        card :=
          NEW(Card).init(Clr[i].r, Clr[i].g, Clr[i].b);
        EXIT
      END
    END;
    p.h := Random.Subrange(Random.Default, 10, 100);
    p.v := Random.Subrange(Random.Default, 10, 100);
    dom := VBT.Domain(zSplit);
    ZSplit.InsertAt(zSplit, card,
      Point.Add(Rect.NorthWest(dom), p))
  END DoNewChild;
```

The interface `Filter` provides routines that apply to any filter, just as `Split` provides routines that apply to any split. To determine what color of card to create, the `DoNewChild` procedure uses `Filter.Child(b)` to get the `TextVBT` child out of the button, and then `TextVBT.Get` to get the text out of the `TextVBT`.

A small random offset is added to the position of each newly inserted card, since it looks better to spread them out.

The program also has `Exit` and `Erase` buttons:

```
PROCEDURE DoExit(<*UNUSED*> v: ButtonVBT.T;
    <*UNUSED*> READONLY cd: VBT.MouseRec) =
  BEGIN Trestle.Delete(main) END DoExit;

PROCEDURE DoErase(<*UNUSED*> v: ButtonVBT.T;
    <*UNUSED*> READONLY cd: VBT.MouseRec) =
  VAR p, q, background: VBT.T;
  <*FATAL Split.NotAChild*>
  BEGIN
```

```
    p := Split.Succ(zSplit, NIL);
    background := Split.Pred(zSplit, NIL);
    WHILE p # background DO
      q := p;
      p := Split.Succ(zSplit, p);
      IF ISTYPE(q, Card) THEN
        Split.Delete(zSplit, q)
      END
    END
  END DoErase;
```

The `DoErase` procedure uses the `Succ` and `Pred` procedures to enumerate the children of the `Zsplit` and delete each one that is a card. These apply to any split. `Succ(v, NIL)` is the first child of the split `v`; `Pred(v, NIL)` is the last child of `v`, which in the case of a `ZSplit` is the background. Care must be taken not to delete the background or the pull-down menu, which could be in the `ZSplit` when `DoErase` is called.

The final section of the `Cards` program builds the pull-down menus and assembles the main `VBT`:

```
PROCEDURE Menu1(): HVSplit.T =
  VAR res: HVSplit.T;
  BEGIN
    res := HVSplit.New(Axis.T.Ver);
    FOR i := FIRST(Clr) TO LAST(Clr) DO
      Split.AddChild(res,
        MenuBtnVBT.TextItem(Clr[i].name, DoNewChild))
    END;
    RETURN res
  END Menu1;

VAR
  menu1 := BorderedVBT.New(Menu1());
  menu2 :=
    BorderedVBT.New(
      HVSplit.Cons(Axis.T.Ver,
        MenuBtnVBT.TextItem("erase", DoErase),
        MenuBtnVBT.TextItem("exit", DoExit)));
  menuBar :=
    ButtonVBT.MenuBar(
      AnchorBtnVBT.New(TextVBT.New("New"), menu1),
      AnchorBtnVBT.New(TextVBT.New("Edit"), menu2));

  main := HVSplit.Cons(Axis.T.Ver,
    menuBar, HighlightVBT.New(zSplit));

<*FATAL TrestleComm.Failure*>
BEGIN
```

```
    Trestle.Install(main);
    Trestle.AwaitDelete(main)
END Cards.
```

The call to `HighlightVBT.New` is not absolutely necessary, because of the free highlight filter provided by `Trestle.Install`. But the `Cards` program looks sharper with a highlight filter that does not extend over the menu bar, so the program inserts one that covers only the `Zsplit`. (Otherwise, if the user dragged a card so that it stuck out beyond the north boundary of the `Zsplit`, the highlight would show up in the menu bar during dragging, while the image of the card itself would be clipped.)

9 About locking level

When a user invokes a slow operation, like sorting a data base or linking a large program, he generally would prefer to continue to issue commands to the system while the slow operation is in progress. For example, he would at least like the option of issuing a `cancel` command. Good user interfaces cater to this desire, but there is a cost in program complexity, since now multiple threads can be operating on the `VBT` tree concurrently—the thread or threads executing the long-running operation, as well as the thread forked by the Trestle runtime that continues to deliver events to the user. This section describes Trestle's synchronization rules, which allow multiple threads to operate on the tree of `VBTs` concurrently.

If a client thread were adjusting the subwindow structure of a split window at the same time that Trestle was, say, locating a mouse click in the structure, then chaos could result. To protect against this, the Trestle system is a big critical section protected by the mutex `VBT.mu`.

To deliver an event, Trestle locks `VBT.mu` and calls the event method for the root window in the tree of splits. Since `VBT.mu` is locked, the event method can access or modify the split structure. In particular, it can locate subwindows and recursively activate their event methods. When the event method for the root window returns, Trestle unlocks `VBT.mu`.

Thus, operations on splits that insert, delete, or move children require that `VBT.mu` be locked. In all our example programs so far, such operations were invoked only from the `mouse` method; since the system automatically acquires `VBT.mu` before calling the `mouse` method, it was not necessary to explicitly lock `VBT.mu`. But if, for example, a program forked a thread that inserted a window in a `ZSplit`, it would be necessary for that thread to lock `VBT.mu` before inserting the window. We will see examples of explicit locking of `VBT.mu` later in the tutorial.

In addition to `VBT.mu`, every `VBT` includes a private mutex that serializes operations on the `VBT` itself. For programming most user interfaces you never need to think about these per- `VBT` mutexes, but if you implement your own split classes then you will have to pay attention to them.

To avoid deadlock, Trestle specifies an order on all its locks, called the *locking order*. A thread that acquires more than one lock at a time must acquire them in the locking order. The order is defined by these two rules:

- The global `mu` precedes every `VBT`.
- Every `VBT` precedes its parent.

For example, if `v` is a `VBT` and `p` is the parent of `v`, then

```
LOCK VBT.mu DO LOCK v DO LOCK p DO ... END END END
```

is legal, but any other order for acquiring the three locks would be illegal. Similarly, if `u` and `v` are `VBTs` and neither is an ancestor of the other, then

```
LOCK u DO LOCK v DO ... END END
```

is illegal, since there is no order defined for `VBTs` unless one is an ancestor of the other.

In the Trestle Reference Manual, every procedure specification indicates which locks must be held at procedure entry, and which must not be held. The notation for this is based on the concept of the *locking level* of a thread, or `LL` for short. The symbol `LL` stands for the set of locks that the thread has acquired. Furthermore, the expression `LL.sup` stands for the maximum of the locks in `LL`, in the locking order. For example, the specification

```
PROCEDURE Delete(v: Split.T; ch: VBT.T)
<* LL.sup = VBT.mu *>
```

means that in order to call `Delete`, a thread must hold `VBT.mu` but not hold any higher lock. Thus it is legal to call `Delete` from within a `mouse` method, since Trestle always calls a `mouse` method with `LL.sup = VBT.mu`. But it would be illegal to lock a `VBT` and then call `Delete`, since `VBT` locks are higher than `VBT.mu` in the locking order.

Similarly, every shared variable or data field is also commented with an indication of which locks must be held to write the data field. (The public Trestle interfaces generally do not reveal shared variables, but you will encounter such comments in the lower-level interfaces that reveal the representation of a `VBT`. For example, the specification

```
TYPE T =
  MUTEX OBJECT
    <* LL >= {VBT.mu, SELF} *>
    n: INTEGER
  END;
```

means that in order to write the `n` field of a variable `x` of type `T`, a thread must lock both `VBT.mu` and `x` itself. It follows that in order to read the field, it suffices to lock either `VBT.mu` or `x`. Locking-level comments on fields and shared variables are always of

the syntactic form `LL >= {m1, ... , mn}` where the `m's` are mutexes or the symbol `SELF`, which represents the object containing the fields. To write the field a thread must hold all the locks; to read the field it must hold at least one of the locks.

Similarly, each method of an object has a locking level. The locking level is specified in the same way as for a procedure, except that `SELF` designates the object containing the method.

For both fields and methods, a locking level pragma applies to all the fields or methods between it and the next pragma.

Notice that in a locking level pragma, the ordering symbols `>=, <=, <,` and `>` are overloaded to denote either set containment or lock order, depending on context. For example, `LL >= {mu, v}` indicates that the thread has both `mu` and `v` locked, while `LL.sup < v` indicates that all locks held by the thread precede `v` in the locking order.

A data field may also be commented `CONST`, meaning that it is readonly after initialization and therefore can be read with no locks at all.

There is one more special notation related to locking levels: a `VBT v` can hold a *share* of the global lock `VBT.mu`. The reason for this is that Trestle may lock `VBT.mu` and then reshape, repaint, or rescreen several `VBTs` concurrently. Thus you can't assume that an activation of your `reshape` method excludes the activation of another `VBT's reshape, repaint,` or `rescreen` method.

This locking level will be referred to as `v's` share of `VBT.mu`, and written `VBT.mu.v`. Holding `VBT.mu` is logically equivalent to holding `VBT.mu.v` for every `v`. Consequently, `VBT.mu.v < VBT.mu` in the locking order. Holding `VBT.mu.v` does not suffice to call a procedure that requires `VBT.mu` to be locked; on the other hand you cannot lock `VBT.mu` while holding `VBT.mu.v`, since this would deadlock.

All the procedures in the Trestle system restore the caller's locking level when they return.

10 Asynchronous painting

Our next example program is called `Plaid`. It draws moving plaid patterns on the display.

All of our examples so far have been synchronous: they never did anything except when Trestle prompted them by calling some `VBT` method. The `Plaid` program is asynchronous: it has a thread of control whose operations on the window are independent of the operations that Trestle orchestrates through method calls.

The `Plaid` program illustrates one reason for using asynchronous threads in a window application: to animate the screen. Another reason is to improve responsiveness by handling lengthy computations in the background. The point of the example is the locking protocol that must be obeyed by asynchronous threads, regardless of the reason for the asynchrony.

The painting thread of the `Plaid` program reads and writes the variables recording the animation state. The `reshape` method also writes these variables, to restart the animation whenever the window is reshaped. Therefore the painting thread

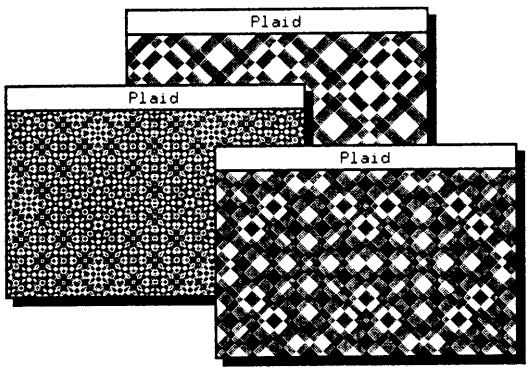

Figure 7: These windows are animated with varying plaid patterns. The program illustrates the locking protocol that allows asynchronous threads to operate on windows.

must synchronize with the `reshape` method. It is easiest to use `VBT.mu` for this synchronization, since `VBT.mu` is automatically locked in the `reshape` method. The painting thread must lock `VBT.mu` explicitly.

Here is the animation algorithm used by the `Plaid` program: visualize a ball rolling on a billiard table at constant speed, bouncing when it hits the rails. A rectangle continuously expands and contracts so that it is always centered in the table with one of its corners at the position of the ball. The rectangle is sampled every few milliseconds. During the interval between an odd sample and the following even sample, the border of the rectangle acts as an "inverting paintbrush" that complements the color of the region it sweeps over, while during the alternate intervals the border of the rectangle has no effect. Figure 7 shows some samples of the patterns that this algorithm produces.

The program is quite simple:

```
MODULE Plaid EXPORTS Main;

IMPORT VBT, Trestle, Thread, Point, Rect, Region,
  PaintOp, TrestleComm;

TYPE PlaidVBT =
  VBT.Leaf OBJECT
    <* LL >= {VBT.mu.SELF} *>
    p, deltaP: Point.T;
    prevRect: Rect.T;
    oddCycle: BOOLEAN;
    c: Thread.Condition
  OVERRIDES
    repaint := Repaint;
    reshape := Reshape
  END;
```

The points `p` and `deltaP` are the position and velocity of the ball, respectively. The rectangle `prevRect` is the position of the most recently sampled rectangle. The boolean `oddCycle` is `TRUE` if the previous cycle was an odd one.

When `Plaid's` domain is empty (as it is when the window is iconic, for example) the asynchronous thread blocks on `c` waiting for the domain to become non-empty. The condition is signaled when the `VBT's` domain becomes non-empty.

If Modula-3 threads are new to you, you should read Chapter 4 of *Systems Programming with Modula-3* [4].

The pragma `<* LL >= {VBT.mu.SELF} *>` means that the fields of the `PlaidVBT` are protected by `v's` share of the global mutex `VBT.mu`, as explained in the previous section.

The call `Advance(p, delta, dom)` advances `p` by `delta`, reflecting it to keep it inside `dom`, which must be non-empty:

```
PROCEDURE Advance(VAR p: Point.T;
  VAR delta: Point.T;
  READONLY dom: Rect.T) =
  BEGIN
    p := Point.Add(p, delta);
    LOOP
      IF p.h < dom.west THEN
        p.h := 2 * dom.west - p.h;
        delta.h := -delta.h
      ELSIF p.h > dom.east - 1 THEN
        p.h := 2 * (dom.east - 1) - p.h;
        delta.h := -delta.h
      ELSIF p.v < dom.north THEN
        p.v := 2 * dom.north - p.v;
        delta.v := -delta.v
      ELSIF p.v > dom.south - 1 THEN
        p.v := 2 * (dom.south - 1) - p.v;
        delta.v := -delta.v
      ELSE
        EXIT
      END
    END
  END Advance;
```

That is, as long as `p` is to the west of `dom`, reflect it in the west edge; as long as it is to the east, reflect it in the east edge, and so on for the north and south edges. Note that in the unusual case that `dom` is small compared to `delta`, the number of bounces required to simulate a single step may be large, but the loop will eventually terminate, since each bounce decreases the distance of `p` from the center of `dom`. The `-1's` in the program are necessary because rectangles are closed on the west and north edges but open on the east and south edges.

The call `PaintDiff(v, r1, r2)` inverts the region `r1-r2` in `v's` domain:

```
PROCEDURE PaintDiff(v: VBT.T; READONLY r1, r2: Rect.T) =
  VAR a: Rect.Partition;
  BEGIN
    Rect.Factor(r1, r2, a, 0, 0);
    a[2] := a[4];
    VBT.PolyTint(v, SUBARRAY(a, 0, 4), PaintOp.Swap)
  END PaintDiff;
```

`PaintDiff` relies on the very useful `Rect.Factor` routine, which computes the difference `r1 - r2` of two rectangles `r1` and `r2`. This difference will not in general be a rectangle, but it can always be expressed as a union of four disjoint rectangles; which `Factor` finds and stores in `a[0]`, `a[1]`, `a[3]`, and `a[4]`. (A `Rect.Partition` is simply an array of five rectangles.) `Factor` also sets `a[2]` to the intersection of `r1` and `r2`, so that the `a[i]'s` form a partition of `r1`. In this program we want `r1 - r2` as an array of rectangles, so we replace `a[2]` with `a[4]`, after which `a[0]` through `a[3]` are the desired partition. (The last two arguments control the order of the rectangles in the partition; `0,0` means that we don't care.)

Finally, the difference is painted using `VBT.PolyTint`, which takes a `VBT`, an array of rectangles and a painting operation, and applies the operation to each pixel of the `VBT's` screen that lies in any of the rectangles.

We could have used `Region` operations instead, but `Rect.Factor` is faster.

Here is the closure that is forked to create the asynchronous thread:

```
TYPE
  Closure = Thread.Closure OBJECT
    v: PlaidVBT
  OVERRIDES
    apply := Painter
  END;

PROCEDURE Painter(cl: Closure) : REFANY =
  VAR
    v := cl.v;
    dom, rect: Rect.T;
    mid: Point.T;
  BEGIN
    LOOP
      LOCK VBT.mu DO
        WHILE Rect.IsEmpty(VBT.Domain(v)) DO
          Thread.Wait(VBT.mu, v.c)
        END;
        dom := VBT.Domain(v);
        Advance(v.p, v.deltaP, dom);
        mid := Rect.Middle(dom);
```

```
            rect := Rect.FromSize(
              2 * ABS(v.p.h - mid.h),
              2 * ABS(v.p.v - mid.v));
            rect := Rect.Center(rect, mid);
            IF v.oddCycle THEN
              PaintDiff(v, rect, v.prevRect);
              PaintDiff(v, v.prevRect, rect)
            END;
            v.oddCycle := NOT v.oddCycle;
            v.prevRect := rect
          END;
          VBT.Sync(v)
        END
      END Painter;
```

`Rect.FromSize(h, v)` returns a rectangle whose width and height are `h` and `v`, respectively. `Rect.Center(rect, mid)` returns a rectangle congruent to `rect`, with middle at `mid`. The call `VBT.Sync(v)` flushes all pending paint commands for `v` to the screen; this call makes the animation smoother, at the cost of making it slower.

Here are the repaint and reshape methods:

```
PROCEDURE Repaint(v: PlaidVBT;
    <*UNUSED*> READONLY rgn: Region.T) =
  BEGIN Reset(v) END Repaint;

PROCEDURE Reshape(v: PlaidVBT;
    <*UNUSED*> READONLY cd: VBT.ReshapeRec) =
  BEGIN Reset(v) END Reshape;

PROCEDURE Reset(v: PlaidVBT) =
  VAR dom := VBT.Domain(v);
  BEGIN <* LL.sup = VBT.mu.v *>
    VBT.PaintTint(v, dom, PaintOp.Bg);
    v.p := Rect.Middle(dom);
    v.prevRect := Rect.Empty;
    v.oddCycle := FALSE;
    IF NOT Rect.IsEmpty(dom) THEN
      Thread.Signal(v.c)
    END
  END Reset;
```

The pragma `LL.sup = VBT.mu.v` means that the "locking level" of the thread is `VBT.mu.v`, that is, `v's` share of `VBT.mu`. The procedure `Reset` needs this locking level, since it writes the fields of `v` that are protected by `VBT.mu.v`. It also has this locking level, since it is called only from the `repaint` and `reshape` methods, which the system always calls with `LL.sup=VBT.mu.v`.

All that remains is to fork the thread and install the window:

```
VAR v := NEW(PlaidVBT,
           deltaP := Point.T{1, 1},
           c := NEW(Thread.Condition));
<*FATAL TrestleComm.Failure*>
BEGIN
  EVAL Thread.Fork(NEW(Closure, v := v));
  Trestle.Install(v);
  Trestle.AwaitDelete(v)
END Plaid.
```

11 Selections and event-time

From the user's point of view, a *selection* is a highlighted occurrence of text or other data that can be made in a window via some gesture, such as sweeping with the mouse. Selections are supported to make it easy for users to cut and paste text and other data between windows, even if the windows are installed by different programs. A particular selection is always in at most one window at a time, namely the *owner* of the selection. If a selection is in no window at all, its owner is `NIL`.

From the programmer's point of view, the selection owner is a `VBT -` valued variable shared between all applications. The procedure `VBT.Acquire` is used to acquire a selection. Whenever a `VBT` acquires a selection, the previous owner is notified, so that it can take down any highlighting or other feedback. Any `VBT` can own a selection, not just a top-level window.

The procedures `VBT.Read` and `VBT.Write` are used to read or write the value of the selection. Calls to `Read` and `Write` are implemented by locating the selection owner (which could be in the same address space as the caller to `Read` or `Write`, or in a different address space) and activating its `read` or `write` method, which is responsible for doing the work. Thus, if your `VBT` acquires ownership of a selection, it should have `read` and `write` methods that read and write the value of the selection. The selection values communicated by these methods can be of any type that can be pickled (see Section 3.6 of *Systems Programming with Modula-3* [4]); in particular, they can be of type `TEXT`.

When the `read` or `write` method is called, it is guaranteed that `LL.sup <= VBT.mu`. This is an unpleasantly weak guarantee: `VBT.mu` might or might not be locked. (Recall that the `read` or `write` method is being invoked by Trestle on behalf of some caller of `Read` or `Write`, and this caller might or might not be in the same address space as a `VBT` whose method is being called. This makes it difficult to provide a more specific locking level.) Since `VBT.mu` might or might not be locked, the selection value must be protected by some lock higher in the locking order, such as a `VBT` lock.

The `VBT` to which user keystrokes are directed is called the *keyboard focus.* Some window managers define the focus to be the window containing the cursor, other window managers move the focus in response to mouse clicks. Trestle applications work with either kind of window manager.

Trestle classifies the keyboard focus as a selection, since it is a global `VBT` - valued variable that can be acquired and released. If you want to receive keystrokes, you must acquire the focus. If this succeeds, you should provide some feedback to the user, for example by displaying a blinking caret. (Even if the window manager is identifying the top-level window containing the focus, you should still let the user know which subwindow contains the focus.) When you are notified that you have lost the focus, you should take down the feedback.

It is also possible to send any selection owner a *miscellaneous code,* which will be delivered by activating the `misc` method in the owner. For example, the way that Trestle notifies a window that it no longer owns a selection is by sending it a miscellaneous code of type `Lost.` Miscellaneous codes are also used for other purposes; for example, to notify windows that they have been deleted, and as a hint that they should take the focus or other selection.

In addition to the keyboard focus, Trestle has two built-in selections called the *source* selection and the *target* selection. Applications that use only one selection should use the source selection, which in Trestle-on-X is identified with the X primary selection. An application can create additional selections and identify them with existing X selections; for example with the Open Look clipboard.

There are many potential race conditions involving selections. For example, suppose that the user clicks in window `A,` expecting it to acquire the keyboard focus. But window `A` is slow—perhaps it is paging or blocked waiting for some slow server—and does not respond. So the user clicks in another window `B,` which acquires the keyboard focus, and types away. A few minutes later, window `A` comes to life and grabs the keyboard focus. Suddenly and unexpectedly the user's typing is redirected to `A` instead of `B.` Similar race conditions can occur with selections other than the keyboard focus—for example, you select a filename, then activate a `delete` command by clicking, then wonder how long you must wait before it is safe to make another selection.

Trestle uses the *event-time protocol* to deal with these race conditions. This means that Trestle keeps track of the *current event time,* which is the timestamp of the last keystroke, mouseclick, or miscellaneous code that has been delivered to any `VBT.` Attempts to read, write, or acquire a selection must be accompanied by a timestamp, and if this timestamp does not agree with the current event time, the attempt fails. This guarantees that only `VBTs` that are responding to the user's latest action can access the selections.

To illustrate selections and event-time, we will present the `TypeIn` interface. A `TypeIn.T` is a `VBT` into which the user can type a one-line text string. It is a simplified version of Trestle's `TypeInVBT.`

A `TypeIn.T` acquires the keyboard focus when the user clicks on it with the mouse. When the `VBT` has the keyboard focus, the user can append to the text by typing graphic characters, and can backspace by typing the key labelled `<x],` "backspace", or "delete". Associated with every `TypeIn.T` is an action procedure that is called whenever the user types a non-graphic character (such as `RETURN).`

Typing control-u or control-U in the `VBT` clears the text, middle-click appends the source selection, and control-click acquires the source selection.

```
INTERFACE TypeIn;

IMPORT VBT, TextVBT;

TYPE T <: Public;
  Public = TextVBT.T OBJECT
    METHODS
      init(txt: TEXT := ""; action: Proc := NIL): T
    END;

TYPE Proc = PROCEDURE(V: T; READONLY cd: VBT.KeyRec);

PROCEDURE New(txt: TEXT := ""; action: Proc := NIL): T;
New(…) is short for NEW(T).init(…).

END TypeIn.
```

Notice that the interface reveals that a `TypeIn.T` is a subtype of a `TextVBT.T`. Thus, for example, a client can use `TextVBT.Put` and `TextVBT.Get` to change the text in a `TypeIn`.

It often happens that to implement a subtype of a class `T` we need more information than ordinary clients of `T`. The extra information is often found in an interface with a name like `TClass`. In particular, the `TextVBT` interface doesn't reveal enough information to allow us to implement `TypeIn`, so the `TypeIn` implementation also imports the `TextVBTClass` interface:

```
INTERFACE TextVBTClass;

IMPORT TextVBT;

REVEAL TextVBT.T <: T;

TYPE T = TextVBT.Public OBJECT
  <* LL >= {SELF} *>
  text: TEXT
  END;
```

If you modify the `text` field, you must call `VBT.Mark`.

```
END TextVBTClass.
```

An importer of `TextVBTClass` can see that a `TextVBT.T` has all the attributes that were made public in the `TextVBT` interface (in fact the only such attribute is the `init` method), and that it also has a `text` field, which contains the text that the `VBT` currently displays. Notice that this field cannot be read or written without locking the `VBT`, and that modifying it destroys the display invariant and therefore incurs the

obligation to mark the `VBT`, which will cause Trestle to call its `redisplay` method in due time.

Here is the implementation of `TypeIn`:

```
MODULE TypeIn;

IMPORT TextVBT, Text, VBT, Latin1Key, KeyboardKey,
  Rect, TextVBTClass, VBTClass;

REVEAL T =
  Public BRANDED OBJECT
    action: Proc
  OVERRIDES
    init := Init;
    misc := Misc;
    read := Read;
    mouse := Mouse;
    key := Key;
  END;

PROCEDURE Init(
  v: T;
  txt: TEXT := "";
  action: Proc := NIL): T =
  BEGIN
    EVAL TextVBT.T.init(v, txt);
    v.action := action;
    RETURN v
  END Init;

PROCEDURE New(
  txt: TEXT := "";
  action: Proc := NIL): T =
  BEGIN
    RETURN NEW(T).init(txt, action)
  END New;

PROCEDURE TakeSelection(v: T; t: VBT.TimeStamp;
  s: VBT.Selection) =
  BEGIN
    TRY
      VBT.Acquire(v, s, t)
    EXCEPT
      VBT.Error => (* Skip *)
    END
  END TakeSelection;

PROCEDURE Misc(v: T; READONLY cd: VBT.MiscRec) =
  BEGIN
```

```
    TextVBT.T.misc(v, cd);
    IF cd.type = VBT.TakeSelection THEN
      TakeSelection (v, cd.time, cd.selection)
    END
  END Misc;
```

A misc code `cd` of type `TakeSelection` is a hint to acquire the selection `cd.selection`. Notice that the `misc` method methodically calls its supertype's `misc` method, and then performs the additional functionality that is particular to the `TypeIn`.

```
PROCEDURE Read(v: T; <*UNUSED*>s: VBT.Selection;
  tc: CARDINAL): VBT.Value RAISES {VBT.Error} =
  BEGIN
    IF tc = TYPECODE(TEXT) THEN
      LOCK v DO RETURN VBT.FromRef(v.text) END;
    ELSE
      RAISE VBT.Error(VBT.ErrorCode.WrongType)
    END
  END Read;
```

Trestle calls `v.read(s, tc)` when `v` owns the selection `s` and some other window requests that the value of the selection be manifested with type `tc`. The `TypeIn.T` is able to manifest selections only with type `TEXT`.

The call `VBT.FromRef(t)` produces a `VBT.Value val` such that `val.toRef()` will return `t`.

When the `read` method is called `LL.sup <= VBT.mu`, hence it is legal to lock `v` and return the value of its `text` field. If this field had been protected by `VBT.mu` instead of `v`, it would have been impossible to code the `read` method satisfactorily, since it is not known in the method whether `VBT.mu` is locked.

```
PROCEDURE Mouse(v: T; READONLY cd: VBT.MouseRec) =
  BEGIN
    TextVBT.T.mouse(v, cd);
    IF cd.clickType = VBT.ClickType.FirstDown THEN
      IF VBT.Modifier.Control IN cd.modifiers THEN
        TakeSelection(v, cd.time, VBT.Source)
      ELSIF cd.whatChanged = VBT.Modifier.MouseM THEN
        TRY
          TYPECASE
            VBT.Read(v, VBT.Source, cd.time).toRef()
          OF
            TEXT(t) =>
              LOCK v DO v.text := v.text & t END;
              VBT.Mark(v)
          ELSE (* skip *)
```

```
            END
          EXCEPT
            VBT.Error => (* skip *)
          END
        ELSE
          TakeSelection(v, cd.time, VBT.KBFocus)
        END
      END
    END Mouse;
```

There are three cases to the `mouse` method: on a control click, the `VBT` acquires the source selection; on an unmodified middle click, it reads the source selection and appends it to the text; on any other mouse click, it acquires the keyboard focus.

Reading and appending the source selection is a bit tricky. The call to `VBT.Read` returns an opaque object that represents the value of the source selection. This opaque object has a `toRef` method that manifests the value as a `REFANY`. If this turns out to be a `TEXT`, the program locks the `TextVBT`, appends the source selection to its text, and marks it for redisplay.

The more obvious alternative is to call

```
TextVBT.Put(v, TextVBT.Get(v) & t).
```

This avoids the need to import `TextVBTClass`. But if some other thread were concurrently operating on the `TextVBT`, it could produce erroneous results.

The fourth argument to `VBT.Read` specifies the typecode of the type in which the reader would like the selection to be manifested, but the code above defaults this argument to `TYPECODE(TEXT)`.

The call to `VBT.Read` raises `VBT.Error` in a variety of situations, such as if no `VBT` owns the source selection. In this case the program ignores the mouse click.

```
PROCEDURE Key (v: T; READONLY cd: VBT.KeyRec) =
  VAR key := cd.whatChanged; BEGIN
    IF NOT cd.wentDown
       OR Rect.IsEmpty(VBT.Domain(v)) THEN
      RETURN
    END;
    IF key >= 0 AND key <= Latin1Key.ydiaeresis THEN
      IF NOT VBT.Modifier.Control IN cd.modifiers THEN
        LOCK v DO
          v.text := v.text &
            Text.FromChar(VAL(cd.whatChanged, CHAR));
        END;
        VBT.Mark(v);
        RETURN
      ELSIF key = Latin1Key.U OR key = Latin1Key.u THEN
        TextVBT.Put(v, "");
```

```
        RETURN
      END
    ELSIF key = KeyboardKey.BackSpace
       OR key = KeyboardKey.Delete THEN
      LOCK V DO
        v.text := Text.Sub(v.text, 0,
          MAX(0, Text.Length(v.text)-1))
      END;
      VBT.Mark(v);
      RETURN
    END;
    IF v.action # NIL THEN v.action(v, cd) END
  END Key;

BEGIN END TypeIn.
```

The `key` method ignores up transitions of keys (the case where `cd.wentDown` is false). It also ignores keystrokes when its domain is empty, since the user could be surprised if his typing affected an invisible window (a window does not lose the keyboard focus just because it is reshaped to empty).

The range 0 to `Latin1Key.ydiaeresis` is the set of keys representing ISO-Latin-1 character codes.

Exercise 6. Modify `TypeIn` so that the text is highlighted when the `VBT` owns the keyboard focus. Hint: A misc code of type `VBT.Lost` is delivered to a `VBT` shortly after it loses a selection or acquires a selection that it already owns.

12 A simple filter

Many user interfaces can be constructed by programming your own leaf `VBTs` and connecting them together with the splits and filters that Trestle provides. But the time will come when you will want to venture into more exciting territory, and implement a split or filter class of your own. In this section we will look at the implementation of `Button`, a simplified version of `ButtonVBT`.

```
INTERFACE Button;

IMPORT VBT, Filter;

TYPE
  T <: Public;

  Public = Filter.T OBJECT
    METHODS <* LL.sup <= VBT.mu *>
      init(ch: VBT.T; action: Proc): T;
    END;
```

The call `v.init(ch, p)` initializes `v` with child `ch` and action procedure `p`, and returns `v`. The button looks like `ch`, but calls the action procedure when the user clicks on it. The shape of the button is rigid at its child's minimum shape.

```
Proc =
  PROCEDURE(self: T; READONLY cd: VBT.MouseRec);
  <* LL.sup = VBT.mu *>
```

There are several new features to explain in these declarations.

A button is a filter, as expected. However, instead of simply declaring `T <: Filter.T`, the type `Button.T` is declared as a subtype of `Public`, which contains an appropriate `init` method. From the point of view of the client the type `Public` might as well have been left anonymous, but that would be inconvenient in the implementation.

Notice that the locking levels are listed for the `init` method and action procedure type. The pragma `LL.sup = VBT.mu` preceding the action procedure type indicates that procedures of this type are called with `VBT.mu` locked, and no `VBTs` locked. The pragma `LL.sup <= VBT.mu` before the `init` method indicates that this method can be called with or without `VBT.mu` locked, provided that no `VBTs` are locked. (It is implicit that the `init` method can only be called on a newly allocated object, which is the reason that the implementation of the `init` method can get by without knowing whether `mu` is locked or not—no thread except the one calling `init` can be accessing the fields of the object.)

An alternative design would make the action a method of the object, and have clients override the method instead of passing the procedure to the `init` method. We chose against this design in order to allow buttons with different action procedures to share method suites.

```
PROCEDURE New(ch: VBT.T; action: Proc): T;
<* LL.sup = VBT.mu *>
```

`New(…)` is equivalent to `NEW(T).init(…)`.

```
END Button.
```

Here is the implementation of the interface:

```
MODULE Button;

IMPORT VBT, Filter, Rect, HighlightVBT,
  VBTClass, Axis;

FROM VBT IMPORT ClickType;

TYPE State = {Idle, Active, Armed};
```

A button is made non-idle by an initial down click and remains non-idle until the next button transition. The non-idle states are *armed* if the cursor is over the button and *active* otherwise. The button is highlighted when it is armed, and calls the action

procedure on an up transition while it is armed. Thus a button click can be cancelled either by chording or by rolling out of the button and releasing. If the user rolls out and then back in without releasing, then the button is re-armed.

```
REVEAL
  T = Public BRANDED OBJECT
    action: Proc;
    state := State.Idle;
    highlighter: HighlightVBT.T := NIL
  OVERRIDES
    mouse := Mouse;
    position := Position;
    shape := Shape;
    init := Init
  END;
```

When the button is highlighted, the `highlighter` field is set to the `HighlightVBT` above the button that is being used to highlight it.

The reason that the type `Public` was given a name instead of being left anonymous in the interface was to allow the type to be named in the revelation above. The keyword `BRANDED` makes this type unique; Modula-3 requires that the concrete type revealed for an opaque type be branded in this way.

```
PROCEDURE Init(v: T; ch: VBT.T; p: Proc) : T =
  BEGIN
    v.action := p;
    EVAL Filter.T.init(v, ch);
    RETURN v
  END Init;
```

Declaring `Button.T` as a subtype of `Filter.T` and including the call to `Filter.T.init` is all that is necessary to get the default behavior of a filter: for example, reshaping the parent reshapes the child, painting on the child paints on the parent, `Filter.Replace` can be used to replace the child of a button, and so on.

```
PROCEDURE Mouse(v: T; READONLY cd: VBT.MouseRec) =
  BEGIN
    Filter.T.mouse(v, cd);
    IF cd.clickType = ClickType.FirstDown THEN
      Arm(v);
      VBT.SetCage(v, VBT.InsideCage)
    ELSIF v.state = State.Armed THEN
      IF cd.clickType = ClickType.LastUp THEN
        v.action(v, cd);
      END;
      Disarm(v);
```

```
        v.state := State.Idle
      ELSE
        v. state := State.Idle
      END
    END Mouse;

  PROCEDURE Position (v: T; READONLY cd: VBT.PositionRec) =
    BEGIN
      Filter.T.position(v, cd);
      IF v.state # State.Idle THEN
        IF cd.cp.gone THEN
          IF v.state = State.Armed THEN Disarm(v) END;
          VBT.SetCage(v, VBT.GoneCage)
        ELSE
          IF v.state = State.Active THEN Arm(v) END;
          VBT.SetCage(v, VBT.InsideCage)
        END
      END
    END Position;
```

The call `Arm(v)` requires that `v` be idle or active; it leaves `v` armed.

```
  PROCEDURE Arm(v: T) =
    BEGIN
      v.state := State.Armed;
      v.highlighter := HighlightVBT.Find(v);
      HighlightVBT.Invert(v.highlighter,
        VBT.Domain(v), 99999)
    END Arm;
```

`HighlightVBT.Find(v)` returns the first ancestor of `v` that is a `HighlightVBT.`
The call `Disarm(v)` requires that `v` be armed; it leaves `v` active.

```
  PROCEDURE Disarm(v: T) =
    BEGIN
      v. state := State.Active;
      HighlightVBT.SetRect(v.highlighter, Rect.Empty, 0);
      v.highlighter := NIL
    END Disarm;

  PROCEDURE Shape(v: T; ax: Axis.T; n: CARDINAL)
    : VBT.SizeRange =
      VAR sh := Filter.T.shape(v, ax, n); BEGIN
        RETURN VBT.SizeRange{lo := sh.lo,
          hi := sh.lo+1, pref := sh.lo}
      END Shape;
```

The behavior of the shape method depends on whether `n` is zero. The call `v.shape(ax, 0)` retums the range of desirable sizes for `v` in the axis `ax`, assuming

nothing is known about its size in the other axis. If `n#0`, the call `sh := v.shape(ax, n)` returns the range of desirable sizes for `v` in the axis `ax` assuming that `v`'s size in the other axis is `n`. This allows the preferred height of a window to depend on its width, for example.

If `sh` is a `VBT.SizeRange`, it represents a range of desirable sizes as follows: `sh.lo` is the minimum desired size, `sh.hi-1` is the maximum desired size, and `sh.pref` is the preferred size.

It is a checked runtime error for a shape method to return an illegal size range. A `SizeRange sh` is legal if `0 <= sh.lo <= sh.pref < sh.hi`. A common error is to return an illegal size range with `sh.lo = sh.hi`.

The shape method for a button returns a `SizeRange` whose interval of desired sizes is a singleton containing only its child's minimum size. For example, if the child is a `TextVBT`, it will have a minimum size just large enough to hold the text, and a maximum size that is very large. The button will be rigid at the child's minimum size. The code relies on the fact that the default `shape` method `Filter.T.shape` recurses on the filter child.

```
PROCEDURE New(
    ch: VBT.T;
    action: Proc) : T =
  BEGIN
    RETURN Init(NEW(T), ch, action)
  END New;

BEGIN END Button.
```

Trestle provides many subtypes of buttons, in which the `mouse` and `position` methods are overridden to modify the behavior. For example, menu buttons highlight when the user moves the cursor into them; no click is required.

Notice that with the program we have presented, there is no way for these other button classes to see the `action` and `state` fields of the button, since these are revealed in the `Button` module rather than in the interface. But it would be confusing to ordinary clients to reveal these in the interface. Therefore, Trestle has an interface `BtnVBTClass` that reveals the fields of a button that are relevant to button class implementations, but not to ordinary clients; this is analogous to the `TextVBTClass` interface described previously.

Exercise 7. If `Disarm` called `HighlightVBT.Find` to locate the highlighter, then the `highlighter` field could be eliminated and buttons would require less memory. Explain why this would be a bad idea.

13 An event-time filter

The event-time operations are implemented via methods. Generally the default values of these methods are satisfactory, but occasionally it is appropriate to override them.

As an example, we will present a simple filter that keeps track of which `VBT` in the subtree below it has the keyboard focus.

A `FocusFilter.T` is a filter that keeps track of which of its descendants last had the keyboard focus. When the filter receives a hint that it should acquire the keyboard focus, it forwards the hint to that descendant. For example, if your application contains several type-in fields, you might want to put a `FocusFilter` around your top-level window, so that when the user directs the window manager to give the window the focus, then the focus will be grabbed by whichever text field had the focus last. (This is a special case of the more general functionality provided by Trestle's `ETAgent` filter.)

```
INTERFACE FocusFilter;

IMPORT Filter, VBT;

TYPE T <: Filter.T;

PROCEDURE New(ch: VBT.T): T;
New(…) is equivalent to NEW(T).init(…).

END FocusFilter.
```

The implementation is very simple:

```
MODULE FocusFilter;

IMPORT VBT, VBTClass, Filter;

REVEAL T = Filter.T BRANDED OBJECT
    <* LL >= {SELF} *>
    lastFocus: VBT.T := NIL
  OVERRIDES
    misc := Misc;
    acquire := Acquire
  END;

PROCEDURE Acquire(v: T; ch: VBT.T; w: VBT.T;
  s: VBT.Selection; ts: VBT.TimeStamp)
  RAISES {VBT.Error} = <* LL.sup = ch *>
  BEGIN
    Filter.T.acquire(v, ch, w, s, ts);
    IF s = VBT.KBFocus THEN
      LOCK v DO v.lastFocus := w END
    END
  END Acquire;
```

The method call `v.acquire(ch, w, s)` acquires the selection `s` for the window `w`, which is a descendant of `ch`, which in turn is a child of `v`. By default, this method simply recurses up the tree of `VBT`s. The acquire method of a `FocusFilter` performs this recursion by calling its supertype's method, `Filter.T.acquire`. If

this does not raise an exception, and if the relevant selection was the keyboard focus, then it records `w` in `v.lastFocus`. The `acquire` method is called with `LL.sup = ch`.

```
PROCEDURE Misc(v: T; READONLY cd: VBT.MiscRec) =
  VAR lastFocus: VBT.T;
  BEGIN
    IF cd.type = VBT.TakeSelection
       AND cd.selection = VBT.KBFocus
    THEN
      LOCK v DO lastFocus := v.lastFocus END;
      IF lastFocus # NIL THEN
        VBTClass.Misc(lastFocus, cd);
        RETURN
      END
    END;
    Filter.T.misc(v, cd)
  END Misc;
```

The method call `v.misc(cd)` processes the miscellaneous code `cd` for the `VBT v`. The default for a filter relays the code to the filter's child. The method for a `FocusFilter` agrees with the default except when the code is of type `TakeSelection` and the selection is the keyboard focus, in which case it forwards the code directly to its descendant that last had the focus. This is done by the procedure `VBTClass.Misc`, which invokes the descendant's method after establishing internal Trestle invariants. Both `v.misc` and `VBTClass.Misc` have locking level `LL.sup = VBT.mu`. Consequently the method must lock `v` in order to read `v.lastFocus`, and release the lock before calling `VBTClass.Misc`.

```
PROCEDURE New(ch: VBT.T): T =
  VAR res := NEW(T); BEGIN
    EVAL res.init(ch);
    RETURN res
  END New;

BEGIN END FocusFilter.
```

14 A proper split

A split that is not a filter is called a *proper split.* In this section we illustrate how to program a rather simple proper split, `HVFill`.

An `HVFill` is intended to provide filler when a `VBT` must occupy a space that is larger than its maximum size. For example, a tiling window manager could wrap an `HVFill`

around a column of windows, in order to paint the unoccupied space below the column with gray or with the background color.

An `HVFill` is designed to be used with exactly two children, the second of which is treated as a filler.

In general, an `HVFill.T` is like an `HVSplit`, but it gives its first child space up to its maximum, the remaining space to its second child, and no space at all to any other children.

The shape of an `HVFill` is the shape of its first child.

```
INTERFACE HVFill;

IMPORT VBT, Split, Axis, ProperSplit;

TYPE T <: Public;
  Public = ProperSplit.T OBJECT METHODS
    init(ax: Axis.T; ch: VBT.T; fill: VBT.T := NIL): T
  END;

PROCEDURE New(ax: Axis.T; ch: VBT.T;
  fill: VBT.T := NIL) : T;

New(…) is short for NEW(T).init(…).

END HVFill.
```

The call `v.init(ax, ch, fill)` initializes `v` to have axis `ax`, first child `ch`, and second child `fill`. If `fill` is defaulted to `NIL`, then `TextureVBT.New(txt := Pixmap.Gray)` is used.

Here is the implementation:

```
MODULE HVFill;

IMPORT VBT, Split, Axis, ProperSplit, VBTClass,
  TextureVBT, Pixmap, Rect, Region, Point;

<*FATAL Split.NotAChild*>

REVEAL
  T = Public BRANDED OBJECT
    ax: Axis.T
  OVERRIDES
    init := Init;
    shape := Shape;
    newShape := NewShape;
    insert := Insert;
    replace := Replace;
    move := Move;
    reshape : = Reshape;
    redisplay := Redisplay;
```

```
    axisOrder := AxisOrder;
  END;
```

An `HvFill` overrides several methods that we have not encountered before. The `newShape` method of a split must be called whenever the result returned by its `shape` method might change; this allows a split parent to cache the results of its child's `shape` method. The default `newShape` method signals a new shape for the parent whenever any of its children signals a new shape; this default worked fine for our previous splits, so there was no need to override the method before. The `insert, replace,` and `move` methods operate on the list of children of a proper split. The `redisplay` method is called to repaint the split after a series of `insert, replace,` and `move` operations have modified its list of children. Finally, the `axisOrder` method specifies whether the vertical size range depends on the horizontal size range, or vice versa.

```
PROCEDURE New(ax: Axis.T;
  ch: VBT.T; fill: VBT.T := NIL): T =
  BEGIN
    RETURN NEW(T).init(ax, ch, fill)
  END New;

PROCEDURE Init(v: T; ax: Axis.T;
  ch: VBT.T; fill: VBT.T := NIL) : T =
  BEGIN
    v.ax := ax;
    IF fill = NIL THEN
      fill := TextureVBT.New(txt := Pixmap.Gray)
    END;
    Split.AddChild(v, ch, fill);
    RETURN v
  END Init;
```

`Split.AddChild(v, ch`$_1$`, ..., ch`$_n$`)` inserts the children `ch`$_1$`,... ch`$_n$ into the split `v`.

```
PROCEDURE Shape(v: T; ax: Axis.T; n: CARDINAL)
   : VBT.SizeRange =
  VAR ch := Split.Succ(v, NIL); BEGIN
    IF ch = NIL THEN
      RETURN ProperSplit.T.shape(v, ax, n)
    ELSE
      RETURN VBTClass.GetShape(ch, ax, n)
    END
  END Shape;
```

The size range of an `HVFill.T` is simply the size range of its first child, assuming it has one. If not, the split defers to its supertype by calling `ProperSplit.T.shape`

(which happens to return a VBT's default size range). Notice that instead of invoking the child's shape method directly, the program uses VBTClass.GetShape.

```
PROCEDURE NewShape(v: T; ch: VBT.T) =
  BEGIN
    IF ch = Split.Succ(v, NIL) THEN
      VBT.NewShape(v)
    END
  END NewShape;
```

The call VBT.NewShape(v) indicates that the preferred size range of the v may have changed. Every VBT has an obligation to call NewShape when necessary; this rule allows split parents to cache the results of their children's shape methods.

The call to VBT.NewShape(v) is translated into a call to p.newShape(v), where p is v's parent. This allows each class of split to deal with changing child shapes in its own way.

For an HVFill, a new child shape is relayed upwards if the child is v's first child and ignored otherwise, since the preferred size range of the HVFill does not depend on the preferred size range of the filler.

The next three methods are responsible for inserting, moving and replacing children. Most of the work involves manipulating the link fields of the doubly linked list of children, which is performed by calling appropriate procedures in the ProperSplit interface.

Every VBT v contains a field v.upRef. This field is used to store information about v needed by v's parent. In particular, if v is a child of a proper split, then v.upRef is an object of type ProperSplit.Child, which contains the list pointers for the doubly linked list of children.

Subtypes of ProperSplit may store additional information in the up-ref. For example, the up-ref of a child of a ZSplit points to a ZSplit.Child, which is a subtype of Propersplit.Child that stores the portion of the child's domain that is visible. The up-ref of a child is initialized when it is inserted into a split.

In the case of an HVFill, no per-child information is required except the link fields, so the children's up-refs are simply of type ProperSplit.Child.

```
PROCEDURE Insert(v: T; pred: VBT.T; ch: VBT.T) =
  <*FATAL split.NotAChild*>
  VAR predCh: ProperSplit.Child;
  BEGIN
    IF pred = NIL THEN
      VBT.NewShape(v);
      predCh := NIL
    ELSE
      predCh := pred.upRef
    END;
    LOCK ch DO
```

```
      LOCK v DO
        ProperSplit.Insert(v, predCh, ch)
      END
    END
  END Insert;
```

The call `v.insert(pred, ch)` must insert the child `ch` into the split `v`, immediately following the existing child `pred` (or first, if `pred = NIL`). The method can assume `pred` is either a child of `v` or `NIL`, and also that `ch` is not a child of a split and is of the same screentype as `pred`; this is taken care of by the generic code in `Split` before it invokes the method.

In the case of an `HVFill`, inserting requires signaling a new shape if the inserted child is first. The remaining work can be done by calling the utility procedures `ProperSplit.PreInsert` and `ProperSplit.Insert`.

The call `ProperSplit.Insert(v, predCh, ch)` inserts `ch` into the list of children of `v`, immediately after the child whose up-ref is `predCh`, and marks `v` for redisplay.

The insert method has locking level `VBT.mu`, while `Propersplit.Insert(v, predCh, ch)` requires that both `v` and `ch` be locked. Changing the tree structure is quite delicate from the synchronization point of view.

```
PROCEDURE Move(v:  T; pred, ch:  VBT.T) =
  VAR predCh: ProperSplit.Child;
  BEGIN
    IF (pred = NIL) # (ch = Split.Succ(v, NIL)) THEN
      VBT.NewShape(v)
    END;
    IF pred # NIL THEN
      predCh := pred.upRef
    ELSE
      predCh := NIL
    END;
    LOCK v DO ProperSplit.Move(v, predCh, ch.upRef) END
  END Move;
```

The call `v.move(pred, ch)` is like `v.insert(pred, ch)`, except that `ch` is an existing child of `v` instead of a new child. In the case of an `HVFill`, the method signals a new shape if the child is moving into or out of first place, but not both. The program computes this exclusive-or using #. The actual manipulation of the linked list is performed by `ProperSplit.Move`, whose arguments are the up-refs of the child to be moved and the predecessor of its new position. `ProperSplit.Move` requires that the split be locked, and marks the split for redisplay.

```
PROCEDURE Replace(v: T; ch, new: VBT.T) RAISES {} =
  VAR predCh: ProperSplit.Child := ch.upRef; BEGIN
    IF ch = Split.Succ(v, NIL) THEN
```

```
      VBT.NewShape(v)
    END;
    IF new # NIL THEN
      LOCK new DO
        LOCK v DO
          ProperSplit.Insert(v, predCh, new)
        END
      END
    END;
    ProperSplit.Delete(v, predCh)
  END Replace;
```

The call `v.replace(ch, new)` replaces the existing child `ch` of `v` with the new child `new`, or deletes `ch` if new is `NIL`. In the case of an `HVFill`, the method simply inserts the `new` child (if it is not `NIL`) and then deletes the old one.

The `ProperSplit` operations called by `Insert`, `Move` and `Replace` never cause any children to be reshaped or repainted, instead they just mark the split for redisplay. The major remaining work is to implement the `redisplay` method, which must reshape the children appropriately. It turns out that this computation is almost the same as the computation required when the parent is reshaped, and therefore the `redisplay` and `reshape` methods share the same code.

```
PROCEDURE Redisplay(v: T) =
  BEGIN
    Redisplay2(v, v.domain)
  END Redisplay;

PROCEDURE Reshape(v: T; READONLY cd: VBT.ReshapeRec) =
  BEGIN
    Redisplay2(v, cd.saved)
  END Reshape;
```

The call `Redisplay2(v, rect)` reshapes the children of `v` to be consistent with their order and with the domain of `v`, assuming that the pixels in the rectangle `rect` are undamaged. Part of the procedure has been left for you to complete as an exercise.

```
PROCEDURE Redisplay2(v: T; READONLY saved: Rect.T) =
  VAR
    ch := Split.Succ(v, NIL);
    fill:= Split.Succ(v, ch);
  BEGIN
    IF ch = NIL THEN RETURN END;
    IF fill # NIL THEN
      VAR p := Split.Succ(v, fill); BEGIN
        WHILE p # NIL DO
          VBTClass.Reshape(p, Rect.Empty, Rect.Empty);
          p := Split.Succ(v, p)
```

```
        END
      END
    END;
    VAR
      tDom := Rect.Transpose(v.domain, v.ax);
      vSize := Rect.HorSize(tDom);
      vXSize := Rect.VerSize(tDom);
      chSize := MIN(vSize,
         VBTClass.GetShape(ch, v.ax, vXSize).hi-1);
      fillsize := vSize - chSize;
      chDom := Rect.Transpose(Rect.FromCorner(
        Rect.NorthWest(tDom), chSize, vXSize), v.ax);
      fillDom := Rect.Transpose(Rect.FromCorner(
        Point.MoveH(Rect.NorthWest(tDom), chSize),
        fillSize, vXSize), v.ax);
    BEGIN
      IF fill = NIL THEN
        ReshapeChild(ch, chDom, saved)
      ELSE
        (* You fill in this part *)
      END
    END
  END Redisplay;
```

`Redisplay2` has to do three things:

First, it must reshape any children beyond the second to be empty; this is done in the `WHILE` loop.

Second, it must compute the new domains of the first child `ch` and the filler `fill`. This uses several new routines from the geometry package:

> `Rect.Transpose(r, ax)`
>
> *If `ax` is `Axis.T.Hor`, return the rectangle `r`, else return `r` with its hozizontal and vertical extents exchanged.*
>
> `Rect.HorSize(r), Rect.VerSize(r)`
>
> *Return the width and height of the rectangle `r`, respectively*
>
> `Rect.FromCorner(pt, n, m)`
>
> *Return the rectangle whose northwest corner is `pt` and whose width and height are `n` and `m`.*
>
> `Point.MoveH(pt, n)`
>
> *Return `Point.T{pt.h+n, pt.v}`.*

The rectangle `tDom` is initialized to be `v's` domain, or `v's` domain transposed, so that in either case we can imagine that the axis is horizontal. This makes it easy to compute `v's` size and cross-size (`vSize` and `vXSize`); also the first child's size

`chSize` and the filler's size `fillSize`. The child's size is its maximum in the axis `ax`, unless this is larger than the parent's size. Notice that the cross-size is passed to the child's `shape` method, as explained in the section on the `Button` filter.

These dimensions allow computing `chDom` and `fillDom` (the domain of the child and filler), first assuming that the axis is horizontal, and then transposing if necessary to be correct for either axis.

Third and finally, the method must reshape `ch` and `fill` to the appropriate domains. This is done by the final `IF` statement.

If the filler is `NIL`, the child can simply be reshaped. In this case, if there is leftover space, it will be left unpainted.

Exercise 8. Fill in the missing part of the `Redisplay2` procedure. Hint: Use the procedure `ReshapeChild` below, and never allow the children's domains to overlap.

The call `ReshapeChild(v, new, saved)` reshapes `v` to have the domain new, allowing it to use the pixels in `saved` while it repaints. The call optimizes the case in which the domain is unchanged (treating it as a repaint), and optimizes away the repaint case in which everything has been preserved (treating it as a no-op):

```
PROCEDURE ReshapeChild(v: VBT.T;
  READONLY new, saved: Rect.T) =
  BEGIN
    IF v.domain # new THEN
      VBTClass.Reshape(v, new, saved)
    ELSIF NOT Rect.Subset(new, saved) THEN
      VBTClass.Repaint(
        v, Region.Difference(Region.FromRect(new),
        Region.FromRect(saved)))
    END
  END ReshapeChild;
```

Finally, we have the `axisOrder` method, which indicates the order in which the `VBT` would prefer to have its size ranges queried. In the case of an `HVFill`, the method defers to its first child:

```
PROCEDURE AxisOrder(v: T): Axis.T =
  VAR ch := Split.Succ(v, NIL); BEGIN
    IF ch = NIL THEN
      RETURN ProperSplit.T.axisOrder(v)
    ELSE
      RETURN ch.axisOrder()
    END
  END AxisOrder;

BEGIN END HVFill.
```

15 The painting method

In this section we present an implementation of a filter that overrides the method that defines the way a child paints. The example is a simplified version of Trestle's `HighlightVBT`.

A `Highlight.T` is a filter that highlights a rectangular outline over its child. The parent screen is obtained from the child screen by inverting a rectangular outline, which can be moved around under client control.

```
INTERFACE Highlight;

IMPORT VBT, Rect, Filter;

TYPE T <: Filter.T;

PROCEDURE New(ch: VBT.T): T; <* LL.sup <= VBT.mu *>
```

`New(ch)` is equivalent to `NEW(T).init(ch)`.

```
PROCEDURE Find(v: VBT.T): T; <* LL.sup = VBT.mu *>
```

Return the lowest ancestor of `v` that is a `Highlight.T`, or `NIL` if there isn't one. If `v` itself is a `Highlight.T`, then `Find(v)` returns `v`.

```
PROCEDURE Invert(v: VBT.T; READONLY r: Rect.T;
  inset: CARDINAL); <* LL.sup = VBT.mu *>
```

Set the highlight for `Find(v)` to be the outline of the given width inset into the given rectangle.

```
END Highlight.
```

The highlight can be cleared either by setting `r` to be `Rect.Empty` or by setting `inset` to be zero. If `inset` is set to be very large, then the entire rectangle `r` is highlighted. `Invert` is a no-op if `Find(v)` is `NIL`.

And here is the implementation:

```
MODULE Highlight;

IMPORT Batch, BatchUtil, BatchRep, FilterClass,
  PaintOp, Rect, Region, ScrnPixmap, VBT, VBTClass,
  Filter;

REVEAL T = Filter.T BRANDED OBJECT
    <* LL >= {VBT.mu, SELF.ch} *>
    rect := Rect.Empty;
    border: CARDINAL := 0
  OVERRIDES
    reshape := Reshape;
    capture := Capture;
```

```
    paintbatch := PaintBatch
  END;
```

No `init` method needs to be written, since the `init` method inherited from `Filter-VBT.T` is satisfactory.

```
PROCEDURE New(ch: VBT.T): T =
  BEGIN RETURN NEW(T).init(ch) END New;

PROCEDURE PaintBatch(v: T;
  <*UNUSED*> ch: VBT.T; ba: Batch.T) =
  BEGIN
    IF NOT Rect.Overlap(v.rect, ba.clip) AND
       NOT Rect.Overlap(v.rect, ba.scrollSource)
    THEN
      VBTClass.PaintBatch(v, ba)
    ELSE
      VAR
        rect: Rect.T;
        inset := Rect.Inset(v.rect, v.border);
      BEGIN
        BatchUtil.Tighten(ba);
        rect := Rect.Meet(v.rect,
          Rect.Join(ba.clip, ba.scrollSource));
        IF Rect.Subset(rect, inset) THEN
          VBTClass.PaintBatch(v, ba)
        ELSE
          PaintDiff(v, rect, inset);
          VBTClass.PaintBatch(v, ha);
          PaintDiff(v, rect, inset)
        END
      END
    END
  END PaintBatch;
```

To improve painting performance, Trestle combines painting commands into *batches,* and processes them a batch at a time.

The method call `v.paintBatch(ch, ba)` must paint the batch `ba` of painting commands on the child `ch` of `v`.

Every batch `ba` has a clipping rectangle `ba.clip,` which contains each pixel modified by any of the painting operations in the batch. It also has a rectangle `ba.scrollSource,` which contains each pixel used as source by any of the scrolling commands in the batch. If both of these rectangles are disjoint from the highlight rectangle `v.rect,` then the method can simply relay the batch to the parent by calling `VBTClass.Paintbatch(v, ba).` This is the fast path through the procedure.

Otherwise, there is more work to do. The method computes `inset,` which is the unhighlighted interior of `v.rect.` It then calls `BatchUtil.Tighten(ba),` which

makes `ba.clip` and `ba.scrollSource` as small as possible. (We could have called `Tighten` before testing for the fast path, but since `Tighten` is moderately expensive and `v.rect` is often empty, this would probably be less efficient.) It then computes `rect`, which contains all pixels that are in the highlight rectangle and are read or written by painting commands in `ba`. If these pixels all lie in `inset`, then none of them are highlighted, and again the method simply recurses on the parent.

Otherwise, the method computes the rectangle difference `rect-inset`, using `Rect.Factor`, as described in the description of `Plaid`. All of the pixels in the difference are currently inverted, and the difference contains all inverted pixels that might be read or written by painting commands in `ba`. The method therefore un-inverts the pixels, relays the batch to the parent, and re-inverts the pixels.

```
PROCEDURE PaintDiff(v: VBT.T; READONLY r1, r2: Rect.T) =
   (* Invert the region r1 - r2 in v's domain. *)
  VAR a: Rect.Partition;
  BEGIN
    IF Rect.Subset(r1, r2) THEN RETURN END;
    Rect.Factor(r1, r2, a, 0, 0);
    a[2] := a[4];
    VBT.PolyTint(v, SUBARRAY(a, 0, 4), PaintOp.Swap)
  END PaintDiff;
```

The `PaintDiff` procedure is the same as the one in the `Plaid` program.

```
PROCEDURE Reshape(v: T; READONLY cd: VBT.ReshapeRec) =
  VAR cdP := cd;
  BEGIN
    IF NOT Rect.Subset(
              Rect.Meet(v.rect, cd.saved), cd.new)
    THEN
      cdP.saved := Rect.Meet(cd.new, cd.saved)
    END;
    Filter.T.reshape(v, cdP)
  END Reshape;
```

The `reshape` method of a `Highlight.T` is like the `reshape` method of a standard filter, except that it must make sure that when the child repaints the new domain it does not use as source any inverted pixels that are in the old domain and not in the new domain. (Using inverted pixels that are in both the old and new domain is no problem, since the `paintbatch` method handles them correctly.)

```
PROCEDURE Capture(
    v: T;
    <*UNUSED*> ch: VBT.T;
    READONLY rect: Rect.T;
    VAR (*OUT*) br: Region.T)
```

```
    : ScrnPixmap.T =
  VAR res: ScrnPixmap.T;
    inset := Rect.Inset(v.rect, v.border);
    clip := Rect.Meet(v.rect, rect);
  BEGIN
    PaintDiff(v, clip, inset);
    res := VBT.Capture(v, rect, br);
    PaintDiff(v, clip, inset);
    RETURN res
  END Capture;
```

The call `v.capture(ch, rect, br)` reads the portion of the screen of `ch` contained in the rectangle `rect` and returns it as a pixmap. The region `br` is set to contain any pixels whose value could not be read, for example because of overlapping windows. For a `Highlight.T`, the method simply inverts the highlight, recurses on the parent, and restores the highlight. It would be possible to optimize this by considering whether `rect` overlaps `v.rect`, but we didn't bother.

```
PROCEDURE Find(v: VBT.T): T =
  BEGIN
    LOOP
      TYPECASE v OF
        T(v) => RETURN v
      ELSE v := v.parent
      END
    END
  END Find;
```

`Find` is a good illustration of the value of `TYPECASE`. Notice that it relies on the fact that `NIL` is a member of type `T`.

```
PROCEDURE Invert(w: VBT.T;
  READONLY r: Rect.T; bd: CARDINAL) =
  VAR v := Find(w); BEGIN
    IF v = NIL OR
       v.rect = r AND v.border = bd THEN
      RETURN
    ELSIF v.ch = NIL THEN
      v.rect := r;
      v.border := bd
    ELSE
      Invert2(v, r, bd)
    END
  END Invert;

PROCEDURE Invert2(v: T;
  READONLY r: Rect.T; bd: CARDINAL) =
```

```
  VAR old, new: Rect.Partition; BEGIN
    LOCK v.ch DO
      Rect.Factor(Rect.Meet(v.rect,  v.domain),
        Rect.Inset(v.rect, v.border), old, 1, 1);
      v.border := bd;
      IF bd = 0 THEN
        v.rect  := Rect.Empty
      ELSE
        v.rect := r
      END;
      Rect.Factor(Rect.Meet(v.rect,  v.domain),
        Rect.Inset(v.rect, v.border), new, 1, 1);
      FOR i  := 0 TO 4 DO
        IF (i # 2) THEN
           (* paint symmetric difference of edge "i" *)
          PaintDiff(v, new[i], old[i]);
          PaintDiff(v, old[i], new[i]);
        END
      END
    END
  END  Invert2;

BEGIN END Highlight.
```

The routine computes `old` and `new`, which are the old and new highlight regions as partitions. By passing `(1, 1)` to `Rect.Factor` in both cases we guarantee that the edges will be computed in the same order, that is, `old[i]` and `new[i]` will be corresponding edges of the old and new outline, for `i = 0, 1, 3,` and `4`.

Then, for each pair of corresponding edges, the routine inverts all pixels that are in the new but not the old, or in the old but not the new.

Exercise 9. Modify `Highlight` to allow a general region to be highlighted instead of just a rectangular outline.

16 Conclusion

We hope that you have enjoyed this tour of Trestle, and that you will go forth and write beautiful programs with beautiful user interfaces!

17 Solutions

Solution 1. The missing lines are:

```
VAR vh  := Axis.Other[hv]; mid  := (lo + hi) DIV 2; BEGIN
  RETURN HVSplit.Cons(hv,
```

```
    New(lo, mid, vh),
    HVBar.New(),
    New(mid, hi, vh))
END
```

Solution 2. The `reshape` method should initialize `delta` as follows:

```
delta := Point.Sub(Rect.Southwest(cd.new),
                   Rect.Southwest(cd.prev))
```

The rest of the method can be left unchanged.

The value of `Rect.Southwest(Rect.Empty)` is defined to be `Point.Origin`, so this solution works even if `cd.new` or `cd.prev` is empty.

Solution 3. The `rescreen` method should reset the spot to be centered at the origin and have the correct diameter:

```
PROCEDURE Rescreen(v: T; cd: VBT.RescreenRec) =
  BEGIN
    v.spot := Circle(VBT.MMToPixels(v, 4.0, Axis.Hor))
  END;
```

A rasterized circle centered at a lattice point looks better if its diameter is odd instead of even. If you care about such details, you can use:

```
PROCEDURE Rescreen(v: T; cd: VBT.RescreenRec) =
  BEGIN
    WITH r = VBT.MMToPixels(v, 4.0, Axis.Hor) DO
      v.spot := Circle(0.5 + FLOAT(ROUND(r - 0.5)))
    END
  END;
```

Solution 4. The missing code is:

```
IF NOT v.drawing THEN RETURN END;
VBT.SetCage(v, VBT.CageFromPosition(cd.cp));
IF NOT cd.cp.gone THEN
  XorPQ(v);
  v.q := cd.cp.pt;
  XorPQ(v)
END
```

Notice that if `v.drawing` is `FALSE`, the position procedure doesn't set a cage. In this case Trestle leaves the cage equal to `VBT.EverywhereCage`.

Solution 5. To describe how `DoScramble` works, we need a few simple facts about permutations.

- A transposition is a permutation that swaps two elements.

- Every permutation is a product of transpositions.
- The parity of a permutation is even if it is a product of an even number of transpositions, odd if it is a product of an odd number of transpositions. The parity doesn't depend on how the permutation is factored into transpositions.

We also define the parity of the empty cell to be even if the sum of its row index and column index is even; odd if the sum is odd. Initially the parity of the empty cell is even, since it is on the diagonal; initially the parity of the configuration is even, since it is the product of zero transpositions. Every move of the puzzle changes the parity of the configuration, and also the parity of the empty cell. Therefore, in every solvable position, the parity of the configuration is the same as the parity of the empty cell. We leave it to the reader to establish the converse, that any configuration whose parity is the same as the parity of the empty cell is solvable.

It follows that solvability is always preserved if the position of the space is preserved and the other pieces are rearranged by an even permutation. This is the strategy of `DoScramble:` it applies a randomly chosen even permutation to the numbered cells.

A random permutation is easily produced by selecting the first element randomly from among all the elements; the second from among the remaining elements, and so forth. Each element is swapped into its place immediately after it is selected. Thus the selected elements always form a prefix of the permutation, and each random selection is made from a suffix of decreasing size. In the last step there are only two elements in the suffix, and the random choice either transposes them or preserves them, with equal probabilities.

To produce a random even permutation, we keep track of the parity as we go, and modify the last step to transpose or not, as required to make the parity even.

Solution 6. Here is a solution that works if each `TypeIn` has a `HighlightVBT` immediately over it. We add an integer field, `kbCount,` to count the number of successful acquires of the keyboard focus minus the number of lost codes. In `TakeSelection,` change

```
VBT.Acquire(v, s, t)
```

to

```
VBT.Acquire(v, s, t);
IF S = VBT.KBFocus THEN
  IF v.kbCount = 0 THEN
    HighlightVBT.Invert(v, v.domain, 99999)
  END;
  INC(v.kbCount)
END
```

Next, in `Misc,` add

```
IF cd.type = VBT.Lost AND cd.selection = VBT.KBFocus
THEN
```

```
    DEC(v.kbCount);
    IF v.kbCount = 0 THEN
      HighlightVBT.SetRect(v, Rect.Empty)
    END
  END
```

Finally, we must add a `reshape` method to move the highlight if `kbCount` is non-zero, remembering to call `TextVBT.T.reshape`.

This solution doesn't work if more than one `TypeIn` share the same highlighter, since, when the focus switches between them, the `Lost` code will be delivered *after* the new owner has set the highlight. Thus the old owner will clear the highlight set by the new owner. An alternative would be to call `TextVBT.SetFont` to change the appearance of the `VBT`, avoiding `HighlightVBT` entirely.

Solution 7. Consider a `ZSplit` containing a dialog box containing a button whose effect is to delete the dialog box from the `ZSplit`. Suppose that the nearest highlighter above the button is above the whole `ZSplit`. When the user clicks down on the button, the button is highlighted; when the user releases over the armed button, the action procedure deletes the dialog from the `ZSplit`; then `Disarm` is called, which can no longer locate the highlighter.

Of course, you could modify the program to call `Disarm` before calling the action procedure, but the approach taken in the text is safest (although not the most economical).

Solution 8. The missing lines are:

```
IF NOT Rect.Overlap(chDom, fill.domain) THEN
  ReshapeChild(ch, chDom, saved);
  ReshapeChild(fill, fillDom, saved)
ELSIF NOT Rect.Overlap(fillDom, ch.domain) THEN
  ReshapeChild(fill, fillDom, saved);
  ReshapeChild(ch, chDom, saved)
ELSE
  ReshapeChild(fill, Rect.Empty, Rect.Empty);
  ReshapeChild(ch, chDom, saved);
  ReshapeChild(fill, fillDom, Rect.Empty)
END
```

The `IF` statement avoids overlapping the children while trying to give children their old domains if possible. If this is impossible for both children, it favors `ch` at the expense of `fill`, under the assumption that the filler is simpler to repaint.

Solution 9 (sketch). Replace calls to `PaintDiff` with calls to `VBT.PaintRegion`; simplify `Invert2` to invert the symmetric difference of the old and new regions (using `Region.SymmetricDifference`). Change various rectangle operations to the equivalent region operations; for example, test for the fast case in `PaintBatch` using `Region.OverlapRect` instead of `Rect.Overlap`.

For a more challenging exercise, change `Highlight` so that the highlighted region is specified by a pixmap, delta vector, and region, and is painted with `PaintOp.Copy` instead of with `PaintOp.Swap`. This would allow dragging an opaque icon over a window, for example. The difficulty is that the opaque painting destroys information, so that when the highlight is moved or removed, you will have to force the child to repaint the damaged region, for example by using `VBT.ForceRepaint`. If the child scrolls pixels that are inside the highlight region, you will have to walk the batch of painting commands (using the procedures in the `BatchUtil` and `PaintPrivate` interfaces) to locate the offending scrolling commands, build up a bad region, and pass it to `VBT.ForceRepaint`. A more cunning technique is to use `VBTClass.ForceRepaint` with `deliver := FALSE` to make the `VBT's` bad region contain the highlight region without causing it to immediately repaint.

18 Acknowledgments

We are grateful to Marc H. Brown for carefully proofreading this tutorial and making many helpful suggestions.

References

[1] Sam Harbison. *Programming with Modula-3.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[2] Mark Manasse and Greg Nelson. Performance analysis of a multiprocessor window system. Research Report 69, Digital Systems Research Center, 1992.

[3] Mark S. Manasse and Greg Nelson. Trestle reference manual. Research Report 68, Digital Systems Research Center, December 1991.

[4] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Index

asynchronous operation, 32

bitmap, 3
BorderedVBT, 3, 21
Button, 43
ButtonVBT, 15
ButtonVBT, menu anchor, 24
ButtonVBT, menu item, 24

cage, 14
cut buffer, 37

domain, 11

event time protocol, 38

Filter, 28
filter VBT, 3
FocusFilter, 48

Highlight, 57
HighlightVBT, 23
HVFill, 49
HVSplit, 3
HVsplit, adjusting bar, 3

initialization methods, 13

keyboard focus, 37

lazy redisplay, 22
LL (locking level), 31, 36
locking level, 36
locking order, 31

m3makefile, 2
marking a VBT, 22
mouse focus window, 27
mouse method, 10, 12

out-of-domain tracking, 27
overlapping window, 23

PaintDiff, 35
painting, 7
painting operation, 8
painting procedure, 9
PaintOp.Bg, 8
PaintOp.BgFg, 8
PaintOp.Copy, 9
PaintOp.Fg, 8
Paintop.Swap, 8
PaintOp.TransparentFg, 9
path, 15
pixmap, 3
point, 5
position method, 14
proper split, 49

race conditions in the user interface, 38
read method, 37
reading a selection, 37
rectangle, 5
redisplay method, 22
region, 5,6
repaint method, 10, 11
rescreen method, 12
reshape method, 10, 11, 33
resource, screen-dependent, 4
resource, screen-independent, 4
RigidVBT, 21

screen-dependent resource, 4
screen-independent resource, 4
selection, 37
shape method, 21
Split, 19
split VBT, 3
Split.Index, 21
Split.Replace, 22
synchronization rules, 30

texture, 3
TextureVBT, 19

TextVBT, 2
Tiling Monster, 4
tint (painting operation), 8
tint pair, 8
tracking the cursor, 13
TrackVBT, 13
TypeIn, 38

VBT, 1, 10
VBT.mu, 30
VBT.PaintRegion, 9
VBT.PaintTint, 9
VBT.PolyTint, 35
VBT.Scroll, 9

write method, 37
writing a selection, 37

ZSplit, 23