68

Trestle Reference Manual

Mark S. Manasse and Greg Nelson

December 1991

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in l984 – their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Trestle Reference Manual

Mark S. Manasse and Greg Nelson

December, 1991

Authors' abstract

This is a reference manual for Trestle, a Modula-3 toolkit for the X window system. Trestle is 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.

Trestle is included in SRC Modula-3 version 2.0, which is available via public ftp.

Trestle includes a fairly standard set of interactors, including menus, buttons, "container" classes that provide overlapping or tiled subwindows, and "leaf" windows that display text or other data. This reference manual also specifies the interfaces that allow you to create your own window classes. Knowledge of X is not required.

A Trestle window is an object whose behavior is determined by its methods. For example, a window's response to a mouse click is determined by calling its mouse method. This is fast becoming the standard architecture for toolkits, but Trestle carries it further than most. For example, you can change the way a Trestle window paints by overriding its paint method; this is useful for sophisticated effects like groupware.

Trestle provides a novel strategy for writing applications that are independent of the type of display screen they are running on. For example, it is easy to write a Trestle application that can be moved back and forth between a color display and a monochrome display where the application will look good on both.

Contents

1 Introduction **1**

2 The VBT interface **7**

2.1 The public methods . . . 7
2.2 Screens and domains . . . 8
2.3 Locking level . . . 8
2.4 ScreenTypes . . . 9
2.5 Splits and leaves . . . 10
2.6 Timestamps, modifiers, mouse buttons, and cursor positions . . . 10
2.7 The mouse method . . . 12
2.8 The mouse focus rule . . . 12
2.9 The position method . . . 13
2.10 Tracking the cursor by setting cages . . . 14
2.11 The key method . . . 16
2.12 The redisplay method . . . 17
2.13 The reshape method . . . 18
2.14 The rescreen method . . . 19
2.15 The repaint method . . . 19
2.16 About painting in general . . . 20
2.17 Scrolling (copying one part of the screen to another) . . . 21
2.18 Painting textures . . . 22
2.19 Filling and stroking paths . . . 24
2.20 Painting pixmaps . . . 25
2.21 Painting text . . . 26
2.22 Synchronization of painting requests . . . 29
2.23 Screen capture . . . 30
2.24 Controlling the cursor shape . . . 30
2.25 Selections . . . 31
2.26 Acquiring and releasing selection ownership . . . 32
2.27 The miscellaneous method . . . 32
2.28 Sending miscellaneous codes . . . 34
2.29 Circumventing event-time . . . 34
2.30 Communicating selection values . . . 34
2.31 The read and write methods . . . 36
2.32 Controlling the shape of a VBT . . . 37
2.33 Putting properties on a VBT . . . 37
2.34 Discarding a VBT . . . 38

3 The Trestle interface **39**
3.1 Window placement . . . 41
3.2 Enumerating and positioning screens . . . 42
3.3 Reading pixels from a screen . . . 43
3.4 Checking on recent input activity . . . 43
3.5 Connecting to a window system . . . 44

4 Splits **45**
4.1 The Split interface . . . 45
4.2 The ZSplit interface . . . 47
4.2.1 Inserting children . . . 48
4.2.2 Moving, lifting, and lowering children . . . 49
4.2.3 Mapping and unmapping children . . . 50
4.2.4 Getting domains . . . 50
4.2.5 Moving children when the parent is reshaped . . . 51
4.3 The HVSplit interface . . . 53
4.3.1 Inserting children . . . 55
4.3.2 Adjusting the division of space . . . 56
4.4 The PackSplit interface . . . 57
4.5 The TSplit interface . . . 59

5 Filters **60**
5.1 The Filter interface . . . 60
5.2 The BorderedVBT interface . . . 60
5.3 The RigidVBT interface . . . 62
5.4 The HighlightVBT interface . . . 63
5.5 The TranslateVBT interface . . . 64
5.6 Buttons . . . 65
5.7 Quick buttons . . . 66
5.8 Menu Buttons . . . 67
5.9 Anchor Buttons . . . 68

6 Some useful Leaf VBTs **71**
6.1 The TextVBT interface . . . 71
6.2 The TextureVBT interface . . . 72
6.3 The HVBar interface . . . 73

7 Resources **75**
7.1 The PaintOp interface . . . 75
7.2 The Cursor interface . . . 79
7.3 The Pixmap interface . . . 80
7.4 The Font interface . . . 81
7.5 The Palette interface . . . 82
7.6 The ScreenType interface . . . 84

7.7 Screen-dependent painting operations . . . 85
7.7.1 Obtaining handles from the oracle . . . 85
7.7.2 The handle object . . . 87
7.8 Screen-dependent cursors . . . 88
7.8.1 Obtaining handles from the oracle . . . 88
7.8.2 The handle object . . . 89
7.9 Screen-dependent pixmaps . . . 90
7.9.1 Obtaining handles from the oracle . . . 90
7.9.2 The handle object . . . 91
7.9.3 The raw representation . . . 91
7.10 Screen-dependent fonts . . . 94
7.10.1 Obtaining handles from the oracle . . . 94
7.10.2 Font attributes . . . 95
7.10.3 Registering fonts . . . 97
7.10.4 The handle object . . . 97
7.10.5 The raw representation . . . 98
7.11 Color maps . . . 100
7.11.1 Obtaining handles from the oracle . . . 100
7.11.2 The handle object . . . 101

8 Geometry interfaces 104
8.1 The Axis Interface . . . 104
8.2 The Point interface . . . 104
8.3 The Interval interface . . . 105
8.4 The Rect interface . . . 107
8.5 The Region interface . . . 110
8.6 The Path interface . . . 113
8.7 The Trapezoid interface . . . 116

9 Implementing your own splits 117
9.1 The VBTClass interface . . . 117
9.1.1 Specifications of the split methods . . . 122
9.1.2 Specifications of the up methods . . . 124
9.1.3 Getting and setting the state of a VBT . . . 125
9.1.4 Procedures for activating the down methods of a VBT . . . 126
9.1.5 Procedures for activating the up methods of a VBT . . . 128
9.2 The FilterClass interface . . . 129
9.3 The ProperSplit interface . . . 130

10 Implementing your own painting procedures 132
10.1 The Batch interface . . . 132
10.2 The BatchUtil interface . . . 132
10.3 The PaintPrivate interface . . . 134

11 Miscellaneous interfaces **139**
11.1 The VBTTuning interface 139
11.2 The TrestleComm interface 139

12 History and Acknowledgments **140**

References **141**

Index **143**

1 Introduction

This report is a programmer's reference manual for Trestle, a Modula-3 window system toolkit.

Trestle has been implemented over two underlying window systems: X [5] and the native Firefly window system developed at SRC [6]. Other implementations are possible, but at present the only widely available implementation is Trestle-on-X.

To use Trestle-on-X, you need a Modula-3 compiler and an X server for your system. The Trestle code is an application library layered on top of Xlib, the standard X client library. Trestle applications obey X's ICCCM protocol for cooperating with the window manager and other applications, so you can use your favorite window manager and mix Trestle applications freely with other X applications.

The reference manual is self-contained but non-tutorial; you would do well to read the *Trestle Tutorial* first (Chapter 7 of [4]). We assume you are familiar with Modula-3 [1, 2, 4].

The Trestle abstraction. A `Trestle.T` is a connection to a window system. The window system is assumed to have a keyboard, a pointing device, and one or more display screens. For example, in Trestle-on-X, a `Trestle.T` is implemented by a connection to an X server.

Each screen is a raster display, whose image is stored in a frame buffer containing a rectangular array of pixels. Changing the contents of a frame buffer is called *painting*, since it changes the image displayed to the user. The different screens can be of different types (e.g., color or black and white).

Trestle imposes an `hv`-coordinate system on each display screen, in which the `h` coordinate increases from left to right and the `v` coordinate increases from top to bottom. The `Trestle` interface allows you to determine the number of screens and also their types, dimensions, and the positions of their coordinate origins.

We will call the pointing device the mouse, although it might be a stylus or other instrument. The mouse generally has one or more buttons that the user can click down and up.

The system displays a *cursor*, a small arrow or other image that points at some pixel of some screen. By moving the mouse the user can move the cursor around the screen or from one screen to another. Applications can change the shape of the cursor to convey information to the user.

The complete `Trestle` interface is described in Section 3.

The VBT abstraction. The key abstraction in Trestle is the "virtual bitmap terminal" or `VBT`. A `VBT` represents a share of the keyboard, mouse, and displays. `VBT`s are comparable to the windows, widgets, and viewers of other systems.

An application is generally organized as a tree of `VBT`s, with the root `VBT` representing the top-level application window. The internal nodes are called *split* `VBT`s or *parent* `VBT`s: they divide their screens between one or more child `VBT`s according to some layout depending on the class of split. At the leaves of the tree are `VBT`s that contain no subwindows.

A typical application consists of a number of leaf VBTs whose behavior is specific to that application, together with some more leaf VBTs that provide buttons, scrollbars, and other "interactors", all held together by a tree of splits that define the geometric layout. A split with only one child is called a *filter*. For example, a BorderedVBT is a filter that adds a border around the child's screen. A split that can have more than one child is called a *proper split*. For example, an HVSplit is a split in which the children are laid out horizontally or vertically.

Sections 4, 5, and 6 describe Trestle's built-in proper splits, filters, and leaves.

To obtain a share of a Trestle display, an application creates a VBT and "installs" it with the procedure Trestle.Install, which allocates some portion of some display to the VBT, within which the application can paint. The VBT is said to be *installed* and is called a *top-level window*. The size and position of top-level windows depends on the arguments to Trestle.Install and on the whim of the window manager.

A VBT imposes an hv-coordinate system on its screen. A top-level VBT's coordinate system need not be the same as the coordinate system of the screen on which it is installed. The translation between the two coordinate systems can be determined through the Trestle.ScreenOf procedure.

In a split VBT, the translation between the parent and child coordinate systems depends on the class of the split. Trestle provides one filter (TranslateVBT) whose sole purpose is to position the child coordinate system origin at the northwest corner of the child's domain, since this is convenient for some applications. All the other built-in splits make the child coordinate system agree with the parent coordinate system, since this is usually the most convenient.

Information flows through a VBT in two directions. Painting commands travel from the leaves of the tree towards the root. Events like mouse clicks and cursor positions travel from the root towards the leaves. A VBT is an object with methods for handling events; to deliver an event to a VBT, the system invokes the appropriate method. The VBT interface in Section 2 specifies the event-handling methods and the painting procedures.

The screen of a VBT is *forgetful*; that is, its contents can be lost at any time, at which point the system activates its repaint method, which is expected to repaint what has been lost. Similarly, the height, width, and coordinate origin of a VBT's screen can change at any time, in which case the system activates its reshape method. Finally, the type of the pixels in a VBT's screen can change (e.g., from color to monochrome), in which case the system activates its rescreen method. These events reflect the fact that the user of the window system can expose portions of a top-level window, reshape top-level windows, and move top-level windows from one display to another.

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. 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. The selection values communicated by Read and Write 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.

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.

The event-time protocol. 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 in a call to a server that is being debugged—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 file name, 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.

When Trestle activates a window's method to deliver it an event, it generally waits for the method to return before it delivers any events to any other windows. This gives the window a fair chance to use the time stamp in the event to access the selections. However, if the method takes an unreasonably long time—more than a few seconds—Trestle may give up on the window and start delivering events to other windows anyway.

As a consequence, if you must do a long-running computation in response to a user event, then you should fork the computation in a separate thread and return from the method promptly, to avoid delaying the user, who may want to click in another window. You should also do any operations that require accessing the selections from the main thread before the method returns, since an event-time operation in the forked thread will fail if the user has continued typing or clicking during the forked computation.

The geometry interfaces. The interfaces `Axis`, `Point`, `Rect`, `Region`, `Trapezoid`, and `Path` are explained in Section 8. In brief, `Axis.T.Hor` and `Axis.T.Ver` name the horizontal and vertical coordinate axes; a `Point.T` (or simply a *point*) is a pair of integers representing a point in the plane; a `Rect.T` is a rectangle of points whose sides are parallel to the coordinate axes; a `Region.T` is an arbitrary set of points represented as a sorted array of rectangles; a `Trapezoid.T` is a set of points bounded by two horizontal lines and two lines with arbitrary slopes; and a `Path.T` is a path in the plane represented by a sequence of straight and curved segments.

Resources. A *pixmap* is a rectangular array of pixels. A *bitmap* is a pixmap in which the pixels are one bit deep. For example, a large pixmap could represent a photographic image; a small bitmap could represent a cursor shape. Trestle also uses a pixmap to represent the infinite texture that results from tiling the plane with translations of the pixmap. Thus whether a pixmap represents an infinite texture or a bounded image depends only on the context in which it is used.

A *font* is a typeface suitable for painting text.

A *painting operation* is an operation code for changing the values of pixels in the frame buffer of a display screen.

Pixmaps, cursor shapes, fonts, and painting operations are collectively called *resources*. Resources come in both *screen-independent* and *screen-dependent* forms. A screen-independent resource varies with the screentype to produce a similar effect on all types of screens. For example, two important screen-independent painting operations are `PaintOp.Fg` and `PaintOp.Bg`, which set pixels to a screen's foreground and background colors. In contrast, a screen-dependent resource is useful only on a particular screentype. If it is used on a `VBT` with the wrong type of screen, the system won't crash, but the effect will be non-deterministic—a screen-dependent painting operation that blackens a pixel on a black-and-white screen might set a 24-bit pixel to chartreuse on a true-color screen.

Screen-independent resources are convenient, but screen-dependent resources are sometimes necessary for exploiting the capabilities of specific display hardware.

The screen-independent resource types are called `Pixmap.T`, `Cursor.T`, `Font.T`, and `PaintOp.T`. The interfaces where these types are defined also provide procedures for generating useful resources. For example, `PaintOp.FromRGB` will produce a screen-independent painting operation that sets a pixel to a particular color; `Font.FromName` will produce a screen-independent font given the name of the typeface.

The corresponding screen-dependent resources are `ScrnPixmap.T`, `ScrnCursor.T`, `ScrnFont.T`, and `ScrnPaintOp.T`. The interfaces where these types are defined also specify the representations of the raw values—the layout of pixmaps in memory, the attributes of fonts, and similar details that all sane people prefer to avoid.

Converting a screen-independent resource into the corresponding screen-dependent resource for a particular type of screen is called *resolving* the resource. The `Palette` interface will give you a screen-independent resource if you give it a closure for resolving the resource. You can therefore use the `Palette` interface to construct your own screen-independent resources. For example, you could produce a (`PaintOp.T`, `Font.T`) pair that produces red Times Roman text on a color display and black italic text on a black-and-white display; or a `Pixmap.T` that selects between a low and a high resolution bitmap depending on the screen resolution.

The closure for resolving the resource will be invoked automatically when a top-level window moves to a new screentype for the first time. The closure will be passed an argument of type `ScreenType.T`, which represents a type of display screen. A `ScreenType.T` determines the depth of the screen's pixels (e.g., one or eight), the method for associating a color with a pixel value (e.g., color-mapped or true-color), the set of allowed operations on its pixels, and the repositories for screen-dependent fonts, cursors, and pixmaps that can be used on the screen.

Implementing your own splits. Most applications can be built by using Trestle's built-in splits and leaves, together with one or more leaf `VBT`s specific to the application. If you are programming a more sophisticated application, you may want to augment the built-in splits with some of your own. Section 9 introduces the interfaces that allow you to do this.

To implement a leaf `VBT`, you only have to supply methods to handle the events that flow down the tree (from the root to the leaves). To implement a split `VBT`, you also have to supply methods to handle the information that flows up the tree, such as painting commands or commands to change the cursor shape. The `VBTClass` interface declares these methods and presents their specifications.

Very few splits override the method for painting, since the default behavior, which is to clip to the child's domain and relay the painting to the parent, is usually what is desired. But some splits do override this method: for example, the `ZSplit`, whose child windows are allowed to overlap one another, has a paint method that clips its children's painting to the visible parts of their domains. And the top level window has a painting method that translates `VBT` painting commands into X painting commands and relays them to the X server. The interfaces `Batch`, `BatchUtil`, and `PaintPrivate` reveal the details necessary to override painting methods.

The remainder of the reference manual consists of complete Modula-3 interfaces printed in typewriter font and interspersed with commentary printed in roman font. Some of the commentary is in the form of "pseudo-Modula-3" program fragments, which are also printed in typewriter font.

The Trestle release that accompanies SRC Modula-3 version 2.0 contains several interfaces that are not documented in this reference manual. For example, the `VTextVBT` interface provides editable text `VBT`s and the `TrestleAux` interface allows you to set window manager parameters and do strange things to top-level windows. The specifications for these interfaces are directly in the Modula-3 interface files.

2 The VBT interface

A VBT.T (or simply a VBT) is the basic window abstraction of the Trestle system.

```
INTERFACE VBT;

IMPORT Word, Axis, Point, Rect, Region, Trapezoid,
  Path, Pixmap, Cursor, Font, PaintOp, ScrnPixmap;
```

2.1 The public methods

A VBT is represented as an object with a private prefix and twelve public methods, which define the way the VBT responds to events. Here are the type declarations that reveal the public methods, while concealing the private prefix:

```
TYPE
  T <: Public;
  Public = Prefix OBJECT
    METHODS
      <* LL.sup = mu *>
      mouse(READONLY cd: MouseRec);
      position(READONLY cd: PositionRec);
      redisplay();
      misc(READONLY cd: MiscRec);
      key(READONLY cd: KeyRec);
      discard();
      <* LL.sup = mu.SELF *>
      reshape(READONLY cd: ReshapeRec);
      rescreen(READONLY cd: RescreenRec);
      repaint(READONLY rgn: Region.T);
      shape(ax: Axis.T; n: CARDINAL): SizeRange;
      <* LL.sup <= mu *>
      read(sel: Selection; tc: CARDINAL): Value
        RAISES {Error};
      write(sel: Selection; val: Value; tc: CARDINAL)
        RAISES {Error};
    END;
  Prefix <: ROOT;
```

For example, if the user reshapes a window, Trestle will call the window's reshape method; if the user exposes some part of the window, Trestle will call the window's repaint method. The remainder of the VBT interface specifies the methods in detail. The pragmas about LL are explained in the section on locking level, below.

You should never call a VBT's methods directly. The VBTClass interface provides wrapper procedures that call the methods indirectly.

2.2 Screens and domains

Every VBT has a *screen* that associates a pixel value with each integer lattice point. We write v[p] to denote the value of the pixel at point p of the screen of the VBT v. Changing the pixel values in a VBT's screen is called *painting*.

The part of a VBT's screen that is visible to the user—or that would be visible if other windows weren't in the way—is called the *domain* of the VBT:

```
PROCEDURE Domain(v: T): Rect.T; <* LL.sup < v *>
```

Return the rectangular extent of the visible part of v's screen.

The domain is an arbitrary rectangle: it can be empty, the coordinate origin can be anywhere inside or outside it, and it does not necessarily correspond to the position of the window on the physical display screen.

When v is reshaped, Domain(v) changes from one rectangle to another. During this transformation Trestle tries to save the old screen until the new screen is fully repainted: thus in the midst of reshaping, v[p] can be useful for some points p outside Domain(v). At other times, Trestle keeps track of v[p] only for points p inside Domain(v).

The pragma LL.sup < v is explained in the next section.

2.3 Locking level

The global mutex mu serializes operations that affect the tree of VBTs:

```
VAR mu: MUTEX;
```

In addition, every VBT includes a private mutex that serializes operations on the VBT itself. The private mutex of a VBT is revealed in the VBTClass interface, not in this interface.

The order in which a thread is allowed to acquire these locks is called the "locking order". It is defined by these two rules:

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

The "locking level" of a thread, or LL for short, is the set of locks that the thread has acquired. The expression LL.sup denotes the maximum of the locks in LL. (The locking order is partial, but LL.sup will be defined for any thread in a correct program, since threads acquire locks in ascending order.)

Each procedure declaration in the Trestle system includes a pragma specifying the locking level at which a thread can legally call the procedure. For example, the pragma LL.sup < v on the Domain procedure allows a thread to call Domain with no locks, or with mu locked, or with descendants of v locked, but forbids calling it with any other VBTs locked.

Similarly, each public data field and method of an object has a locking level. In both cases, a locking level pragma applies to all the fields or methods between it and the next pragma. These pragmas may contain the special identifier `SELF`, which refers to the object itself.

The locking level for a method is identical to the locking level for a procedure: it specifies the locking level at which a thread can legally call the method. For example, whenever the `mouse`, `position`, `redisplay`, `misc`, `key`, or `discard` methods of a `VBT` are called, the locking level satisfies `LL.sup = mu`.

The locking level for a writable data field is of the form

```
LL >= {mu1, ..., muN}.
```

This specifies that in order to write the field, a thread must hold all of the locks `mu1` through `muN`. As a consequence, a thread can read the field if it holds any of the locks.

(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 <= mu` indicates that all locks held by the thread precede `mu` 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 `mu`; its share is denoted by `mu.v`. This is explained in the section of this interface that specifies the `reshape` method.

All the procedures in the Trestle system restore the caller's locking level when they return. For example, calling `Domain(v)` has no net effect on a thread's locking level.

2.4 ScreenTypes

Pixel values are integers. The color associated with a pixel value is determined in some manner that depends on the *screentype* of the `VBT`. A value `st` of type `VBT.ScreenType` represents a screentype:

```
TYPE
  ScreenType <: ScreenTypePublic;
  ScreenTypePublic = OBJECT (*CONST*)
    depth: INTEGER;
    color: BOOLEAN;
    res: ARRAY Axis.T OF REAL
  END;
```

The integer `st.depth` is the number of bits per pixel in screens of type `st`. The boolean `st.color` is `TRUE` if the pixels are colored, `FALSE` if they are black and white or gray-scale. The array `st.res` gives the horizontal and vertical resolution of the screen in pixels per millimeter for desk-top displays, or in visually equivalent units for other displays.

The screentype of a newly-allocated VBT is NIL; it becomes non-NIL only when the VBT is connected to a window system.

Here are two procedures for reading the screentype of a VBT and for converting distances to screen coordinates:

```
PROCEDURE ScreenTypeOf(v: T): ScreenType;
<* LL.sup < v *>
```

Return the screentype of v.

```
PROCEDURE MMToPixels(v: T; mm: REAL; ax: Axis.T)
: REAL; <* LL.sup < v *>
```

Return the number of pixels that correspond to mm *millimeters on* v*'s screentype in the axis* ax*; or return* 0 *if* v*'s screentype is* NIL.

The ScreenType interface reveals more details, for example, about color maps.

2.5 Splits and leaves

User interfaces are usually constructed from a tree of VBTs whose root is the "top-level window" known to the window manager. VBTs are classified into two main subtypes based on their positions in the tree:

```
TYPE
  Split <: T;
  Leaf <: T;

PROCEDURE Parent(v: T): Split; <* LL.sup < v *>
```

Return v*'s parent, or* NIL *if* v *has no parent.*

A Split (also called a parent VBT) divides its screen up among its children according to some layout policy that depends on the class of split. Each pixel of the parent screen represents a pixel of one of the child VBTs, which is said to control that pixel. For example, overlapping windows are provided by a class of split called a ZSplit, for which the children are ordered bottom to top, and each pixel v[p] of the parent domain is controlled by the top-most child whose domain includes p.

See the Split interface for common operations on splits (e.g., enumerating children).

A Leaf is a VBT in which the twelve public methods make the Leaf ignore all events, be indifferent about its shape, and do nothing when discarded. It is provided as a starting point: you can define a useful subtype of Leaf by overriding the methods that are relevant to the new class.

Almost all subtypes of VBT are subtypes of either Split or Leaf.

2.6 Timestamps, modifiers, mouse buttons, and cursor positions

The following types are used in several of the event methods:

```
TYPE
  TimeStamp = Word.T;

  Modifier =
    {Shift, Lock, Control, Option,
     Mod0, Mod1, Mod2, Mod3,
     MouseL, MouseM, MouseR,
     Mouse0, Mouse1, Mouse2, Mouse3, Mouse4};

  Button = [Modifier.MouseL..Modifier.Mouse4];

  Modifiers = SET OF Modifier;

  ScreenID = INTEGER;

  CursorPosition = RECORD
    pt: Point.T;
    screen: ScreenID;
    gone, offScreen: BOOLEAN;
  END;

CONST
  Buttons = Modifiers{FIRST(Button)..LAST(Button)};
```

Trestle has an internal unsigned clock register that is incremented every few milliseconds. When Trestle reports a mouse or keyboard event to a `VBT`, it also reports the value of the clock register when the event occurred, which is called the *timestamp* of the event. Timestamps serve as unique identifiers for the associated events. Also, the absolute time interval between two events can be computed by subtracting their timestamps with `Word.Minus` and multiplying by `Trestle.TickTime()`, which is the absolute interval between clock ticks.

A few keys on the keyboard are defined to be *modifiers*, like `Shift`, `Control`, and `Option`. When Trestle reports a mouse or keyboard event to a `VBT`, it also reports the set of modifier keys and buttons that were down when the event occurred. Thus the application can distinguish shifted mouse clicks from unshifted mouse clicks, for example.

The modifier `Shift` is reported if either of the keyboard's shift keys is down; similarly for `Control` and `Option`. The modifier `Lock` is reported if the lock key is locked down. If the keyboard has a key labelled `lock` but this key does not have mechanical alternate action, then the modifier `Lock` reflects the simulated state of the lock key (that is, alternate presses of the lock key turn the modifier on or off). Trestle does not define whether it reports up and down transitions for lock keys while the modifier is set.

Some Trestle servers interpret other keys as modifiers: the type definition accommodates up to four additional modifiers, `Mod0` through `Mod3`.

The mouse buttons are reported as modifiers. The naming of the first three buttons assumes a three-button mouse; in general it is assumed that there are at most eight buttons.

When Trestle reports a mouse position event to a VBT v, it also reports a value cp of type CursorPosition. The point cp.pt is the position of the cursor; the integer cp.screen identifies the screen of the window system where the event occurred; and cp.offScreen is TRUE if the position is on a different screen than v, and FALSE otherwise. If cp.offScreen is FALSE, then cp.pt is in v's coordinate system, otherwise cp.pt is in the coordinate system of cp.screen. The boolean cp.gone is TRUE if v doesn't control the position cp.pt, and FALSE if it does. If cp.offScreen is TRUE, then so is cp.gone. A position is controlled by a VBT w if a mouse-click at that position would ordinarily be delivered to w. All positions controlled by a VBT are in its domain; every pixel in the domain of a split is controlled by at most one child of that split. You should think of the positions controlled by a VBT as the visible positions in its domain.

2.7 The mouse method

Trestle calls a VBT's mouse method to report mouse clicks. The method will be called with LL.sup = mu, and takes an argument of type MouseRec.

```
TYPE MouseRec = RECORD
  whatChanged: Button;
  time: TimeStamp;
  cp: CursorPosition;
  modifiers: Modifiers;
  clickType: ClickType;
  clickCount: INTEGER;
END;

ClickType =
  {FirstDown, OtherDown, OtherUp, LastUp};
```

The method call v.mouse(cd) indicates that the mouse button cd.whatChanged went down or up at time cd.time and cursor position cd.cp.

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 field cd.modifiers reflects the state of the modifiers (either just before or just after the button transition; it is not specified which).

If cd.clickType is FirstDown, then cd.cp.gone will be FALSE.

The field cd.clickCount is the number of preceding transitions of the button that were near in time and space. For example, clickCount=3 on the final up transition of a double click. Some Trestle implementations have auxilliary interfaces that allow you to set the amount of time and mouse motion allowed.

2.8 The mouse focus rule

A split relays mouse clicks to whichever child of the split controls the pixel at the

position of the click—more or less. If this rule were applied blindly, a child could receive a down-click and never receive the corresponding up-click, which would make it impossible to program many user interfaces that involve dragging. Therefore the actual rule is more complicated.

Each split `sp` contains a variable `mouseFocus(sp)`, which records the child of the split that has received a transition of type `FirstDown` but not yet received a subsequent transition of type `LastUp`. If there is no such child, `mouseFocus(sp)` is `NIL`. The split `sp` relays the `MouseRec` `cd` by the "mouse focus rule":

```
IF some child ch controls cd.cp THEN
  w := ch;
  w.mouse(cd)
ELSE
  w := NIL
END;
IF cd.clickType = ClickType.FirstDown THEN
  mouseFocus(sp) := w
ELSE
  IF mouseFocus(sp) # NIL AND mouseFocus(sp) # w THEN
    cd.cp.gone := TRUE;
    mouseFocus(sp).mouse(cd)
  END;
  IF cd.clickType = ClickType.LastUp THEN
    mouseFocus(sp) := NIL
  END
END
```

The mouse focus is guaranteed to receive all button transitions until the last button comes up, no matter where it occurs.

2.9 The position method

Trestle calls a `VBT`'s position method to report cursor positions. The method will be called with `LL.sup = mu`, and takes an argument of type `PositionRec`.

```
TYPE PositionRec = RECORD
  cp: CursorPosition;
  time: TimeStamp;
  modifiers: Modifiers;
END;
```

The method call `v.position(cd)` indicates that at the time `cd.time` the cursor position was `cd.cp` and the set of modifiers keys that were down was `cd.modifiers`.

The next section explains how to control the delivery of cursor positions.

2.10 Tracking the cursor by setting cages

Every VBT v contains a field cage(v), which represents a set of cursor positions. As long as the cursor's position is inside v's cage, Trestle won't report the position to v. As soon as the cursor's position moves outside cage(v), Trestle reports the position to v, after first resetting v's cage to contain all cursor positions. Resetting the cage inhibits further reporting of cursor positions: to continue tracking, the position method must set a new cage.

```
TYPE
  Cage = RECORD
    rect: Rect.T;
    inOut: InOut;
    screen: ScreenID;
  END;
  InOut = SET OF BOOLEAN;

CONST
  AllScreens: ScreenID = -1;
```

The cage cg contains the cursor position cp if

- cp.pt is in cg.rect,
- cp.gone is in cg.inOut, and
- either cg.screen = AllScreens or cg.screen = cp.screen.

Trestle imposes the restriction on cages that if cg.screen = AllScreens, then cg.rect must be Rect.Full or Rect.Empty, and if cg contains no cursor positions, then it must be equal as a record to EmptyCage (which is declared below). For example, here are some useful cages:

```
CONST
  GoneCage =
    Cage{Rect.Full, InOut{TRUE}, AllScreens};
  InsideCage =
    Cage{Rect.Full, InOut{FALSE}, AllScreens};
  EverywhereCage =
    Cage{Rect.Full, InOut{FALSE, TRUE}, AllScreens};
  EmptyCage =
    Cage{Rect.Empty, InOut{}, AllScreens};
```

GoneCage contains all cursor positions that are "gone"; set it on a VBT to wait for the cursor to be over a position controlled by the VBT. The cage InsideCage is the complement of GoneCage: it contains all positions that the VBT controls. The cage EverywhereCage contains all cursor positions, and EmptyCage contains none.

Here is the procedure for setting the cage of a VBT:

```
PROCEDURE SetCage(v: T; READONLY cg: Cage);
<* LL.sup < v *>
```

Set `cage(v)` to the intersection of `cage(v)` with `cg`.

In the usual case, `SetCage` is called from `v`'s position method, at which point `v`'s cage is `EverywhereCage` and therefore the intersection just comes out to `cg`. In unusual cases, it will be found that intersecting the new cage with the old is what is required.

The procedure `CageFromPosition` is helpful for tracking the cursor continuously. By setting `CageFromPosition(cp)` in response to each cursor position `cp`, you can track the cursor as long as it moves within your `VBT`. There are two additional optional boolean arguments: setting `trackOutside` allows you to track the cursor over the whole screen containing the `VBT`; setting `trackOffScreen` allows you to track the cursor even onto other screens:

```
PROCEDURE CageFromPosition(
  READONLY cp: CursorPosition;
  trackOutside, trackOffScreen: BOOLEAN := FALSE)
  : Cage; <* LL arbitrary *>
```

`CageFromPosition(cp)` returns the cage that contains only the position `cp`; or `GoneCage` if either `cp.gone` or `cp.offScreen` is `TRUE` and the corresponding argument is not.

More precisely, `CageFromPosition` is equivalent to:

```
IF NOT cp.gone OR
  trackOutside AND NOT cp.offScreen OR
  trackOffScreen
THEN
  RETURN the cage containing only the position cp
ELSIF cp.offScreen AND trackOutside THEN
  RETURN Cage{Rect.Full, InOut{FALSE,TRUE}, cp.screen}
ELSE
  RETURN GoneCage
END
```

Finally, the following two procedures are occasionally useful:

```
PROCEDURE Outside(
  READONLY cp: CursorPosition; READONLY c: Cage)
  : BOOLEAN; <* LL arbitrary *>
```

Return whether the position `cp` is outside the cage `cg`.

```
PROCEDURE CageFromRect(READONLY r: Rect.T;
  READONLY cp: CursorPosition): Cage; <* LL arbitrary *>
```

Return `Cage{r, InOut{cp.gone}, cp.screen}`.

The effect of `SetCage(v, CageFromRect(r, cp))` is to suspend cursor positions as long as the cursor stays inside the rectangle `r` and has the same value of `gone` as `cp` does. This is useful when sweeping text selections, for example.

Splits relay cursor positions to their children. If several of the children are tracking the cursor at the same time, the order in which positions are relayed to the different children can be important. The order is determined by the following rule, which specifies the way a split `sp` forwards a `PositionRec` `cd` to its children (the variable `current(sp)` is the child that controls the last cursor position seen by `sp`):

```
IF some child ch controls cd.cp THEN
  w := ch
ELSE
  w := NIL
END;
goneCd := cd;
goneCd.cp.gone := TRUE;
IF w # current(sp) THEN
  Deliver(current(sp), goneCd)
END;
FOR all ch other than w and current(sp) DO
  Deliver(ch, goneCd)
END;
IF w # NIL THEN Deliver(w, cd) END;
current(sp) := w
```

where

```
Deliver(v, cd) =
  IF Outside(cd.cp, cage(v)) THEN
    cage(v) := EverywhereCage;
    v.position(cd)
  END
```

A split maintains its cage to be a subset of the intersection of its children's cages, so that it will receive any cursor positions that it owes its children.

2.11 The key method

Trestle calls a `VBT`'s `key` method to report keystrokes. The method will be called with `LL.sup = mu`, and takes an argument of type `KeyRec`.

```
TYPE
  KeyRec = RECORD
    whatChanged: KeySym;
    time: TimeStamp;
    wentDown: BOOLEAN;
```

```
    modifiers: Modifiers;
  END;

  KeySym = INTEGER;

CONST
  NoKey: KeySym = 0;
```

The method call `v.key(cd)` indicates that the key `cd.whatChanged` went up or down at time `cd.time`. The boolean `cd.wentDown` is true if the key went down; false if it went up. The set `cd.modifiers` reflects the state of the modifiers (either just before or just after the transition; it is not specified which).

A `KeySym` represents a symbol on a key of the keyboard. For example, there are separate `KeySyms` for upper and lower case letters. The interfaces `Latin1Key` and `KeyboardKey` specify the `KeySym` codes for many symbols that occur on standard keyboards. These interfaces are shipped with SRC Trestle but are not included in the printed version of the reference manual. The codes are chosen to agree with the X Keysym codes (see X Window System, Scheifler et al., [5] Appendix E).

If the keyboard, like most keyboards, has two symbols on some of the keys, then the `KeySym` for the down transition and later up transition might be different. For example, if the user pushes the left shift key, then the `z`/`Z` key, and then releases the keys in the same order, Trestle would report these four transitions:

left shift down, `modifiers = {} or {Shift}`
`Z` down, `modifiers = {Shift}`
left shift up, `modifiers = {} or {Shift}`
`z` up, `modifiers = {}`

Although the same physical `Z`/`z` key went down and up, the down transition is reported for the `Z KeySym` and the up transition is reported for the `z KeySym`.

The constant `NoKey` is simply an unused `KeySym` code.

To get Trestle to deliver keystrokes to a `VBT`, you make the `VBT` the owner of the keyboard focus by calling the procedure `VBT.Acquire`.

2.12 The redisplay method

A typical `VBT` has a "display invariant" that defines what its screen looks like as a function of its state. When the state changes, the display invariant is reestablished by updating the screen.

When a series of changes are made, each of which invalidates the display invariant, it is undesirable to update the screen after every change. For example, if the border width and the border texture of a `BorderedVBT` both change, it is better not to paint the intermediate state.

Therefore, Trestle keeps track of a set of `VBTs` that have been "marked for redisplay". Procedures that invalidate a `VBT`'s display invariant mark the `VBT` instead of updating the screen directly. Trestle automatically schedules a call to the `redisplay` method of every marked window (unless the window's screentype is `NIL`). The method

takes no arguments: the call `v.redisplay()` must reestablish `v`'s display invariant. It will be called with `LL.sup = mu`.

The default redisplay method for a `Leaf` calls the reshape method with an empty `saved` rectangle.

There are several procedures related to redisplay:

```
PROCEDURE Mark(v: T); <* LL.sup < v *>
```

Mark `v` for redisplay.

```
PROCEDURE IsMarked(v: T): BOOLEAN; <* LL.sup < v *>
```

Return `TRUE` if `v` is marked for redisplay.

```
PROCEDURE Unmark(v: T); <* LL.sup < v *>
```

If `v` is marked for redisplay, unmark it.

A marked window is automatically unmarked when it is redisplayed, reshaped, or rescreened. Thus the `Unmark` procedure is rarely needed.

2.13 The reshape method

Trestle calls a `VBT`'s reshape method to report changes in its domain. The method will be called with `LL.sup = mu.v` (as explained below), and takes an argument of type `ReshapeRec`.

```
TYPE ReshapeRec = RECORD
  new, prev, saved: Rect.T;
  marked: BOOLEAN
END;
```

The method call `v.reshape(cd)` indicates that the domain of `v` has changed from `cd.prev` to `cd.new`. The rectangle `cd.saved` is the subset of the previous domain that Trestle has preserved for the client in case it is of use in painting the new domain. This is the only case in which Trestle tries to save portions of a `VBT`'s screen outside its domain. After the reshape method returns, Trestle will generally forget the old parts of the screen. The boolean `cd.marked` indicates whether `v` was marked when it was reshaped; in any case, `v` is automatically unmarked as it is reshaped.

If `new = Rect.Empty` then the window is no longer visible (for example, this happens when the window is iconized). Any background threads that are painting should be stopped, since their efforts are useless.

The default reshape method for a `Leaf` calls the `repaint` method to repaint the whole new domain.

When the reshape method is called, `mu` is locked, and it will remain locked until the method returns. However, Trestle may lock `mu` and then reshape, repaint, or rescreen several `VBTs` concurrently, so 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 mu, and written mu.v. Holding mu is logically equivalent to holding mu.v for every v. Consequently, mu.v < mu in the locking order. Holding mu.v does not suffice to call a procedure that requires mu to be locked; on the other hand you cannot lock mu while holding mu.v, since this would deadlock.

2.14 The rescreen method

Trestle calls a VBT's rescreen method to report changes to its screentype. The method will be called with LL.sup = mu.v, and takes an argument of type RescreenRec.

```
TYPE RescreenRec = RECORD
  prev: Rect.T;
  st: ScreenType;
  marked: BOOLEAN;
END;
```

The method call v.rescreen(cd) indicates that the screentype of v has changed to cd.st and that its domain has changed from cd.prev to Rect.Empty. (Typically the VBT will be reshaped to a non-empty domain on the new screentype.) It is possible that cd.st=NIL. The boolean cd.marked indicates whether v was marked when it was rescreened; in any case, v is automatically unmarked as it is rescreened. VBT.Leaf.rescreen reshapes v to empty.

2.15 The repaint method

Trestle calls a VBT's repaint method to report that part of its screen has been exposed and must be repainted. The method will be called with LL.sup = mu.v, and takes an argument of type Region.T.

There are some subtleties if you are scrolling (that is, copying bits from one part of the screen to another) at the same time that Trestle is activating your repaint method. To explain them we will become more formal and precise.

Every VBT v has a "bad region" bad(v). For each point p that is in Domain(v) and not in bad(v), the pixel v[p] is displayed to the user; that is, if vis[p] denotes what is actually visible at pixel p, then we have the basic invariant

vis[p] = v[p] for all p controlled by v and outside bad(v)

Trestle can expand bad(v) at any time, as though cosmic rays had damaged the pixels.

Whenever bad(v) contains pixels that are controlled by v, Trestle will call v's repaint method by setting exposed(v) (the "exposed region" of v) to include all such pixels, and then executing the following code:

```
< bad(v) := the set difference bad(v) - exposed(v);
  FOR p in exposed(v) DO v[p] := vis[p] END >;
v.repaint(exposed(v));
exposed(v) := the empty set
```

That is, as a pixel `p` is removed from `bad(v)` and added to `exposed(v)`, the screen `v[p]` is changed to `vis[p]`, so that the basic invariant is maintained. You can imagine that the cosmic ray's damage has now reached `v[p]`, not just `vis[p]`. The angle brackets indicate that the shrinking of `bad(v)` and the damaging of `v[p]` occur atomically, so that the basic invariant is maintained. (In particular, the basic invariant is true whenever you call the procedure `VBT.Scroll`, where you can find more about the bad region and the exposed region.)

Sometimes it is convenient to do all painting from the repaint method; in which case the following procedure is useful:

```
PROCEDURE ForceRepaint(v: T; READONLY rgn: Region.T);
<* LL.sup < v *>
```

Set `bad(v) := Region.Join(rgn, bad(v))`. If the resulting `bad(v)` is non-empty, schedule an activation of `v`'s repaint method.

2.16 About painting in general

Trestle's painting procedures all follow the same pattern. The arguments to the procedure specify:

- a *destination*, which is a set of pixels in a `VBT`'s screen. For example, the destination could be a rectangle, a trapezoid, a shape bounded by a curved path, or a region.
- a *source*, which is conceptually an infinite array of pixels, not necessarily of the same depth as those on the screen. For example, the source could be a texture, a text string in some font, an explicit bitmap or image, or the `VBT`'s screen itself.
- an *operation*, which is a function that takes a destination pixel value and a source pixel value and produces a destination pixel value. For example, the operation could be planewise `XOR`.

The effect of the painting procedure is to apply the operation to each pixel in the destination region. That is, if `v` is the `VBT`, the effect of the painting procedure is to set `v[p] := op(v[p], s[p])` for each point `p` in the destination, where `op` is the operation, `v[p]` is the pixel at point `p` of `v`'s screen, and `s[p]` is the source pixel at point `p`.

Two useful operations are `PaintOp.Bg` and `PaintOp.Fg`, defined by

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

These operations ignore their arguments; they set each destination pixel to a constant value, regardless of its previous value or the source value. The actual 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 operation is `PaintOp.Copy`, defined by

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

For example, `PaintOp.Copy` can be used to paint an eight-bit pixmap source on an eight-bit pixmap screen. It would be an error to use `PaintOp.Copy` with a one-bit source and an eight-bit screen—the system wouldn't crash, but anything could happen to the destination pixels.s

For more painting operations, see the `PaintOp` interface.

2.17 Scrolling (copying one part of the screen to another)

```
PROCEDURE Scroll(
    v: Leaf;
    READONLY clip: Rect.T;
    READONLY delta: Point.T;
    op: PaintOp.T := PaintOp.Copy); <* LL.sup < v *>
```

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

The `Scroll` procedure uses `v`'s screen as source. It can therefore be used to copy pixels from one part of `v`'s screen to another. Any operation can be used for combining the translated pixels with the destination pixels, but the operation defaults to `PaintOp.Copy`.

The source rectangle can be computed from `clip` by subtracting `delta`. More precisely, `Scroll(v, clip, delta, op)` is equivalent to:

for each pair of points `p`, `q` such that
    `p` is in `clip`,
    `p = q + delta`, and
    `q` is in `Domain(v)`
simultaneously assign
    `v[p] := op(v[p], v[q]);`
    if `q` is in `exposed(v)` and `p` is not,
        or if `q` is in `bad(v)`
    then add `p` to `bad(v)`

By "simultaneously" it is meant that the pairs `p`, `q` are enumerated in an order so that no destination pixel of an early pair corresponds to a source pixel of any later pair.

Recall the bad region and exposed region `bad(v)` and `exposed(v)` from the description of the repaint method.

If you do all your painting from within the `repaint`, `reshape`, and `redisplay` methods, then you can ignore the subtleties involving the `bad(v)` and `exposed(v)`. But if you have any asynchronous threads that call `Scroll`, you have to be careful. For example, suppose you do all your painting from a concurrent worker thread, and arrange for your repaint and reshape methods to simply add entries to the worker thread's queue recording the painting that must be done. Then you must be careful to avoid the following sequence of events:

- The worker thread removes from its work queue an item indicating that it must repaint some region A, and determines that the best way to do this is to scroll some other region B.
- The `repaint` method is activated with exposed region B; it adds B to the work queue and returns. As it returns, the system sets the VBT's bad and exposed regions to be empty. (See the description of the `repaint` method.)
- The worker thread copies the garbage from B into A.

Eventually the worker thread will get around to repainting B, but the damage to A will never be repaired.

To avoid this race condition, the repaint method should convey the bad region to the worker thread by a separate communication path, rather than simply put it the ordinary work queue. The worker thread can thus avoid using bad bits as the source of scroll operations.

Of course it is possible for the scrolling to happen after the `repaint` method is called but before the method has conveyed the bad region to the worker thread. There is no way to prevent this sequence of events, but there is no need to, either: in this case the source of the scroll operation will be in the exposed region (since the `repaint` method has not yet returned), and therefore (by the specification above) the call to `Scroll` will expand the bad region. This will eventually lead to the repaint method being activated a second time, repairing the damage.

In short, in order to allow concurrent painting, we do not clear the exposed region until the `repaint` method returns, and we specify that a scroll from a `q` in `bad(v)` or `exposed(v)` to a `p` that is not in `bad(v)` invalidates the destination.

Notice that a scroll from `exposed(v)` to `exposed(v)` does not invalidate the destination. This allows the repaint method to paint a portion of `exposed(v)` and then scroll that portion to other parts of `exposed(v)`—unusual, but legal.

2.18 Painting textures

This section describes procedures for texturing rectangles, regions, and trapezoids.

```
PROCEDURE PaintTexture(
    v: Leaf;
    READONLY clip: Rect.T;
    op: PaintOp.T := PaintOp.BgFg;
    src: Pixmap.T;
    READONLY delta := Point.Origin); <* LL.sup < v *>
```

Paint the rectangle `clip` with the texture `src+delta` using the operation `op`.

A *texture* is an infinite periodic pixmap. A texture `txt` is represented by a pixmap `src` with a finite non-empty rectangular domain `Domain(src)`; the rule is that `txt` is the result of tiling the plane with translates of the pixmap `src`. Using the convenient procedure `Rect.Mod` we can state this rule as: `txt[p] = src[Rect.Mod(p, Domain(src))]`.

The texture `src+delta` is the translation of the texture `src` by the vector `delta`.

Putting this all together, `PaintTexture(v, clip, op, src, delta)` is equivalent to:

> for each pair of points `p`, `q` such that
> `p` is in `clip` and
> `p = q + delta`
> assign
> `v[p] := op(v[p], src[Rect.Mod(q, Domain(src))]).`

Note that setting `delta` to `Point.Origin` causes the texture to be aligned in an absolute coordinate system independent of the domain of the window (which helps to make textures in different windows match), while setting it to the northwest corner of `v`'s domain causes the texture to be aligned in the window's coordinate system (which allows a window to be reshaped by scrolling the old domain into the new).

If `src`'s domain is empty, the effect is undefined but limited to the clipping region.

The default paint operation for PaintTexture is `BgFg`, defined by

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

This paint operation is only appropriate if `src` is one-bit deep; the effect is to copy the source to the destination, interpreting 0 as background and 1 as foreground.

```
PROCEDURE PaintTint(
    v: Leaf;
    READONLY clip: Rect.T;
    op: PaintOp.T); <* LL.sup < v *>
```

Paint the rectangle `clip` with the texture `Pixmap.Solid` using the operation `op`.

For example, `PaintTint(v, clip, PaintOp.Bg)` paints `clip` with the background color, and `PaintTint(v, clip, PaintOp.Fg)` paints `clip` with the foreground color.

```
PROCEDURE PolyTint(
    v: Leaf;
    READONLY clip: ARRAY OF Rect.T;
    op: PaintOp.T); <* LL.sup < v *>
```

Paint each rectangle `clip[i]` in order with the texture `Pixmap.Solid` using the operation `op`.

```
PROCEDURE PolyTexture(
    v: Leaf;
    READONLY clip: ARRAY OF Rect.T;
    op: PaintOp.T := PaintOp.BgFg;
    src: Pixmap.T;
```

```
    READONLY delta := Point.Origin); <* LL.sup < v *>
```

Paint each rectangle `clip[i]` *in order with the texture* `src+delta` *using the operation* `op`*.*

```
PROCEDURE PaintRegion(
    v: Leaf;
    READONLY rgn: Region.T;
    op: PaintOp.T := PaintOp.BgFg;
    src: Pixmap.T := Pixmap.Solid;
    READONLY delta := Point.Origin); <* LL.sup < v *>
```

Paint the region `rgn` *with the texture* `src+delta` *using the operation* `op`*.*

```
PROCEDURE PaintTrapezoid(
    v: Leaf;
    READONLY clip: Rect.T;
    READONLY trap: Trapezoid.T;
    op: PaintOp.T := PaintOp.BgFg;
    src: Pixmap.T := Pixmap.Solid;
    READONLY delta := Point.Origin); <* LL.sup < v *>
```

Paint the intersection of `clip` *and* `trap` *with the texture* `src+delta` *using the operation* `op`*.*

2.19 Filling and stroking paths

Trestle also supports PostScript-like graphics operations [3]:

```
TYPE
  WindingCondition = {Odd, NonZero};
  EndStyle = {Round, Butt, Square};
  JoinStyle = {Round, Bevel, Miter};
```

```
PROCEDURE Fill(
    v: Leaf;
    READONLY clip: Rect.T;
    path: Path.T;
    wind := WindingCondition.NonZero;
    op: PaintOp.T := PaintOp.BgFg;
    src: Pixmap.T := Pixmap.Solid;
    READONLY delta := Point.Origin); <* LL.sup < v *>
```

Paint the intersection of `clip` *and the region entwined by* `path` *with the texture* `src+delta` *using the operation* `op`*.*

The point `p` is entwined by `path` if the winding number of `path` around `p` satisfies the winding condition `wind`. To ensure that the winding number is defined even for the points on the path, the path is regarded as translated north by ϵ and west by ϵ^2, where ϵ is infinitesimal.

```
PROCEDURE Stroke(
    v: Leaf;
    READONLY clip: Rect.T;
    path: Path.T;
    width: CARDINAL := 0;
    end := EndStyle.Round;
    join := JoinStyle.Round;
    op: PaintOp.T := PaintOp.BgFg;
    src: Pixmap.T := Pixmap.Solid;
    READONLY delta := Point.Origin); <* LL.sup < v *>
```

Paint the intersection of `clip` *and the stroke determined by* `path`, `end`, *and* `join` *with the texture* `src+delta` *using the operation* `op`.

The exact results of `Stroke` are different on different Trestle implementations. The approximate specification is like PostScript:

If `end = Round` and `join = Round`, the path is drawn by a circular brush of diameter `width` that traverses the path.

If `end = Butt`, then the ends of unclosed trails in the path are stroked by a line segment of length `width` centered and perpendicular to the path in the neighborhood of the endpoint. If `end = Square`, the path is extended at the endpoint by a straight line segment of length `width/2` tangent to the path and a butt end is drawn.

If `join = Bevel`, the joint between two patches is constructed by using `Butt` endstyles for them and then filling the triangular notch that remains. If `join = Miter`, then instead of just filling the triangular notch, the outer edges of the two lines are extended to meet at a point, and the resulting quadrilateral is filled.

If `width = 0`, `join` is ignored and `end` determines whether the final endpoint of an open subpath should be drawn: if `end` is `Butt`, the final endpoint is omitted, otherwise it is drawn.

Finally, there is a convenience procedure for stroking a path containing a single straight line segment:

```
PROCEDURE Line(
    v: Leaf;
    READONLY clip: Rect.T;
    p, q: Point.T;
    width: CARDINAL := 0;
    end := EndStyle.Round;
    op: PaintOp.T := PaintOp.BgFg;
    src: Pixmap.T := Pixmap.Solid;
    READONLY delta := Point.Origin); <* LL.sup < v *>
```

Like `Stroke` *applied to the path containing the segment* `(p,q)`.

2.20 Painting pixmaps

The following procedure paints a pixmap without replicating it into an infinite texture:

```
PROCEDURE PaintPixmap(
    v: Leaf;
    READONLY clip: Rect.T := Rect.Full;
    op: PaintOp.T := PaintOp.BgFg;
    src: Pixmap.T;
    READONLY delta: Point.T); <* LL.sup < v *>
```

Translate the pixmap `src` *by* `delta` *and paint it on the screen of* `v`*, using the operation* `op` *and clipping to the rectangle* `clip`*.*

More precisely, `PaintPixmap(v, clip, op, src, delta)` is equivalent to

for each pair of points `p`, `q` such that
`p` is in `clip`,
`q` is in `Domain(src)`, and
`p = q + delta`,
assign
`v[p] := op(v[p], src[q])`

Since a `Pixmap.T` is a screen-independent resource, you can't read its domain without specifying the `VBT` it is to be used on:

```
PROCEDURE PixmapDomain(v: T; pix: Pixmap.T): Rect.T;
<* LL.sup < v *>
```

Return the domain of `pix` *on the screentype of* `v`*.*

It is also possible to paint screen-dependent pixmaps:

```
PROCEDURE PaintScrnPixmap(
    v: Leaf;
    READONLY clip: Rect.T := Rect.Full;
    op: PaintOp.T := PaintOp.Copy;
    src: ScrnPixmap.T;
    READONLY delta: Point.T); <* LL.sup < v *>
```

Like `PaintPixmap`*, but with a screen-dependent pixmap instead of a screen-independent pixmap.*

If `src` does not have an appropriate screentype for `v`, the effect of the procedure is undefined but limited to the clipping region.

Because Trestle batches painting operations, the pixmap `src` must be regarded as still in use after `PaintScrnPixmap` returns. If you wish to free the pixmap by calling `src.free()`, you should first call `VBT.Sync(v)`.

2.21 Painting text

The text painting procedures take an optional array of displacements, whose entries have the following type:

```
TYPE
  DeltaH = [-512 .. 511];
  Displacement =
    RECORD index: CARDINAL; dh: DeltaH END;
```

A displacement `d` causes all characters whose index in the text is `d.index` or greater to be displaced `d.dh` pixels to the right. The first character has index `0`. The `d.index` values in an array of displacements must be non-decreasing.

```
PROCEDURE PaintText(
    v: Leaf;
    READONLY clip: Rect.T := Rect.Full;
    READONLY pt: Point.T;
    fnt: Font.T := Font.BuiltIn;
    t: TEXT;
    op: PaintOp.T := PaintOp.TransparentFg;
    READONLY dl := ARRAY OF Displacement{});
<* LL.sup < v *>
```

Paint the text `t` onto the screen of `v`, starting at position `pt`, using the font `fnt`, the operation `op`, and the displacement list `dl`.

The arguments to `PaintText` must satisfy at least one of the following two conditions:

- the background operation is transparent; that is, `op(p, 0) = p` for any pixel `p`, or
- the font is self-clearing (see below) and `dl` is empty.

If neither condition is true, the effect of `PaintText` is implementation-dependent, but is confined to the clipping rectangle.

The `ScrnFont` interface defines the properties of fonts. Here we introduce names for the properties needed to explain `PaintText`. If `f` is a font and `ch` is a character, then

- `printWidth(ch, f)` is the printing width of `ch`; that is, the amount to increment the reference point when `ch` is printed in font `f`;
- `bits(ch, f)` is the bitmap for `ch` in `f`, which is positioned with `ch`'s reference point at the origin;
- `height(ch, f)` is the height of `ch` above the baseline; that is, the number of rows of `bits(ch, f)` whose `v`-coordinate is at most zero; and `depth(ch, fnt)` is the number of rows of `bits(ch, f)` whose `v`-coordinate exceeds zero;
- `ascent(f)` and `descent(f)` are the logical extent of `f` above and below the baseline. Some characters may extend higher or lower.

A font is *self-clearing* if

- each character's height and depth equal the font's ascent and descent, and

- each character's `printWidth` equals the width of its bitmap and each character's reference point is at the west boundary of its bitmap (or each character's `printWidth` equals the negative of the width of its bitmap and each character's reference point is at the east boundary of its bitmap).

The call to `PaintText` is equivalent to the following loop:

```
rp := pt;
i := 0;
LOOP
  IF dl # NIL THEN
    FOR j := 0 TO HIGH(dl^) DO
      IF dl[j].index = i THEN INC(rp.h, dl[j].dh) END
    END
  END;
  IF i = Text.Length(t) THEN EXIT END;
  PaintPixmap(v, clip, op, bits(t[i], fnt), rp);
  rp.h := rp.h + PrintWidth(t[i], fnt);
  i := i + 1
END
```

The following two procedures are useful for computing the sizes of texts. Since fonts are screen-independent, they take the `VBT` whose screentype is to be used:

```
PROCEDURE BoundingBox
  (v: Leaf; txt: TEXT; fnt: Font.T): Rect.T;
  <* LL.sup < v *>
```

Return the bounding box of the text `txt` if it were painted at the origin on the screen of `v`.

More precisely, let `r` be the smallest rectangle that contains the bounding boxes of the characters of `txt` if `txt` were painted on `v` in the font `fnt` with `txt`'s reference point at the origin. Then `BoundingBox` returns a rectangle with the same horizontal extent as `r`, but whose height and depth are the maximum height and depth of any character in the font.

```
PROCEDURE TextWidth
  (v: Leaf; txt: TEXT; fnt: Font.T): INTEGER;
  <* LL.sup < v *>
```

Return the sum of the printing widths of the characters in `txt` in the font `fnt`.

`TextWidth` returns the displacement of the reference point that would occur if `t` were painted on `v` in font `fnt`. It may differ from the width of `BoundingBox(txt, fnt)`, since the printing width of the last character can be different from the width of its bounding box, and the reference point for the first character might not be at the left edge of `txt`'s bounding box.

You can paint characters out of an array instead of a `TEXT`:

```
PROCEDURE PaintSub(
    v: Leaf;
    READONLY clip: Rect.T := Rect.Full;
    READONLY pt: Point.T;
    fnt: Font.T := Font.BuiltIn;
    READONLY chars: ARRAY OF CHAR;
    op: PaintOp.T := PaintOp.TransparentFg;
    READONLY dl :=  ARRAY OF Displacement{});
    <* LL.sup < v *>
```

Like `PaintText` *applied to the characters in* `chars`*.*

2.22 Synchronization of painting requests

To improve painting performance, Trestle combines painting commands into batches, and sends them to the server a batch at a time.

Most applications can ignore the batching, but the procedures in this section can be of use in applications where the timing of paint operations is critical.

For example, when replacing one line of text with another in a non-self-clearing font, the old text must be erased before the new text is painted. If the painting command that erases the old text happens to fall at the end of a batch, there may be a delay of several milliseconds between the time it affects the screen and the time the following paint text command affects the screen, which can produce an undesirable flickering effect. The chances of this happening can be greatly reduced by enclosing the two commands in a *group*, using the following two procedures:

```
PROCEDURE BeginGroup(v: Leaf; sizeHint: INTEGER := 0);
<* LL.sup < v *>
```

Begin a group of painting commands.

```
PROCEDURE EndGroup(v: Leaf); <* LL.sup < v *>
```

End the current group of painting commands.

If a group of painting commands are bracketed by `BeginGroup` and `EndGroup`, Trestle will try to avoid introducing delays between the commands, such as might otherwise be introduced by batching. Trestle assumes that you will generate the painting commands and the `EndGroup` in rapid succession.

Increasing the value of `sizeHint` may improve atomicity, at the cost of throughput. The maximum useful value of `sizeHint` is the total size in bytes of the painting commands in the group, which you can compute using the interface `PaintPrivate`.

```
PROCEDURE Sync(v: Leaf); <* LL.sup < v *>
```

Wait until all painting commands issued to `v` *prior to the call to* `Sync` *have been executed.*

2.23 Screen capture

```
PROCEDURE Capture(
    v: T;
    READONLY clip: Rect.T;
    VAR (*out*) br: Region.T)
    : ScrnPixmap.T; <* LL.sup < v *>
```

Return a pixmap containing the part of `v`'s screen in the rectangle `rect`.

The screentype of the result will be the same as the screentype of `v`. Because a `VBT`'s screen is forgetful, it may be impossible to read the requested region. In this case `br` is set to contain all positions of pixels that were not copied. Naturally, Trestle makes `br` as small as it can. If none of the bits are available, the result may be `NIL`.

2.24 Controlling the cursor shape

Every `VBT` `v` contains a field `cursor(v)`, which is set with the following procedure:

```
PROCEDURE SetCursor(v: T; cs: Cursor.T);
<* LL.sup < v *>
```

Set `cursor(v)` to `cs`.

A split displays the cursor of its mouse focus, or of its current child if its mouse focus is `NIL`. Only if the cursor of the relevant child is `Cursor.DontCare` or if there is no relevant child does the split display its own cursor.

To be more precise, the shape of the cursor over the top level window `v` is determined by the following recursive procedure:

```
GetCursor(v) =
  IF NOT ISTYPE(v, Split) THEN
    RETURN cursor(v)
  ELSE
    IF mouseFocus(v) # NIL THEN
      cs := GetCursor(mouseFocus(v))
    ELSIF current(v) # NIL THEN
      cs := GetCursor(current(v))
    ELSE
      cs := Cursor.DontCare
    END;
    IF cs = Cursor.DontCare THEN
      RETURN cursor(v)
    ELSE
      RETURN cs
    END
  END
```

2.25 Selections

Trestle maintains an internal table of named selections, which initially contains several selections of general use, and which can be extended by users:

```
TYPE Selection = RECORD sel: CARDINAL END;

PROCEDURE GetSelection(name: TEXT): Selection;
<* LL arbitrary *>
```

Return the selection with the given name, creating it if necessary.

```
PROCEDURE SelectionName(s: Selection): TEXT;
<* LL arbitrary *>
```

Return the name used to create `s`*, or* `NIL` *if* `s` *is unknown.*

```
VAR (*CONST*)
  NilSel:  Selection (* := GetSelection("NilSel") *);
  Forgery: Selection (* := GetSelection("Forgery") *);
  KBFocus: Selection (* := GetSelection("KBFocus") *);
  Target:  Selection (* := GetSelection("Target") *);
  Source:  Selection (* := GetSelection("Source") *);
```

`NilSel` and `Forgery` are reserved for Trestle's internal use. The owner of `KBFocus` (the keyboard focus) is the `VBT` that receives keystrokes.

We offer the following suggestions for the use of target and source selections:

- The target selection. If text, this should be underlined black or reverse video. The selection gesture should not require modifiers like shift or control.
- The source selection. If text, this should be underlined gray. The source gesture should be a modified version of the gesture for making the target selection.

An operation like "copy" should replace the target selection with the value of the source selection.

The following exception declaration provides for the errors that can occur in dealing with selections.

```
EXCEPTION Error(ErrorCode);

TYPE ErrorCode =
  {EventNotCurrent, TimeOut, Uninstalled, Unreadable,
   Unwritable, UnownedSelection, WrongType};
```

Explanation of error codes:

- `EventNotCurrent`: Raised by attempts to access a selection with an event time that is not current.

- `TimeOut`: If you attempt to read or write a selection, and the selection owner's method does not return for an unreasonably long time, then Trestle stops waiting and raises this exception.
- `Uninstalled`: Raised by event-time operations on uninstalled `VBT`s; that is, on `VBT`s none of whose ancestors have been connected to a window system by one of the installation procedures in the `Trestle` interface.
- `Unreadable`,`Unwritable`: Raised by attempts to read an unreadable selection, or write an unwritable selection.
- `UnownedSelection`: Raised by attempts to read, write, or deliver miscellaneous codes to the owner of an unowned selection.
- `WrongType`: Raised by attempts to read or write a selection with a type not supported by the selection owner.

2.26 Acquiring and releasing selection ownership

```
PROCEDURE Acquire(
    v: T;
    s: Selection;
    t: TimeStamp)
  RAISES {Error}; <* LL.sup < v *>
```

Make `v` the owner of selection `s`, provided that `t` is the current event.

If `Acquire(v, s, t)` is successful, the previous owner of the selection will receive a miscellaneous code of type `Lost` (even if the owner is `v`). The window system affected is the one to which `v` is connected. The possible error codes are `EventNotCurrent` and `Uninstalled`.

```
PROCEDURE Release(v: T; s: Selection);
<* LL.sup < v *>
```

If the current owner of `s` is `v`, then a `Lost` code is queued for delivery to `v` and the owner of `s` becomes `NIL`

The window system affected is the one to which `v` is connected. `Release` is a no-op if the current owner is not `v` or if `v` is not installed.

2.27 The miscellaneous method

Trestle calls a `VBT`'s `misc` method to deliver miscellaneous codes. The method will be called with `LL.sup = mu`, and takes an argument of type `MiscRec`.

Trestle maintains an internal table of named miscellaneous code types, which initially contains several types of general interest, and which can be extended by users.

```
TYPE MiscRec = RECORD
  type: MiscCodeType;
```

```
  detail: MiscCodeDetail;
  time: TimeStamp;
  selection: Selection;
END;

  MiscCodeType = RECORD typ: CARDINAL END;
  MiscCodeDetail = ARRAY [0 .. 1] OF INTEGER;

PROCEDURE GetMiscCodeType(name: TEXT): MiscCodeType;
<* LL arbitrary *>
```

Return the MiscCodeType with the given name, creating it if necessary.

```
PROCEDURE MiscCodeTypeName(type: MiscCodeType): TEXT;
<* LL arbitrary *>
```

Return the name used to create `s`*, or* `NIL` *if* `s` *is unknown.*

```
CONST
   NullDetail = MiscCodeDetail {0, ..};

VAR (*CONST*)
   Deleted: MiscCodeType;
   Disconnected: MiscCodeType;
   TakeSelection: MiscCodeType;
   Lost: MiscCodeType;
   TrestleInternal: MiscCodeType;
```

These "variables" are really constants for the following codes:

```
GetMiscCodeType("Deleted")
GetMiscCodeType("Disconnected")
GetMiscCodeType("TakeSelection")
GetMiscCodeType("Lost")
GetMiscCodeType("TrestleInternal")
```

The method call `v.misc(cd)` sends `v` the misc code relevant to `cd.selection` as part of the event `cd.time`. The meaning of the `type` and `detail` fields is up to the application, except for the following.

A `Deleted` code is delivered to a top-level window when it is explicitly deleted from its server, either by a user command to the window manager or under program control. A `Disconnected` code is delivered to a top-level window when it is disconnected from its server, either because the server crashed or because the network connection was lost. A `TakeSelection` code is delivered to a top-level window when the user has gestured that it would like the window to acquire the indicated selection; most often the keyboard focus. (The nature of the gesture is between the user and the window manager. Many applications also acquire the keyboard focus in response to mouse clicks.) A `Lost` code with `selection = s` will be delivered to a window when it loses ownership of `s`. `TrestleInternal` codes are reserved for the implementation.

The timestamp in a `TakeSelection` code is the timestamp for the current event and is therefore valid for event-time operations. The timestamps in `Deleted`, `Disconnected`, and `Lost` codes are not. The selection field is relevant in `Lost` and `TakeSelection` codes; it is irrelevant in `Deleted` and `Disconnected` codes.

2.28 Sending miscellaneous codes

You can send a miscellanous code to the owner of a selection by using the following procedure:

```
PROCEDURE Put(
    v: T;
    s: Selection;
    t: TimeStamp;
    type: MiscCodeType;
    READONLY detail := NullDetail)
  RAISES {Error}; <* LL.sup < v *>
```

Create a `MiscRec` with the given fields and enqueue it for delivery to the owner of selection `s`, if `t` is the current event-time.

The window system affected is the one to which `v` is connected. The possible error codes are `EventNotCurrent`, `Uninstalled`, and `UnownedSelection`. If the selection is unowned it is possible that the `Put` will be silently ignored.

2.29 Circumventing event-time

The following procedure offers an escape from the event-time protocol. For example, a long-running thread that has no idea what the current event time is can forge a miscellaneous code to itself and use its timestamp to acquire the keyboard focus. (Your users may not like it if you do this.)

```
PROCEDURE Forge(
    v: T;
    type: MiscCodeType;
    READONLY detail := NullDetail)
  RAISES {Error}; <* LL.sup < v *>
```

Create a `MiscRec` with the given `type` and `detail` fields, with selection field `Forgery`, and with a newly created timestamp and enqueue it for delivery to `v`.

The timestamp will be valid for event-time operations (provided that it is used promptly). Forging codes that have meaning to the window manager (e.g., a `Deleted` code) could have unexpected effects if they are delivered to installed windows or their descendants. The only possible error code is `Uninstalled`.

2.30 Communicating selection values

When you read the value of a Trestle selection you get a result of type `Value`:

```
TYPE
  Value <: Value_Public;
  Value_Public =
    OBJECT METHODS toRef(): REFANY RAISES {Error} END;
```

Call the `toRef` method to convert the `Value` into a `REFANY`.

The simplest way to construct a `Value` is with the following procedure:

```
PROCEDURE FromRef(r: REFANY): Value;
<* LL.sup <= mu *>
```

Return a `Value` `v` such that `v.toRef()` is equal to the result of pickling and unpickling `r`.

On a system without pickles, the value `r` must have type `TEXT`. If `r` does not have type `TEXT`, any exceptions raised by pickling lead to checked run-time errors.

Using `FromRef` leads to synchronous transmission of selection values—that is, the value is transferred as part of the call to `Read` or `Write`. To get asynchronous behavior, allocate your own `Values` and override the `toRef` method. Trestle will transmit the `Value` to the other application, and only when that application calls the `toRef` method will your `toRef` method be called.

The `toRef` method in a `Value` will be called with `LL.sup <= mu`. The `toRef` method can raise the error `Unreadable` if, for example, the address space of the selection owner has been destroyed. It can also raise the error `WrongType` if the underlying `REFANY` cannot be represented in the address space calling the method; this can only happen with non-`TEXT` selections.

The procedure `Ready` tests whether a value is synchronous or asynchronous:

```
PROCEDURE Ready(v: Value): BOOLEAN; <* LL.sup <= mu *>
```

Return `TRUE` if calling `v.toRef()` will return quickly; return `FALSE` if calling `v.toRef()` might be slow or block.

Finally, here are the procedures for reading and writing selections:

```
PROCEDURE Read(
    v: T;
    s: Selection;
    t: TimeStamp;
    tc: INTEGER := -1)
    : Value
  RAISES {Error}; <* LL.sup <= mu *>
```

Return the value of selection `s` as a reference of type `tc`, if `t` is the current event-time.

If `tc = -1`, `Read` uses the typecode for `TEXT`. The window system affected is the one to which `v` is connected. The `KBFocus` selection is always unreadable. If the selection owner's read method is erroneous, calling the `toRef` method of the returned `Value`

may produce a reference with a typecode other than `tc`. The possible error codes are `EventNotCurrent`, `Uninstalled`, `Unreadable`, `WrongType`, `TimeOut`, and `UnownedSelection`.

```
PROCEDURE Write(
    v: T;
    s: Selection;
    t: TimeStamp;
    val: Value;
    tc: INTEGER := -1)
  RAISES {Error}; <* LL.sup <= mu *>
```

Replace the selection `s` with the value `v`, which encodes a reference with typecode `tc`, assuming `t` is the current event-time.

If `tc = -1`, `Write` uses the typecode for `TEXT`. The window system affected is the one to which `v` is connected. The `KBFocus` selection is always unwritable. The possible error codes are `EventNotCurrent`, `Uninstalled`, `Unwritable`, `TimeOut`, and `WrongType`.

2.31 The read and write methods

Trestle calls a `VBT`'s read and write methods to access any selections that it owns. The method will be called with `LL.sup <= mu` (see below).

The signature of the read method is

```
(s: Selection; tc: CARDINAL): Value RAISES {Error}
```

Trestle calls `v.read(s, tc)` whenever `v` is the owner of selection `s` and some application passes `s` and `tc` to `Read`. The method should return the value of the selection, or raise `Error(Unreadable)` if for some reason the value cannot be delivered, or `Error(WrongType)` if the selection cannot be converted to the requested type. The methods will be called with `LL.sup <= mu`; in fact, if the caller of `Read` is in the same address space, `LL` for the method call is the same as `LL` for the caller of `Read`, else `LL` for the method call is `{}`.

The signature of the write method is

```
(s: Selection; val: Value; tc: CARDINAL)
RAISES {Error}
```

Trestle calls `v.write(s, val, tc)` whenever `v` is the owner of selection `s` and some application passes `s`, `val`, and `tc` to `Write`. The method should replace the selection with the value of `val`, or raise the exception with error code `Unwritable` if for some reason the selection is not writable, or with error code `WrongType` if the selection cannot be written with the requested type. Trestle does not enforce any consistency between `tc` and the typecode of the reference `val.toRef()`. For example, if `val.toRef()` is `NIL`, the meaning could be determined by `tc`. The locking level is the same as for the read method.

While a read or write method is active in a descendant of an installed window, Trestle will block the delivery to that window of any mouse or key events, misc codes, or cursor positions. If the computations are long, it is therefore preferable to do them asynchronously, to avoid blocking the user.

2.32 Controlling the shape of a VBT

The preferred shape of a `VBT` is represented by a pair of records of type `SizeRange`, one for each axis:

```
TYPE SizeRange = RECORD lo, pref, hi: CARDINAL END;

CONST DefaultShape =
  SizeRange{lo := 0, pref := 0, hi := 99999};
```

If a `VBT`'s preferred shape in the axis `ax` is the `SizeRange` `sh`, then the desirable sizes for the `VBT` in axis `ax` range from `sh.lo` to `sh.hi-1`, and its preferred size is `sh.pref`.

A `SizeRange` `sh` is illegal unless `sh.lo <= sh.pref < sh.hi`.

When a parent `VBT` divides its screen up between its children, it tries to satisfy its children's shape requirements, which it finds by calling the children's shape method.

The signature of the shape method is

```
(ax: Axis.T; n: CARDINAL): SizeRange
```

The behavior of the shape method depends on whether `n` is zero. The call `v.shape(ax, 0)` returns the preferred shape for `v` in the `ax` axis, assuming nothing is known about its size in the other axis. If `n#0`, the call `sh := v.shape(ax, n)` returns the preferred shape for `v` in the `ax` axis assuming that `v`'s size in the other axis is `n`. When the method is called, `LL.sup = mu.v`.

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

The child must not assume that its shape requirement is satisfied, since, for example, the requirements of a split's children can be inconsistent.

The default `shape` method for a `Leaf` returns `DefaultShape`.

When the preferred shape of a `VBT` changes, you should call `NewShape`:

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

Notify `v`'s parent that its preferred size range may have changed.

Typically, the parent will mark itself, and any change will take effect at the time of the next redisplay. Notice that the locking level allows `NewShape` to be called from a `reshape` or `rescreen` method; it can also be called from a thread that has `mu` locked.

2.33 Putting properties on a VBT

Associated with each window is a "property set", which is a set of non-nil traced references.

```
PROCEDURE PutProp(v: T; ref: REFANY); <* LL.sup < v *>
```

Add `ref` to `v`'s property set, replacing any existing reference of the same type as `ref`. This is a checked runtime error if `ref` is `NIL`.

```
PROCEDURE GetProp(v: T; tc: INTEGER): REFANY;
<* LL.sup < v *>
```

Return the element of `v`'s property set with typecode `tc`, or `NIL` if no such element exists.

```
PROCEDURE RemProp(v: T; tc: INTEGER); <* LL.sup < v *>
```

Remove the element with typecode `tc` from `v`'s property set, if one exists.

2.34 Discarding a VBT

It is good form to call `VBT.Discard(v)` when `v` is about to be garbage-collected:

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

Prepare for and call `v.discard()`.

The discard method will be called with `LL.sup = mu`, and takes no argument. The method should perform any class-dependent cleanup that is needed. The default discard method is a no-op.

```
END VBT.
```

3 The Trestle interface

The `Trestle` interface provides routines for connecting to window systems; installing, decorating, and moving top-level windows, and performing related operations.

```
INTERFACE Trestle;

IMPORT VBT, Rect, Point, Region, ScrnPixmap,
  TrestleComm;

TYPE
  T <: ROOT;
```

A `Trestle.T` identifies an instance of a window system. All the routines in this interface that take a `Trestle.T` accept the value `NIL`, which represents the default window system obtained by calling `Connect(NIL)`.

```
PROCEDURE Install(
    v: VBT.T;
    applName: TEXT := NIL;
    inst: TEXT := NIL;
    windowTitle: TEXT := NIL;
    iconTitle: TEXT := NIL;
    bgColorR: REAL := -1.0;
    bgColorG: REAL := -1.0;
    bgColorB: REAL := -1.0;
    iconWindow: VBT.T := NIL;
    trsl: T := NIL)
  RAISES {TrestleComm.Failure}; <* LL.sup <= VBT.mu *>
```

Initiate the installation of `v` as a decorated top-level window of the window system `trsl`.

`Install` may return before the installation is complete. `Install` is a checked runtime error if `v` is not detached, or if `v` is in the process of being installed. The position of the window on the screen depends on the window manager.

The text `applName` is the application name; it defaults to the application name from the process environment.

The text `inst` distinguishes windows with the same application name. For example, a text editor might use the full path name of the file being edited as the instance. The default is the value of the environment variable `WINSTANCE`.

Trestle does not require that the pair `(applName, inst)` be unique, but session management tools will work more smoothly if it is.

The text `windowTitle` will be placed in the window's title bar when the window is not iconic. It defaults to the concatenation of `applName`, a space, and `inst`, or just to `applName` if `inst` is `NIL`.

The icon for the window will contain the text `iconTitle` together with `iconWindow` (if it is not `NIL`). For example, `iconWindow` might be a small `BitmapVBT`.

Some window managers ignore `iconWindow`. The default for `iconTitle` is `inst`, or `applName` if `inst` is `NIL`.

The triple `bgColorR`, `bgColorG`, `bgColorB` specify the red, green, and blue components of the background color for the window and icon titles. If they are defaulted, the window manager's default background color will be used; if they are not defaulted they should be between `0.0` and `1.0`. Some window managers ignore the background color.

An installed window's maximum, minimum, and preferred size will be reported to the window manager, initially and whenever they change. However, a `StableVBT` filter is inserted above each installed window, so that a new preferred size will not be reported if the window's current size satisfies the new max and min constraints. Use `StableVBT.Disable` to force a new preferred size.

It is a checked runtime error if either `v` or `iconWindow` is already installed.

Installing a window inserts one or more filters above it, including a `HighlightVBT`, a `StableVBT`, and filters that make screen-independent resources work.

```
PROCEDURE AwaitDelete(v: VBT.T); <* LL = {} *>
```

Wait until `v` is deleted or disconnected from whatever window system it is installed on.

`AwaitDelete` will not return until after the `Deleted` or `Disconnected` code has been delivered and processed by the window. It is a noop if `v` is already deleted or is not installed.

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

Delete `v` from wherever it is installed.

`Delete` automatically releases any selections owned by `v` or any of `v`'s descendants. Before `Delete(v)` returns, lost codes will be delivered for any such selections. If `v` owned the mouse focus, `v` will also receive a synthesized mouse transition of type `LastUp`. Then `v` will receive a `Deleted` code, and finally `Delete` will return. At this point `v` is disconnected and can be re-installed.

```
PROCEDURE Decorate(
    v: VBT.T;
    instance: TEXT := NIL;
    windowTitle: TEXT := NIL;
    iconTitle: TEXT := NIL;
    bgColorR: REAL := -1.0;
    bgColorG: REAL := -1.0;
    bgColorB: REAL := -1.0;
    applName: TEXT := NIL;
    iconWindow: VBT.T := NIL)
  RAISES {TrestleComm.Failure}; <* LL.sup = VBT.mu *>
```

Change the decorations of `v` to the given values

Any parameter that is defaulted will not be changed, unless v has been `Attached` since it was last decorated, in which case the default value is computed as in `Install`. `Decorate` is a noop if `v` is not an installed decorated window.

```
PROCEDURE GetDecoration(v: VBT.T;
  VAR instance, windowTitle, iconTitle, applName: TEXT;
  VAR bgColorR, bgColorG, bgColorB: REAL;
  VAR iconWindow: VBT.T): BOOLEAN; <* LL.sup = VBT.mu *>
```

If `v` is decorated, fetch `v`'s decorations, and return `TRUE`. Otherwise, return `FALSE`.

3.1 Window placement

```
PROCEDURE Attach(v: VBT.T; trsl: T := NIL)
  RAISES {TrestleComm.Failure}; <* LL.sup = VBT.mu *>
```

Attach `v` to the window system `trsl`, leaving it invisible.

`Attach` is like `Install`, except (1) the locking level is different, (2) the attachment is completed before `Attach` returns, (3) the window becomes undecorated, and (4) the window remains invisible until you call `Overlap`, `Iconize`, or `MoveNear`. Before calling one of these, most clients will want to call `Decorate`.

```
PROCEDURE Overlap(
    v: VBT.T;
    id: ScreenID;
    READONLY nw: Point.T)
  RAISES {TrestleComm.Failure}; <* LL.sup = VBT.mu *>
```

Move the northwest corner of `v` to the point `nw` on the screen `id`.

If `v` is undecorated, this produces a window with no title bar or border, and the user will probably not be able to move, iconize or delete the window; this is a bad idea unless you're implementing pop-up or pull-down menus.

```
PROCEDURE Iconize(v: VBT.T)
  RAISES {TrestleComm.Failure}; <* LL.sup = VBT.mu *>
```

Make the window `v` become iconic.

```
PROCEDURE MoveNear(v, w: VBT.T)
  RAISES {TrestleComm.Failure}; <* LL.sup = VBT.mu *>
```

Move the window `v` to be near the window `w`.

The exact effect of `MoveNear` depends on the window manager. If `w` is `NIL` or is not installed where `v` is, then `MoveNear` will attempt to bring `v` to the attention of the user; in particular, if `v` is an overlapping window, `v` will be brought to the top; if `v` is

an icon, it will be deiconized; if v is in the invisible state produced by Attach, it will be opened in some visible place.

Overlap, Iconize, and MoveNear are all no-ops if v is not installed. The effects of Iconize and MoveNear are undefined for undecorated windows.

```
PROCEDURE InstallOffscreen(
    v: VBT.T;
    width, height: CARDINAL;
    preferredScreenType: VBT.ScreenType)
  RAISES {TrestleComm.Failure}; <* LL.sup = VBT.mu *>
```

Give v a domain with the given dimensions in the off-screen memory of the window system to which it is attached.

InstallOffscreen rescreens v to preferredScreenType, or something as much like it as supported for off-screen windows. The window v must be in the floating state produced by Attach. The usual purpose is to paint on v and then use VBT.Capture to retrieve the contents of its screen as a pixmap. You should delete v when you are done with it. Until v is deleted, you should not pass it to Overlap, Iconize, MoveNear or InstallOffscreen.

3.2 Enumerating and positioning screens

A window system may have multiple screens. Each screen is identified by an integer.

```
TYPE ScreenID = INTEGER;

CONST NoScreen: ScreenID = -1;

TYPE ScreenOfRec = RECORD
  id: ScreenID;
  q: Point.T;
  trsl: T;
  dom: Rect.T
END;

PROCEDURE ScreenOf(
  v: VBT.T; READONLY p: Point.T)
  : ScreenOfRec; <* LL.sup < v *>
```

Return information about where v is installed.

If v is an installed window then after res := ScreenOf(v, p) we have

- res.id is the ID of the screen currently containing v;
- res.q is the point in screen coordinates that corresponds to the point p in window coordinates;
- res.trsl is the window system on which v is installed; and
- res.dom is the domain of the screen res.id.

The point `p` need not be in the domain of `v`. If `v` is not installed, then `res.trsl` will be `NIL`, `res.id` will be `NoScreen`, and the other fields will be arbitrary. If the window manager is moving `v` between screens when `ScreenOf` is called, then `res.id` will be `NoScreen` and `res.dom` and `res.q` will be arbitrary.

```
TYPE
  Screen = RECORD
    id: ScreenID;
    dom: Rect.T;
    delta: Point.T;
    type: VBT.ScreenType
  END;
  ScreenArray = REF ARRAY OF Screen;

PROCEDURE GetScreens(trsl: T := NIL): ScreenArray
  RAISES {TrestleComm.Failure}; <* LL.sup = VBT.mu *>
```

Return an array of descriptors of the screens of the window system `trsl`.

For each `Screen s` in the returned array, the rectangle `s.dom` is the domain of the `VBT` at the root of the screen. The screens all lie in a global coordinate system, within which the user moves the cursor. The point `p` in screen coordinates corresponds to the point `p+s.delta` in global coordinates. (Some window systems don't support this; in which case `s.delta` will be set to `Point.Origin` for all screens.) The value `s.type` is the screentype of the screen's root `VBT`. `GetScreens` returns `NIL` if the window system has no screens.

3.3 Reading pixels from a screen

```
PROCEDURE Capture(
    id: ScreenID;
    READONLY clip: Rect.T;
    VAR (* out *) br: Region.T;
    trsl: T := NIL)
    : ScrnPixmap.T
  RAISES {TrestleComm.Failure};
  <* LL.sup = VBT.mu *>
```

Read the contents of `clip` from screen `id` of `trsl`.

`Capture(id, clip, br, trsl)` is like `VBT.Capture(r, clip, br)`, where `r` is the `VBT` at the root of screen `id` of the window system `trsl`.

3.4 Checking on recent input activity

```
PROCEDURE AllCeded(trsl: T := NIL): BOOLEAN
    RAISES {TrestleComm.Failure}; <* LL.sup = VBT.mu *>
```

Return whether there is pending input from `trsl`.

If a program calls `AllCeded(t)` and `TRUE` is returned, then there are no mouse clicks or keystrokes on their way to any top-level windows installed by the program on `t`. For example, when the VT100 terminal emulator has observed a key-down and waited for half a second and observed no key-up and concludes that it should go into auto-repeat mode, it verifies that `AllCeded` returns `TRUE` to make sure that the up transition is not on its way, to avoid erroneously entering auto-repeat mode.

```
PROCEDURE TickTime(trsl: T := NIL): INTEGER;
<* LL.sup <= VBT.mu *>
```

Return the number of microseconds per `VBT.TimeStamp`, in events reported to `VBTs` connected to the window system `trsl`.

3.5 Connecting to a window system

```
PROCEDURE Connect(inst: TEXT := NIL): T
  RAISES {TrestleComm.Failure}; <* LL.sup <= VBT.mu *>
```

Connect to the window system named `inst`.

In general, the format and interpretation of `inst` are implementation-dependent. Here are the rules when using an X server:

If `inst` is `NIL`, it defaults to the value of the environment variable `DISPLAY`, unless this variable is undefined, in which case it defaults to `:0`.

The syntax of `inst` should be:

```
<machine name>(":" | "::")<number>("" | "." <number>)
```

where `<machine name>` is an arbitrary string of characters (possibly empty) and `<number>` is a non-negative decimal integer. It denotes an X server according to the rules on page 27 of the second edition of *X Window System*, by Scheifler et. al., Digital Press, 1990 [5].

For example, `nemesia:0` denotes the first window system on the machine `nemesia`, and `:0` denotes the first window system on the machine calling `Connect`.

The exception is raised if the designated window system doesn't exist or cannot be connected to.

```
END Trestle.
```

4 Splits

4.1 The Split interface

The Split interface provides the functionality that is common to all splits; for example, enumerating and deleting children.

This interface is for clients of splits; see the `VBTClass` and `ProperSplit` interfaces for information about implementing your own split classes.

```
INTERFACE Split;

IMPORT VBT, Point, Rect;

TYPE T = VBT.Split;

EXCEPTION NotAChild;
```

A `Split.T` is a `VBT` that divides its screen up between one or more child `VBTs`. The children of a split are ordered; they can be enumerated with the `Succ` and `Pred` procedures:

```
PROCEDURE Succ(v: T; ch: VBT.T): VBT.T
RAISES {NotAChild}; <* LL >= {VBT.mu} *>
```

Return the child of `v` that follows the child `ch`.

The successor of `NIL` is the first child; the successor of the last child is `NIL`; the successor of `NIL` is `NIL` if there are no children. The exception `NotAChild` is raised if `ch` is not a child of `v`.

```
PROCEDURE Pred(v: T; ch: VBT.T): VBT.T
RAISES {NotAChild}; <* LL >= {VBT.mu} *>
```

Return the child of `v` that precedes the child `ch`.

More precisely, `Pred(v,ch) = x` iff `Succ(v,x) = ch`. All of Trestle's standard splits implement `Succ` and `Pred` in constant time.

```
PROCEDURE NumChildren(v: T): CARDINAL;
<* LL >= {VBT.mu} *>
```

Return the number of children of `v`.

```
PROCEDURE Nth(v: T; n: CARDINAL): VBT.T;
<* LL >= {VBT.mu} *>
```

Return the child of `v` with index `n`.

More precisely, `Nth(v, n)` is the child of `v` with `n` predecessors, or `NIL` if `v` has at most `n` children. Warning: for Trestle's standard splits, `Nth` requires time proportional to `n`, so it would be wasteful to enumerate the children by calling it repeatedly; use `Succ` instead.

```
PROCEDURE Index(v: T; ch: VBT.T): CARDINAL
RAISES {NotAChild}; <* LL >= {VBT.mu} *>
```

Return the index of `v`*'s child* `ch`*.*

`Index(v, ch)` is the value `n` such that `Nth(v, n) = ch`. `Index(v, NIL)` equals `NumChildren(v)`.

```
PROCEDURE Locate(v: T; READONLY pt: Point.T): VBT.T;
<* LL.sup = VBT.mu *>
```

Return the child of `v` *that controls the point* `pt`*, or* `NIL` *if there is no such child.*

```
PROCEDURE Delete(v: T; ch: VBT.T) RAISES {NotAChild};
<* LL.sup = VBT.mu *>
```

Delete the child `ch` *of the split* `v`*, detach* `ch`*, and mark* `v` *for redisplay.*

```
PROCEDURE Replace(v: T; ch, new: VBT.T)
RAISES {NotAChild}; <* LL.sup = VBT.mu *>
```

Replace child `ch` *of* `v` *with* `new`*, detach* `ch` *(which must not be* `NIL`*), and mark* `v` *for redisplay.*

```
PROCEDURE Insert(v: T; pred, new: VBT.T)
RAISES {NotAChild}; <* LL.sup = VBT.mu *>
```

Add `new` *as a child of* `v` *following* `pred`*.*

Some split classes can accomodate only a bounded number of children (for example, filters). If `Insert(v, pred, new)` is applied to a split `v` that cannot accomodate an additional child, then `pred` (or the original first child, if `pred = NIL`) is deleted from the split and discarded. The precise semantics are defined by the individual splits. `Insert` raises `NotAChild` if `pred` isn't a child of `v`, and is a checked run-time error if `new` isn't detached.

```
PROCEDURE Move(v: T; pred, ch: VBT.T)
RAISES {NotAChild}; <* LL.sup = VBT.mu *>
```

Move child `ch` *of* `v` *to follow* `pred`*. Both* `ch` *and (if non-*`NIL`*)* `pred` *must be children of* `v`*.*

```
PROCEDURE AddChildArray(v: T;
  READONLY new: ARRAY OF VBT.T);
<* LL.sup = VBT.mu *>
```

Insert the non-`NIL` *elements of* `new` *at the end of the* `v`*'s list of children.*

`AddChildArray` is equivalent to

```
pred := Pred(v, NIL);
FOR i := 0 TO LAST(new) DO
  IF new[i] # NIL THEN
```

```
        InsertAfter(v, pred, new[i]);
        pred := new[i]
      END
    END

PROCEDURE AddChild(v: T;
  v0, v1, v2, v3, v4, v5, v6, v7, v8, v9: VBT.T := NIL);
<* LL.sup = VBT.mu *>
```

Add the given children to `v`.

`AddChild` is equivalent to

```
    AddChildArray(v,
      ARRAY OF VBT.T{v0, v1, ..., v9})

END Split.
```

4.2 The ZSplit interface

A `ZSplit.T` is a parent window with overlapping child windows.

Each child has a stacking order given (conceptually) by a `z` coordinate. A pixel of the parent's screen that is in the domain of more than one child is controlled by whichever of these children is highest in the `z` coordinate. The portions of the domains of the children that extend outside the parent domain will be clipped.

`Split.Succ` enumerates the children from top to bottom.

The bottom child is called the *background*. An initial background can be specified when the `ZSplit` is created; usually it remains the background throughout the life of the `ZSplit`. Usually the background has the same domain as the parent, and therefore controls all pixels that are not controlled by any other child. In the unusual case that the background child has a domain different from the parent domain, there may be some parent pixels that are not controlled by any child. The `ZSplit` will ignore these pixels when asked to repaint.

The shape of a `ZSplit` is the shape of its background child (if it has no children its shape is the default shape for a `VBT`). When the preferred shape of a non-background child changes, the `ZSplit` reshapes the child to its new preferred shape, preserving its *offset*, which is the vector between the northwest corners of the parent and child.

```
INTERFACE ZSplit;

IMPORT VBT, Rect, Split, Point;

TYPE
  T <: Public;
  Private <: Split.T;
  Public = Private OBJECT METHODS
    <* LL <= VBT.mu *>
```

```
    init(bg: VBT.T := NIL;
      saveBits := FALSE;
      parlim: INTEGER := -1): T
  END;
```

The call `v.init(...)` initializes `v` as a `ZSplit`.

It is only legal to call the `init` method for a newly-allocated `ZSplit` (as in the definition of the procedure `New` below) or from the `init` method of a subclass. This restriction applies to all the `init` methods in Trestle, although it will not be repeated for each one.

The `ZSplit` will be given the initial background child `bg` if `bg#NIL`; it will be given no children if `bg=NIL`. If `bg` is non-`NIL` it will be mapped initially. If `saveBits` is `TRUE`, the split will try to save the children's old bits when reformatting; if the children don't use them anyway, it is faster to let `saveBits` default to `FALSE`. The value of `parlim` is the minimum area of a child for which a separate thread will be forked to reshape or repaint it; if it is `-1`, it is set to an appropriate default (see the `VBTTuning` interface).

```
PROCEDURE New(
    bg: VBT.T := NIL;
    saveBits := FALSE;
    parlim: INTEGER := -1)
    : T; <* LL <= VBT.mu *>
```

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

4.2.1 Inserting children

The default `Split.Insert` call is rarely useful for a `ZSplit`: it inserts the new child at the parent's northwest corner, unmapped. `Split.AddChild` is even less useful, since it adds children as the background, which is almost certainly not what you want. The following procedures are more useful for inserting children into a `ZSplit`:

```
PROCEDURE InsertAfter(
    v: T;
    pred, ch: VBT.T;
    READONLY dom: Rect.T;
    alsoMap: BOOLEAN := TRUE) RAISES {Split.NotAChild};
<* LL.sup = VBT.mu *>
```

Insert `ch` as a new child of `v` with domain `dom`, and mark `v` for redisplay.

The new child is inserted immediately after (that is, below) `pred`; if `pred=NIL` the new child is inserted first (that is, on top). If the height or width of `dom` does not satisfy `ch`'s size contraints, then the height and width of the child are projected into range; its

offset is preserved. This is a checked runtime error if `ch` is not detached. If `alsoMap` is `TRUE`, `ch` is mapped, otherwise it is unmapped.

It is occasionally useful to insert a new child below all existing children except the background, in which case the following procedure is handy:

```
TYPE Altitude = {Top, Bot};

PROCEDURE Insert(
    v: T;
    ch: VBT.T;
    READONLY dom: Rect.T;
    alt := Altitude.Top;
    alsoMap: BOOLEAN := TRUE); <* LL.sup = VBT.mu *>
```

Insert `ch` at the top if `alt = Altitude.Top`; insert `ch` just above the background if `alt = Altitude.Bot`.

That is, `Insert` is equivalent to

```
IF alt = Altitude.Top THEN
  pred := NIL
ELSE
  pred := Split.Pred(v, Split.Pred(v, NIL))
END;
InsertAfter(v, pred, ch, dom, alsoMap)
```

Finally, instead of providing the new child's domain it can be useful to provide only the northwest corner and let the child's domain be determined by its shape constraints:

```
PROCEDURE InsertAt(
    v: T;
    ch: VBT.T;
    nw: Point.T;
    alt := Altitude.Top;
    alsoMap: BOOLEAN := TRUE); <* LL.sup = VBT.mu *>
```

Insert `ch` with its preferred shape and its northwest corner at `nw`. The `alt` and `alsoMap` parameters are interpreted as in `Insert`.

4.2.2 Moving, lifting, and lowering children

```
PROCEDURE Move(ch: VBT.T; READONLY dom: Rect.T);
<* LL.sup = VBT.mu *>
```

Change the domain of `ch` to be `dom` and mark `ch`'s parent for redisplay.

If the height or width of `dom` do not satisfy `ch`'s size constraints, then they are projected into range, preserving the northwest corner of `dom`. The stacking order of `ch` is not changed. `Move` is a checked runtime error if `ch`'s parent is not a `ZSplit`. Note that this has nothing to do with `Split.Move`, unlike the next procedure.

```
PROCEDURE Lift(ch: VBT.T; alt := Altitude.Top);
<* LL.sup = VBT.mu *>
```

Lift `ch` *to the top or lower it to be just above the background, depending on* `alt`. `Lift` *is equivalent to:*

```
v := VBT.Parent(ch);
IF alt = Altitude.Top THEN
  pred := NIL
ELSE
  pred := Split.Pred(v, Split.Pred(v, NIL))
END;
Split.Move(v, pred, ch)
```

4.2.3 Mapping and unmapping children

You can *unmap* a child of a `ZSplit`, which reshapes the child to be empty after recording the child's shape and offset. When you later *map* the child, the recorded shape and offset are restored. An unmapped child is rescreened when the parent is rescreened, and its recorded shape and offset are updated when the parent is reshaped, just like the domains of the mapped children.

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

If `ch` *is mapped, unmap it and mark its parent for redisplay.*

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

If `ch` *is unmapped, map it and mark its parent for redisplay.*

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

Return `TRUE` *if ch is* `mapped` *and* `FALSE` *if* `ch` *is unmapped.*

`Map`, `Unmap`, and `IsMapped` are checked runtime errors if `ch`'s parent is not a `ZSplit`.

4.2.4 Getting domains

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

Return the effective domain of `ch`.

The effective domain is the same as the normal domain, except (1) if the parent has been marked for redisplay, `GetDomain` returns the domain that `ch` will receive when the redisplay happens, or (2) if the domain of the parent is `Rect.Empty`, `GetDomain` returns the domain `ch` would receive if the parent were reshaped to its last non-empty domain, or (3) if the child is unmapped, `GetDomain` returns the domain the child would have if it were mapped.

`GetDomain` is a checked runtime error if the parent of `ch` is not a `ZSplit`.

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

Return the last non-empty value of `v.domain`*, or* `Rect.Empty` *if* `v.domain` *has always been empty.*

4.2.5 Moving children when the parent is reshaped

You can supply procedures to control what happens to the children when a `ZSplit` is reshaped. If you don't supply a procedure, the default behavior is as follows: the initial background child is always reshaped to have the same domain as the parent. The other children are reshaped so as to preserve their shape and their offsets (even if this makes them extend outside the parent domain). The rule is different if the parent is reshaped to `Rect.Empty`: in this case the `ZSplit` records its children's shapes and offsets and reshapes them all to `Rect.Empty`. When the `ZSplit` is later reshaped to a non-empty domain, it reshapes the initial background child to have the same domain as the parent, and restores the saved dimensions and offsets of the other children.

In the unusual case that the initial background child is deleted, subsequent background children do not automatically inherit the special reshaping behavior of the initial background child.

To override the default behavior, use `SetReshapeControl`:

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

Set the reshape control object for the child `ch` *to be* `rc`*.*

```
TYPE ReshapeControl = OBJECT METHODS
  apply(ch:VBT.T; READONLY old, new, prev: Rect.T)
  : Rect.T <* LL.sup = VBT.mu.ch *>
END;
```

`SetReshapeControl` arranges that whenever the `ZSplit` parent `v` of `ch` is reshaped from domain `old` to domain `new`, then if the previous domain of `ch` is `prev`, the new domain of `ch` will become `rc.apply(ch, old, new, prev)` (if this rectangle doesn't satisfy `ch`'s size constraints, its height and width will be projected into range, preserving its offset).

These methods of the `ReshapeControl` objects may be called concurrently for different children. (This is why the apply method has only a share of `VBT.mu`.) The stacking order is not changed by reshaping.

When a `ZSplit` child is replaced by `Split.Replace`, the new child inherits the old child's reshape control object.

`SetReshapeControl` is a checked runtime error if the parent of `ch` is not a `ZSplit`.

If the `ZSplit` is reshaped to `Rect.Empty`, it will reshape its children to `Rect.Empty` without calling their reshape control methods. Similarly, if the parent is subsequently reshaped to its original rectangle, it will restore the children's previous domains without calling the methods.

One useful reshape control method provided by this interface is `ChainReshape`, in which some set of the child's west, east, north, and south edges are "chained" to the corresponding edges of the parent. Chaining an edge means that the distance between the child edge and the corresponding parent edge will be preserved. For example, if both the west and east edges are chained, then the child's horizontal extent will be inset into the parent's horizontal extent by fixed amounts on both sides. For another example, suppose that the the east edge is chained and the west edge is not. In this case the distance between the east edges of the child and parent will be preserved, but the west edge of the child will move so as to preserve the width of the child. The north and south edges control the vertical extent in a similar manner.

```
TYPE
  Ch = {W, E, N, S};
  ChainSet = SET OF Ch;
  ChainReshapeControl = ReshapeControl OBJECT
    chains: ChainSet
  OVERRIDES
    apply := ChainedReshape
  END;

VAR (*CONST*)
  NoChains, WChains, EChains, WEChains, NChains,
  WNChains, ENChains, WENChains, SChains,
  WSChains, ESChains, WESChains, NSChains,
  WNSChains, ENSChains, WENSChains: ChainReshapeControl;
```

The "variables" above are constants for the following reshape control objects:

```
   NEW(ChainReshapeControl, chains := ChainSet{}),
   NEW(ChainReshapeControl, chains := ChainSet{Ch.W}),

   ...

   NEW(ChainReshapeControl,
       chains := ChainSet{Ch.W,Ch.E,Ch.N,Ch.S})
```

```
PROCEDURE ChainedReshape(
  self: ChainReshapeControl;
  ch: VBT.T;
  READONLY oldParentDomain, newParentDomain,
    oldChildDomain: Rect.T): Rect.T;
```

Return the rectangle that results from chaining each edge in `self.chains` *to the corresponding edge of the parent domain, and leaving the other edges unconstrained.*

If both edges in a dimension are chained, the offset and extent of the child will both vary to satisfy the chain constraints; if one edge is chained, the offset will vary and the extent will be fixed; if both edges are unchained, the offset and the extent will both be fixed.

For example, the default behavior for the initial background child is `WENSChains`, and the default behavior for all other children is `WNChains`.

One final reshape control method is sometimes useful:

```
PROCEDURE ScaledReshape(
  self: ReshapeControl;
  ch: VBT.T;
  READONLY oldParentDomain, newParentDomain,
    oldChildDomain: Rect.T) : Rect.T;
```

Return the integer approximation to the rectangle that results from scaling the old child domain to occupy the same relative position of the changing parent domain.

```
VAR (*CONST*) Scaled: ReshapeControl;
```

This "variable" is really a constant for the following reshape control object:

```
  NEW(ReshapeControl, apply := ScaledReshape)


END ZSplit.
```

4.3 The HVSplit interface

An `HVSplit.T` is a parent window that splits its screen into a row or column of child windows, depending on the *axis* of the split.

If the axis is horizontal, `Split.Succ` enumerates the children from west to east; if the axis is vertical, it enumerates them from north to south.

An `HVSplit` can be *adjustable* or *unadjustable*, a property that affects the way its space is divided between its children.

The *size* of a child is the extent of its domain in the axis of split, the *cross-size* is its extent in the other axis. For example, for a vertical split, a child's size is its height and its cross-size is its width.

The children of an `HVSplit` all have the same cross-size as the parent. To determine the sizes of the children, the `HVSplit` begins by computing the range of desirable sizes and the preferred size for each child by calling its shape method, passing the method the cross-size, so that, for example, the height of a child of a vertical split can depend on its width. At this point there are several cases.

If the sum of the minimum sizes of the children is greater than the size of the parent, then the split is said to be *overfull*. In this case the children are considered in order and given their minimum sizes, as long as there is room. The first child that doesn't fit is given all the space that's left, and the remaining children are given size zero.

If the split is not overfull, then the children are stretched according to the TeX model of boxes and glue. The details depend on whether the split is adjustable or unadjustable. For an adjustable split, each child's *stretchability* is its maximum desirable size minus its current size, and its *shrinkability* is its current size minus its minimum desirable size. If the size of the parent is increased by some amount `X`, then the sizes of the children are increased by amounts that total to `X` and are proportional to the children's stretchabilities. Similarly, if the size of the parent is decreased by some amount `X`, then the sizes of the children are decreased by amounts that total to `X` and are proportional to the children's shrinkabilities.

For a non-adjustable split, all the children's sizes are first set to their preferred sizes, and then they are stretched or shrunk the same as an adjustable split. Thus for a non-adjustable split each redistribution of space depends only on the children's shape methods, not on their current sizes.

A non-adjustable split is best if the layout can be controlled purely by stretchability and shrinkability. If the layout is also changed under user or program control, an adjustable split is required. For example, in a column of editable text windows, you should make the vertical split adjustable, since if the user makes one window big, and then the parent changes size slightly, you do not want the big window child to snap back to being small. On the other hand if you are using a horizontal split to center a `ButtonVBT` between two stretchy `TextureVBT`s, you should make it unadjustable, since in this case you always want to compute the division of space from the children's original shapes.

If the sum of the maximum sizes of the children is less than the size of the parent, the split is said to be *underfull*. There are no special rules for the underfull case: the TeX stretching algorithm is used without change. This produces a state in which the children are stretched larger than their maximum sizes.

A split is *infeasible* if it is overfull or underfull, and *feasible* otherwise.

The shape of an `HVSplit` is computed as follows: its maximum, minimum, and preferred sizes are obtained by adding up the corresponding values of its children. The cross-size range is the intersection of the cross-size ranges of its children (if this intersection is empty, the children's maximum cross-sizes are increased until the intersection is non-empty). The preferred cross-size of `v` is the maximum of the preferred cross-sizes of its children, projected into `v`'s cross-size range.

```
INTERFACE HVSplit;

IMPORT VBT, Split, Axis, Rect, Interval;

TYPE
  T <: Public;
  Private <: Split.T;
```

```
Public = Private OBJECT METHODS
  <* LL.sup <= VBT.mu *>
  init(hv: Axis.T;
    saveBits := FALSE;
    parlim := -1;
    adjustable := TRUE): T
END;
```

The call `v.init(...)` initializes `v` as an `HVSplit` with axis `hv` and no children.

If `saveBits` is `TRUE`, the implementation will try to save the children's old bits when reshaping; if the children don't use them anyway, it is faster to let `saveBits` default to `FALSE`. The value of `parlim` is the minimum area of a child for which a separate thread will be forked to `reshape` or `repaint`; if it is `-1`, it is set to an appropriate default (see the `VBTTuning` interface).

```
PROCEDURE New(
    hv: Axis.T;
    saveBits := FALSE;
    parlim := -1;
    adjustable := TRUE): T;
<* LL.sup <= VBT.mu *>
```

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

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

Return the axis of `v`.

4.3.1 Inserting children

See the `Split` interface to insert and reorder children.

```
PROCEDURE Cons(
    hv: Axis.T;
    ch0, ch1, ch2, ch3, ch4,
      ch5, ch6, ch7, ch8, ch9: VBT.T := NIL;
    saveBits := FALSE;
    parlim := -1;
    adjustable := TRUE): T; <* LL.sup = VBT.mu *>
```

Create an `HVSplit` with axis `hv` and children `ch0, ch1, ....`

`Cons` is equivalent to the following:

```
result := New(hv, saveBits, parlim, adjustable);
Split.AddChild(result, ch0, ch1, ..., ch9);
RETURN result
```

```
PROCEDURE ConsArray(
    hv: Axis.T;
    READONLY ch: ARRAY OF VBT.T;
    saveBits := FALSE;
    parlim := -1;
    adjustable := TRUE): T; <* LL.sup = VBT.mu *>
```

Create an `HVSplit` *with axis* `hv` *and children* `ch[0], ch[1], ....`

`ConsArray` ignores any `NIL`s in the array `ch`. It is equivalent to:

```
VAR result := New(hv, saveBits, parlim, adjustable);
BEGIN
  Split.AddChildArray(result, ch);
  RETURN result
END
```

4.3.2 Adjusting the division of space

The *division point after a child* is the sum of the sizes of all children up to and including the child.

```
PROCEDURE Adjust(v: T; ch: VBT.T; totsz: INTEGER)
RAISES {Split.NotAChild}; <* LL.sup = VBT.mu *>
```

Change the sizes of the children of `v` *so that the division point after* `ch` *is as close to* `totsz` *as possible, and mark* `v` *for redisplay.*

`Adjust` respects the size constraints on the children, and resizes children near the division point in preference to children far from the division point. For example, a sufficiently small adjustment will be made by resizing only `ch` and its successor. An adjustment large enough to make one of these children reach its max or min size will also resize the neighbor of that child, and so forth.

`Adjust` is a no-op if the split is infeasible or non-adjustable.

```
PROCEDURE FeasibleRange(v: T; ch: VBT.T): Interval.T
RAISES {Split.NotAChild}; <* LL.sup = VBT.mu *>
```

Return the interval of feasible positions for the division point after `ch`*.*

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

Return the largest size of a child that can be inserted into `v` *without making* `v` *infeasible.*

If the split is infeasible, `AvailSize` returns 0 and `FeasibleRange` returns the empty interval. Both procedures assume the total size available is the total of all child sizes.

```
END HVSplit.
```

4.4 The PackSplit interface

A `PackSplit.T` is a parent window whose children are packed into multiple rows or columns, depending on the *axis* of the split.

If the axis is horizontal, the children are packed into rows from west to east, moving south to a new row when the current row fills up. This is the normal style used in placing words in a paragraph.

If the axis is vertical, the children are packed into columns from north to south, moving east to a new column when the current column fills up. This is the normal style used in placing paragraphs in a newspaper article.

A `PackSplit` always gives its children their preferred height and width, even if this makes them extend outside the parent domain (in which case they will be clipped).

If the axis is horizontal, the children in any given row have their north edges aligned, and all children that are first in their row have their west edges aligned with the west edge of the parent. A child will be horizontally clipped if its requested horizontal size exceeds the parent's horizontal size; in this case the child will be alone in its row.

If the axis is vertical, the children in any given column have their west edges aligned, and all children that are first in their column have their north edge aligned with the north edge of the parent. A child will be vertically clipped if its requested vertical size exceeds the parent's vertical size; in this case the child will be alone in its column.

The *size* of a window is the extent of its domain in the axis of the `PackSplit`; its *cross-size* is its extent in the other axis.

The range of desirable sizes and the preferred size of a `PackSplit` are just the default for a `VBT`. The shape method uses the size to determine the cross-size that is just large enough to pack in all the children at their preferred sizes, and returns as its range of desirable cross-sizes a singleton interval containing only this cross-size.

```
INTERFACE PackSplit;

IMPORT VBT, PaintOp, Pixmap, Axis;

TYPE
  T <: Public;
  Private <: VBT.Split;
  Public = Private OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    init(hv := Axis.T.Hor;
      hgap, vgap := 1.5;
      txt: Pixmap.T := Pixmap.Solid;
      op: PaintOp.T := PaintOp.Bg;
      nwAlign := FALSE;
      saveBits := FALSE): T
  END;
```

The call `v.init(...)` initializes `v` as an empty packsplit with axis `hv`.

For a horizontal `PackSplit`, `hgap` is the gap to leave between children in each row; `vgap` is the gap to leave between rows. For a vertical `PackSplit`, `vgap` is the gap to leave between children in each column; `hgap` is the gap to leave between columns. The gaps are specified in millimeters.

The area not covered by children is painted using the painting operation `op` and the texture `txt+delta`, where `delta` is the origin unless `nwAlign` is set to `TRUE`, in which case `delta` will be set to the northwest corner of `v`.

If `saveBits` is `TRUE`, the implementation will try to save the children's old bits when reshaping; if the children don't use the old bits, it is more efficient to let `saveBits` default to `FALSE`.

```
PROCEDURE New(
  hv := Axis.T.Hor;
  hgap, vgap := 1.5;
  txt: Pixmap.T := Pixmap.Solid;
  op: PaintOp.T := PaintOp.Bg;
  nwAlign := FALSE;
  saveBits := FALSE): T; <* LL.sup <= VBT.mu *>
```

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

```
PROCEDURE Set(
  v: T;
  txt: Pixmap.T;
  op: PaintOp.T := PaintOp.BgFg;
  nwAlign := FALSE); <* LL.sup = VBT.mu *>
```

Change the texture displayed by `v` and mark `v` for redisplay.

```
PROCEDURE Get(
  v: T;
  VAR txt: Pixmap.T;
  VAR op: PaintOp.T;
  VAR nwAlign: BOOLEAN
  ); <* LL.sup = VBT.mu *>
```

Fetch the texture displayed by `v`.

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

Return the axis of `v`.

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

Return the `hgap` of `v`.

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

Return the `vgap` of `v`.

```
END PackSplit.
```

4.5 The TSplit interface

A `TSplit.T` is a parent window that giving its entire screen to one child at a time. The child being displayed is called the *current child*. The current child can be `NIL`, in which case the `TSplit` ignores all events.

```
INTERFACE TSplit;

IMPORT VBT, Split;

TYPE
  T <: Public;
  Private <: Split.T;
  Public = Private OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    init(fickle := TRUE): T
  END;
```

The call `v.init(fickle)` initialize `v` as an empty `TSplit`.

If `fickle` is `TRUE`, then the shape of `v` will be the shape of its current child, or a `VBT`'s default shape if the current child is `NIL`. If `fickle` is `FALSE`, then in each axis the size range of `v` will be the intersection of the size ranges of its children (if this intersection is empty, the children's maxsizes are increased until the intersection is non-empty). The preferred size of `v` is the the maximum of the preferred sizes of its children, projected into `v`'s size range. If `v` has no children, its shape is a `VBT`'s default shape.

```
PROCEDURE SetCurrent(v: T; ch: VBT.T)
RAISES {Split.NotAChild}; <* LL.sup = VBT.mu *>
```

Set the current child of `v` to be `ch` and mark `v` for redisplay.

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

Return the current child of `v`.

```
PROCEDURE Cons(ch0, ch1, ch2, ch3, ch4: VBT.T := NIL;
  fickle := TRUE): T; <* LL.sup = VBT.mu *>
```

Create a `TSplit` with children `ch0, ch1, ....`

`Cons` is equivalent to

```
    v := NEW(T).init(fickle);
    Split.AddChild(v, ch0, ch1, ch2, ch3, ch4);
    IF ch0 # NIL THEN SetCurrent(v, ch0) END;
    RETURN v


END TSplit.
```

5 Filters

5.1 The Filter interface

A `Filter.T` is a `Split.T` with at most one child.

```
INTERFACE Filter;

IMPORT Split, VBT;

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

The call `v.init(ch)` initializes `v` as a filter with child `ch` and returns `v`.

`Split.Move` on a filter is a noop. `Split.Insert` replaces the child, if any, and detaches it.

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

Return the child of `v`, or `NIL` if there is no child.

`Filter.Child(v)` is equivalent to `Split.Succ(v, NIL)`.

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

Replace `v`'s child by `ch`, detach and return `v`'s old child, and mark `v` for redisplay.

`Filter.Replace` is similar to `Split.Replace`, except that it returns the old child instead of taking the old child as an argument, and if `ch` is `NIL` it is similar to `Split.Delete`.

```
END Filter.
```

5.2 The BorderedVBT interface

A `BorderedVBT.T` is a filter whose parent's screen consists of the child's screen surrounded by a border. The parent's shape is determined from the child's shape by adding the border size.

```
INTERFACE BorderedVBT;

IMPORT VBT, Filter, PaintOp, Pixmap;
```

```
TYPE
  T <: Public;
  Public = Filter.T OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    init(ch: VBT.T;
      size: REAL := Default;
      op: PaintOp.T := PaintOp.BgFg;
      txt: Pixmap.T := Pixmap.Solid): T
  END;
```

The call `v.init(...)` initializes `v` as a `BorderedVBT` with child `ch` and marks `v` for redisplay.

The border size is given in millimeters. The border will be painted in the untranslated texture `txt` using the paint op `op`.

```
CONST Default = 0.5;

PROCEDURE New(
    ch: VBT.T;
    size: REAL := Default;
    op: PaintOp.T := PaintOp.BgFg;
    txt: Pixmap.T := Pixmap.Solid)
    : T; <* LL.sup <= VBT.mu *>
```

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

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

Change the size of the border of `v` to `newSize` millimeters and mark `v` for redisplay.

```
PROCEDURE SetColor(
  v: T;
  op: PaintOp.T;
  txt := Pixmap.Solid);
<* LL.sup = VBT.mu *>
```

Change the `op` and `texture` of `v` and mark `v` for redisplay.

```
PROCEDURE Get(
  v: T;
  VAR size: REAL;
  VAR op: PaintOp.T;
  VAR txt: Pixmap.T); <* LL.sup = VBT.mu *>
```

Fetch `v`'s parameters.

```
END BorderedVBT.
```

5.3 The RigidVBT interface

A `RigidVBT.T` is a filter whose size range is set explicitly, independently of its child's size range. In spite of its name, it size range does not have to be fixed to a single value.

All dimensions in this interface are specified in millimeters.

```
INTERFACE RigidVBT;

IMPORT VBT, Filter, Axis;

TYPE
  T <: Public;
  Public = Filter.T OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    init(ch: VBT.T; sh: Shape): T
  END;

TYPE
  SizeRange = RECORD lo, pref, hi: REAL END;
  Shape = ARRAY Axis.T OF SizeRange;
```

The call `v.init(...)` initializes `v` as a rigid `VBT` with child `ch` and shape `sh`.

A `RigidVBT.SizeRange` is like a `VBT.SizeRange`, but in millimeters instead of pixels, using `REAL`s instead of `INTEGER`s, and the range is `[lo..hi]` instead of `[lo..hi-1]`.

```
PROCEDURE New(ch: VBT.T; sh: Shape): T;
```

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

```
PROCEDURE FromHV(
  ch: VBT.T;
  hMin, vMin: REAL;
  hMax, vMax, hPref, vPref: REAL := -1.0) : T;
  <* LL.sup <= VBT.mu *>
```

Return a `RigidVBT` with child `ch` and the given shape.

If `hMax` or `hPref` are defaulted, they are assumed to be the same as `hMin`, and similarly for `vMax`, `vPref` and `vMin`. That is, `FromHV` is equivalent to:

```
IF hMax = -1.0 THEN hMax := hMin END;
IF vMax = -1.0 THEN vMax := vMin END;
IF hPref = -1.0 THEN hPref := hMin END;
IF vPref = -1.0 THEN vPref := vMin END;
RETURN New(ch,
  Shape{SizeRange{h, hMax, hPref},
        SizeRange{v, vMax, vPref}})

END RigidVBT.
```

5.4 The HighlightVBT interface

A `HighlightVBT.T` is a filter that highlights a rectangular outline over its child.

The parent screen is obtained from the child screen by texturing an outline inset in a rectangle, using an inverting painting operation.

The parent keeps its screen correct as the child paints. Since the parent screen is always correct, it never needs to mark itself for redisplay.

```
INTERFACE HighlightVBT;

IMPORT VBT, Rect, Point, Filter, Pixmap, PaintOp;

TYPE
  T <: Public;
  Public = Filter.T OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    init(ch: VBT.T;
      op: PaintOp.T := PaintOp.TransparentSwap;
      txt: Pixmap.T := Pixmap.Gray;
      READONLY delta := Point.T{h := 0, v := 1}): T
  END;
```

The call `v.init(ch, ...)` initializes `v` as a `HighlightVBT` with child `ch` and the given parameters, and returns `v`.

The highlight rectangle is initially empty. The filter brings up the highlight by calling

```
VBT.PaintTexture(v, highlight region, op, txt, delta)
```

and brings down the highlight the same way; therefore the painting operation must be its own inverse for the filter to work correctly.

The default values for the texture and delta are such that the highlight will be visible over white, black, or the standard gray texture. (If delta were `(0,0)` instead of `(0,1)`, the highlight would look fine over white or black but would be barely noticeable over standard gray.)

```
PROCEDURE New(
  ch: VBT.T;
  op := PaintOp.TransparentSwap;
  txt: Pixmap.T := Pixmap.Gray;
  READONLY delta := Point.T{h := 0, v := 1}) : T ;
<* LL.sup <= VBT.mu *>
```

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

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

Return the lowest (possibly improper) ancestor of `v` that is a `HighlightVBT.T` or `NIL` if there isn't one.

```
PROCEDURE SetRect(
  v: VBT.T;
  READONLY rect: Rect.T;
  inset: CARDINAL := 2);
 <* LL.sup = VBT.mu *>
```

Set the rectangle and inset of `Find(v)` to the given values.

The inset is given in pixels, not in millimeters.

```
PROCEDURE SetTexture(
  v: VBT.T;
  txt: Pixmap.T;
  READONLY delta := Point.Origin;
  op := PaintOp.TransparentSwap);
 <* LL.sup = VBT.mu *>
```

Set the `txt`, `delta`, and `op` of `Find(v)` to the given values.

```
PROCEDURE Get(
  v: VBT.T;
  VAR rect: Rect.T;
  VAR inset: CARDINAL;
  VAR txt: Pixmap.T;
  VAR delta: Point.T;
  VAR op: PaintOp.T): BOOLEAN; <* LL.sup = VBT.mu *>
```

Fetch the parameters for the `HighlightVBT` above `v`, and return `TRUE`. If `v` has no such ancestor, return `FALSE`.

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

Highlight the outline inset into the rectangle `r` with width `inset`, using a solid texture.

`Invert` operates on `Find(v)`. It is equivalent to:

```
SetTexture(v, Pixmap.Solid);
SetRect(v, r, inset)
```

`SetRect`, `SetTexture`, and `Invert` are no-ops if `Find(v)` is `NIL`.

```
END HighlightVBT.
```

5.5 The TranslateVBT interface

A `TranslateVBT.T` is a filter that maintains a translation between the coordinate systems of the child and parent such that the child's coordinate system has its origin

at the northwest corner of the child domain. The child can be `NIL`, in which case the `TranslateVBT` ignores all events.

```
INTERFACE TranslateVBT;

IMPORT VBT, Filter;

TYPE T <: Filter.T;
```

The call `v.init(ch)` initializes `v` as a `TranslateVBT` with child `ch`.

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

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

```
END TranslateVBT.
```

5.6 Buttons

A `ButtonVBT.T` is a filter with an associated action procedure that is called when the user clicks on the button or makes some other appropriate gesture.

Different subtypes of `ButtonVBTs` invoke the action procedure on different user gestures, but all `ButtonVBTs` have the three methods `pre`, `post`, and `cancel`. They all interpret user gestures in such a way that the sequence of calls will be in the regular expression

```
( (pre cancel) | (pre action post) )*
```

The minimum, maximum, and preferred size of a `ButtonVBT` are all equal to the minimum size of its child, in each axis.

```
INTERFACE ButtonVBT;

IMPORT VBT, Filter, PackSplit, PaintOp;

TYPE
  T <: Public;
  Public = Filter.T OBJECT (*CONST*)
    action: Proc
  METHODS
    <* LL.sup = VBT.mu *>
    pre();
    post();
    cancel();
    <* LL.sup <= VBT.mu *>
    init(ch: VBT.T;
      action: Proc;
      ref: REFANY := NIL): T;
  END;
```

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

The call `v.init(...)` initializes `v` with child `ch` and action proc `action` and adds `ref` to `v`'s property set if it is not `NIL`. The action procedure can access `ref` (if it is not `NIL`) by calling `VBT.GetProp`.

The mouse and position methods of a `ButtonVBT.T` call the `pre` method on a down click, and then call the `cancel` method if the user chords by clicking another mouse button or if the user moves the mouse out of the button. Otherwise they call the action procedure `proc` if the user releases the mouse button.

The default `pre` method highlights the button, the default `post` and `cancel` methods unhighlight it. Consequently there should be a `HighlightVBT` somewhere above the button. Since `Trestle.Install` automatically inserts a `HighlightVBT`, you usually don't have to worry about this.

The action procedure is a field rather than a method in order to allow buttons with different action procedures to share their method suites.

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

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

```
PROCEDURE MenuBar(
  ch0, ch1, ch2, ch3, ch4, ch5,
    ch6, ch7, ch8, ch9: VBT.T := NIL;
  op: PaintOp.T := PaintOp.Bg)
  : PackSplit.T; <* LL.sup = VBT.mu *>
```

Return a `PackSplit` with the given children, left-justified, and with its background painted with `op`.

`MenuBar` is convenient for building a horizontal row of buttons. If the row fills up, the extra buttons will wrap to the next line.

```
END ButtonVBT.
```

5.7 Quick buttons

A `QuickBtnVBT.T` is a button that activates immediately on down-clicks. Quick buttons are useful for boolean toggles and radio buttons.

A `QuickBtnVBT` has its `pre`, `action`, and `post` methods called on every mouse click of type `FirstDown` in its domain. Its `cancel` method is never called. Its default `pre` and `post` methods are no-ops.

```
INTERFACE QuickBtnVBT;

IMPORT ButtonVBT, VBT;

TYPE T <: ButtonVBT.T;
```

The call `v.init(ch, action, ref)` initializes `v` as a quick button with child `ch` and action procedure `action`, and adds `ref` to `v`'s property set if it is not `NIL`.

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

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

```
END QuickBtnVBT.
```

5.8 Menu Buttons

A `MenuBtnVBT.T` is a button suitable for the items in pop-up and pull-down menus.

When the cursor rolls into a menu button, the `pre` method is called and the button is *readied*. If it receives a mouse transition of type `LastUp` while it is readied, the `action` and `post` methods are called. The `cancel` method is called if the cursor leaves the button or the user chords with the mouse while the button is readied .

```
INTERFACE MenuBtnVBT;

IMPORT ButtonVBT, VBT;

TYPE T <: ButtonVBT.T;
```

The call `v.init(ch, action, ref)` initializes `v` as a menu button with child `ch` and action procedure `action`, and adds `ref` to `v`'s property set if it is not `NIL`.

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

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

```
PROCEDURE TextItem(
  name: TEXT; action: ButtonVBT.Proc;
  ref: REFANY := NIL): T; <* LL.sup = VBT.mu *>
```

Return a menu button that displays the text `name`.

`TextItem` is a convenience procedure for making a menu button with a `TextVBT` child. The borders are initialized to make the button suitable for stacking into a menu using a vertical `HVSplit`. More precisely, `TextItem` is equivalent to:

```
        New(TextVBT.New(name, 0.0, 0.5, 3.0, 0.5),
            action, ref)

    END MenuBtnVBT.
```

5.9 Anchor Buttons

An `AnchorBtnVBT.T` is a button that activates a pull-down menu when you click on it or roll into it from another anchor button.

Associated with each anchor button `b` is

- `b.menu`, the menu to be activated,
- `b.hfudge` and `b.vfudge`, dimensions in millimeters that control where the menu is popped up,
- `b.n`, a count of the number of `ZSplit` ancestors of `b` to skip when looking for the `ZSplit` to insert the menu into.

A down click on an anchor button `b` *activates* it by:

- calling the method `b.pre()`, and then
- inserting the window `b.menu` so that its northwest corner is `b.hfudge` millimeters to the right and `b.vfudge` millimeters below the southwest corner of `b`. The menu will be inserted into the (`b.n`)th `ZSplit` ancestor of `b` (counting the first `ZSplit` ancestor as zero), or as an undecorated top-level window if `b` has at most `b.n` `ZSplit` ancestors.

The anchor button will be deactivated when it gets another mouse transition or when the user rolls the mouse over a sibling anchor button, in which case the sibling will be activated. Two anchor buttons are siblings if they have the same "anchor parent". The anchor parent is specified when the anchor button is created; if it is `NIL`, then the normal parent is used as the anchor parent. When an anchor button is deactivated, its cancel method is called and its menu is deleted from its `ZSplit`.

The default `pre` method highlights the anchor button; the default `cancel` method unhighlights it.

In the common case in which the user down-clicks on the anchor, rolls over the menu, and up-clicks on one of the items, the upclick will be delivered to the item first, which will invoke the appropriate action, and then will be delivered to the anchor button (since the anchor button has the mouse focus), which will delete the menu.

A `HighlightVBT` is automatically inserted over the menu when it is inserted, and discarded when the menu is deleted. This allows the menu items to highlight themselves without interfering with the highlighting of the anchor button.

The `action` procedure and `post` method of an anchor button are never called. The `pre` and `cancel` methods can be overridden; for example, the `pre` method could

prepare the menu before it is inserted. This is the reason the menu field is revealed in the type declaration.

The same menu can be associated with several anchor buttons, provided that only one of them is active at a time.

```
INTERFACE AnchorBtnVBT;

IMPORT ButtonVBT, VBT, ZSplit, Point;

TYPE
  T <: Public;
  Public = ButtonVBT.T OBJECT
    menu: VBT.T
  METHODS <* LL.sup <= VBT.mu *>
    init(ch: VBT.T;
      menu: VBT.T;
      n: CARDINAL := 0;
      anchorParent: VBT.T := NIL;
      hfudge, vfudge := 0.0;
      ref: REFANY := NIL): T
  END;
```

The call `v.init(...)` initializes the button with the given attributes, and adds `ref` to `v`'s property set if it is not `NIL`. This includes a call to `ButtonVBT.T.init(v, ch)`.

You must not change the menu while the `AnchorBtnVBT` is active.

```
PROCEDURE New(
  ch: VBT.T;
  menu: VBT.T;
  n: CARDINAL := 0;
  anchorParent: VBT.T := NIL;
  hfudge, vfudge := 0.0;
  ref: REFANY := NIL): T; <* LL.sup <= VBT.mu *>
```

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

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

Set the anchor parent of `v` to be `p`. If `v` is active, this is a checked runtime error.

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

Return the anchor parent of `v`.

```
PROCEDURE Set(v: T;  n: CARDINAL; hfudge, vfudge: REAL);
<* LL.sup = VBT.mu *>
```

Set the attributes of `v`. If `v` is active, this is a checked runtime error.

```
PROCEDURE Get(v: T; VAR n: CARDINAL;
  VAR hfudge, vfudge: REAL); <* LL.sup = VBT.mu *>
```

Fetch the attributes of `v`.

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

Return `TRUE` *if and only if* `v` *is active.*

```
END AnchorBtnVBT.
```

6 Some useful Leaf VBTs

6.1 The TextVBT interface

A `TextVBT.T` is a `VBT` that displays a text string.

The minimum size of a `TextVBT` is just large enough to display its text (surrounded by any margins that were supplied when the `TextVBT` was created), except that if its text is empty its minimum size is just large enough to display the text "`X`". Its preferred size is the same as its minimum size, and its maximum size is very large.

```
INTERFACE TextVBT;

IMPORT VBT, Font, PaintOp, Rect;

TYPE
  T <: Public;
  Public = VBT.Leaf OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    init(txt: TEXT;
      halign, valign: REAL := 0.5;
      hmargin: REAL := 0.5;
      vmargin: REAL := 0.0;
      fnt: Font.T := Font.BuiltIn;
      bgFg: PaintOp.ColorQuad := NIL): T
  END;
```

The call `v.init(...)` initializes `v` as a `TextVBT` that displays the text `txt` in the font `fnt`, and returns `v`.

The text will be painted with `bgFg`'s foreground; the background will be painted with `bgFg`'s background. If `bgFg` is `NIL` these default to `PaintOp.Fg` and `PaintOp.Bg`. The text should not contain any newline characters: it will be treated as a single line. If `halign = 0.0`, the west boundary of the text will be indented by the given `hmargin` (in millimeters) from the west boundary of the `VBT`; if `halign = 1.0`, the east boundary of the text will be inside the east boundary of the `VBT` by the given `hmargin`; for other values of `halign`, the horizontal position of the text is computed by linear interpolation. In particular, `halign = 0.5` centers the text horizontally. The vertical position is determined by `vmargin` and `valign` in a similar way.

Control-left-click in the text sets the source selection to be a readonly version of the text. Thus you can copy the text out of any `TextVBT`.

```
PROCEDURE New(
    txt: TEXT;
    halign, valign: REAL := 0.5;
    hmargin: REAL := 0.5;
    vmargin: REAL := 0.0;
    fnt: Font.T := Font.BuiltIn;
```

```
    bgFg: PaintOp.ColorQuad := NIL) : T;
    <* LL.sup <= VBT.mu *>
```

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

```
PROCEDURE Put(v: T; txt: TEXT); <* LL.sup < v *>
```

Change the text displayed by v to be txt and mark v for redisplay.

```
PROCEDURE Get(v: T): TEXT; <* LL.sup < v *>
```

Return the text displayed by v.

```
PROCEDURE SetFont(
  v: T;
  fnt: Font.T;
  bgFg : PaintOp.ColorQuad := NIL);
<* LL.sup = VBT.mu *>
```

Set v's font and bgFg to the given values and mark v for redisplay. If bgFg is defaulted, PaintOp.bgFg is used.

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

Return v's font.

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

Return v's color quad.

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

Return the current bounding rectangle of v's text.

```
END TextVBT.
```

6.2 The TextureVBT interface

A `TextureVBT.T` is a `VBT` that displays a texture, possibly colored. Its preferred and minimum sizes are zero and its maximum size is very large, in each axis.

```
INTERFACE TextureVBT;

IMPORT VBT, PaintOp, Pixmap;

TYPE
  T <: Public;
  Public = VBT.Leaf OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    init(op: PaintOp.T := PaintOp.BgFg;
```

```
      txt: Pixmap.T := Pixmap.Solid;
      nwAlign: BOOLEAN := FALSE): T
  END;
```

The call `v.init(...)` initializes `v` as a `TextureVBT` displaying `txt` with the painting operation `op`.

The domain of `v` will be painted using the painting operation `op` and the texture `txt+delta`, where `delta` is the origin unless `nwAlign` is set to `TRUE`, in which case `delta` will be set to the northwest corner of `v`.

```
PROCEDURE New(
  op: PaintOp.T := PaintOp.BgFg;
  txt: Pixmap.T := Pixmap.Solid;
  nwAlign: BOOLEAN := FALSE): T;  <* LL.sup <= VBT.mu *>
```

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

```
PROCEDURE Set(
  v: T;
  op: PaintOp.T := PaintOp.BgFg;
  txt: Pixmap.T := Pixmap.Solid;
  nwAlign: BOOLEAN := FALSE);
<* LL.sup = VBT.mu *>
```

Change `v`'s texture and mark it for redisplay.

```
PROCEDURE Get(
  v: T;
  VAR op: PaintOp.T;
  VAR txt: Pixmap.T;
  VAR nwAlign: BOOLEAN); <* LL.sup = VBT.mu *>
```

Fetch `v`'s texture.

```
END TextureVBT.
```

6.3 The HVBar interface

An `HVBar.T` is an adjustable bar that allows a user to adjust the division of space between the children of an `HVSplit`.

An `HVBar` must be a child of an `HVSplit`. When the user pushes a mouse button over the bar, the cursor changes shape and the outline of the bar is highlighted. The highlight follows the cursor as long as the button is down. When the button comes up, the bar calls `HVSplit.Adjust` to move the bar to the currently highlighted position. If the user tries to move the bar outside the range of positions that are consistent with the size constraints of the children of the parent `HVSplit`, the highlighted bar will not follow the cursor. If the user chords while dragging, then adjusting mode is cancelled.

The bar has methods that you can override that are called each time the bar is moved, or continuously during adjustment.

In order for the bar to highlight correctly, some ancestor of the `HVSplit` on which it is installed must be a `HighlightVBT`. Since `Trestle.Install` automatically inserts a `HighlightVBT` over top-level windows, you usually don't have to worry about this.

```
INTERFACE HVBar;

IMPORT VBT, PaintOp, Pixmap, TextureVBT;

TYPE
  T <: Public;
  Public = TextureVBT.T OBJECT METHODS
    <* LL = VBT.mu *>
    pre(READONLY cd: VBT.MouseRec);
    post(READONLY cd: VBT.MouseRec);
    during(n: INTEGER);
    <* LL <= VBT.mu *>
    init(size: REAL := DefaultSize;
      op: PaintOp.T := PaintOp.BgFg;
      txt: Pixmap.T := Pixmap.Gray): T
  END;
```

The call `v.init(...)` initializes `v` as an `HVBar` with the given properties and returns `v`. This includes calling `TextureVBT.T.init(v, op, txt)`.

The argument `size` gives the number of millimeters that the bar will occupy in the parent `HVSplit`.

An adjusting bar `b` calls `b.pre(cd)` when it begins adjusting in response to a mouse click `cd`. It calls `b.during(k)` each time the mouse moves during dragging, where `k` is the coordinate that the `lo` (i.e., west or north) edge of the bar would move to if dragging were stopped at that instant. Finally, the bar calls `b.post(cd)` when it stops adjusting in response to an upclick or chord `cd`. The `HVSplit` will be adjusted (but not redisplayed) before `b.post(cd)` is called.

The default `pre` and `during` methods highlight the position the bar would move to if dragging were stopped. The default `post` method removes the highlighting.

```
CONST
  DefaultSize = 2.5;

PROCEDURE New(
    size := DefaultSize;
    op := PaintOp.BgFg;
    txt := Pixmap.Gray): T; <* LL.sup <= VBT.mu *>
```

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

```
END HVBar.
```

7 Resources

In this section we introduce resources (painting operations, cursors, pixmaps, and fonts). We will introduce the screen-independent forms first, then the screen-dependent forms.

A screen-independent resource is represented as an integer, which is simply an index into a table called a "palette". Unless you are re-implementing Trestle over a new window system, you can ignore the palette and treat the integers as opaque values that serve only to distinguish one resource from another. To prevent one kind of screen-independent resource from being confused with another, the integers are wrapped into one-componenet records.

A few screen-independent resources are *predefined*, which means that constant integers are assigned to them in the public interface. Each interface that defines a screen-independent resource declares a subrange type `Predefined` that contains the integers that are predefined. These types will be handy when we get to the screen-dependent resources; until then you can ignore them.

7.1 The PaintOp interface

A `PaintOp.T` is a screen-independent painting operation.

A painting operation `op` takes a source pixel `s` and a destination pixel `d` and produces a new value `op(d, s)` for the destination pixel.

A painting operation that ignores the source pixel is called a *tint*. If `op` is a tint, we just write `op(d)` instead of `op(d, s)`. If the effect of a tint is to set the destination pixel to some fixed value independent of its initial value, then the tint is said to be *opaque*.

The locking level is `LL.sup <= VBT.mu` for all of the procedures in this interface.

```
INTERFACE PaintOp;

TYPE
  T = RECORD op:INTEGER END; Predefined = [0..16];

CONST
  Bg = T{0};
  Fg = T{1};
  Transparent = T{2};
  Swap = T{3};

  Copy = T{4};
```

`Bg`, `Fg`, `Transparent`, and `Swap` are Trestle's four basic tints.

`Bg` sets the destination pixel to the screen's background color; `Fg` sets it to the screen's foreground color; `Transparent` is the identity function; `Swap` is a self-inverting operation that exchanges the foreground and background pixels. More

precisely, consider a particular screentype and let `bgpix` and `fgpix` be the foreground and background pixel for that screentype. Then for any pixel `d`,

```
Bg(d) = bgpix
Fg(d) = fgpix

Transparent(d) = d

Swap(bgpix) = fgpix
Swap(fgpix) = bgpix
Swap(Swap(d)) = d
Swap(d) # d
```

The operation `Copy` copies source to destination:

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

`Copy` is not a tint, and should be used only when the source pixels are of the same screentype as the destination pixels (for example, with `VBT.Scroll`, or when painting a pixmap of the same type as the screen).

```
CONST
  BgBg = Bg;
  BgFg = T{5};
  BgTransparent = T{6};
  BgSwap = T{7};

  FgFg = Fg;
  FgBg = T{8};
  FgTransparent = T{9};
  FgSwap = T{10};

  TransparentTransparent = Transparent;
  TransparentBg = T{11};
  TransparentFg = T{12};
  TransparentSwap = T{13};

  SwapSwap = Swap;
  SwapBg = T{14};
  SwapFg = T{15};
  SwapTransparent = T{16};
```

The sixteen operations above all have names of the form `XY`, where `X` and `Y` are one of the four basic tints. They are defined by the rule:

```
XY(dest, source) =
  IF source = 0 THEN X(dest) ELSE Y(dest) END
```

For example, `BgFg` can be used to paint a one bit deep source interpreting zeros as background and ones as foreground.

Obviously these sixteen painting operations should be used only with one-bit deep sources. However, not all one-bit deep sources are of the same screentype: for example, different screentypes might have different rules for representing bitmaps. To accomodate this unfortunate fact of life, we associate with every screentype `st` another screentype `st.bits`, which is the type of bitmap sources appropriate for `st`. The depth of `st.bits` is always one. If the depth of `st` is one, then it is possible (but not certain) that `st.bits = st`. When using one of sixteen operations above on a `VBT` with screentype `st`, the source must have type `st.bits`. You will be happy to recall that this will be taken care of automatically if you use screen-independent bitmaps and fonts.

Next there is a procedure for generating colored painting operations.

```
TYPE
  Mode = {Stable, Normal, Accurate};
  BW = {UseBg, UseFg, UseIntensity};

PROCEDURE FromRGB(
   r, g, b: REAL;
   mode := Mode.Normal;
   gray := -1.0;
   bw := BW.UseIntensity): T;
```

Return a tint that will set a pixel to the color `(r,g,b)`.

The values `r`, `g`, and `b` should be in the range `0.0` to `1.0`; they represent the fractions of red, green, and blue in the desired color.

The `gray` argument controls what the tint will do on a gray-scale display. If `gray` is between zero and one, it specifies the intensity of the tint. If `gray` is defaulted to `-1`, then the tint will use the intensity of the color `(r,g,b)`.

The `bw` argument controls what the tint will be on a monochrome display. If `bw` is `UseBg` or `UseFg`, then the tint will be `Bg` or `Fg`, respectively. If `bw` is `UseIntensity`, then the tint will be `Fg` if `r`, `g`, and `b` are all zero (that is, if the color is black), and `Bg` otherwise.

The `mode` argument is relevant on color and gray-scale displays. When the total number of pixel colors desired by all of the applications that are running exceeds the number of available colors, then some applications' colors will change (usually in an unpleasantly random way).

To reduce the likelihood that your color will change randomly (at the cost of fidelity), set `mode` to `Stable`. To increase the fidelity of the pixel to the specified intensities (at the cost of increased danger of random change), set `mode` to `Accurate`. For example, an icon window should use stable colors; a color editor should use accurate colors.

```
PROCEDURE Pair(op0, op1: T): T;
```

Return an operation `op` such that `op(d,0) = op0(d)` and `op(d,1) = op1(d)`.

For example,

```
Pair(FromRGB(1.0,1.0,1.0), FromRGB(1.0,0.0,0.0))
```

will paint a bitmap with zeros as white and ones as red.

```
PROCEDURE SwapPair(op0, op1: T): T;
```

Return an operation that swaps the pixels painted by `op0` and `op1`.

`SwapPair` requires that `op0` and `op1` be opaque, that is, they must set the destination to particular pixels (say, `pix0` and `pix1`). Then the tint `op` returned by `SwapPair` satisfies:

```
op(pix0) = pix1
op(pix1) = pix0
op(op(p)) = p  for any pixel p
```

For example, `Swap = SwapPair(Bg, Fg)`.

Sometimes it is handy to collect several related painting operations into a single object:

```
TYPE
  ColorQuad = OBJECT
    bg, fg, bgFg, transparentFg: T
  END;

PROCEDURE MakeColorQuad(bg, fg: T): ColorQuad;
```

Return `ColorQuad{bg,fg,Pair(bg,fg),Pair(Transparent,fg)}`.

```
TYPE
  ColorScheme = ColorQuad OBJECT
    swap, bgTransparent, bgSwap, fgBg, fgTransparent,
      fgSwap, transparentBg, transparentSwap,
      swapBg, swapFg, swapTransparent: T;
  END;

PROCEDURE MakeColorScheme(bg, fg: T): ColorScheme;
```

Return the fifteen painting operations other than `Transparent` that can be made by combining `bg`, `fg`, and `Transparent`, using `SwapPair` and `Pair`.

In `MakeColorQuad` and `MakeColorScheme`, `bg` and `fg` should be tints.

```
VAR (*CONST*) bgFg: ColorScheme;
```

This "variable" is really a constant for `MakeColorScheme(Bg, Fg)`.

```
END PaintOp.
```

7.2 The Cursor interface

A `Cursor.T` is a screen-independent specification of a cursor shape. The call `VBT.SetCursor(v, cs)` sets the cursor of `v` to be `cs`.

The locking level is `LL.sup <= VBT.mu` for all of the procedures in this interface.

```
INTERFACE Cursor;

IMPORT Pixmap, Point;

TYPE T = RECORD cs: INTEGER END; Predefined = [0..2];

CONST
  DontCare = T{0};
  TextPointer = T{1};
  NotReady = T{2};
```

You should set `Cursor.DontCare` when you don't care about the cursor shape; `Cursor.TextPointer` when the cursor is to be used for editing text, and `Cursor.NotReady` to indicate that the application is not receptive to user input.

```
TYPE Raw = RECORD
    plane1, plane2: Pixmap.Raw;
    hotspot: Point.T;
    color1, color2, color3: RGB;
  END;
  BW = {UseBg, UseFg, UseIntensity};
  RGB = RECORD
    r, g, b: REAL;
    gray := -1.0;
    bw := BW.UseIntensity
  END;
```

A `Raw` represents a cursor with explicit offset, bitmaps, and colors.

The `plane1` and `plane2` are depth-1 pixmaps. They must have the same bounding rectangle, and the hotspot must lie within the bounding rectangle or on its east or south edge. If the hotspot is illegal, it will be moved to the closest legal position.

The cursor's hotspot is kept on top of the mouse's location on the screen. The cursor's image tracks the mouse relative to the hotspot. For example, if the hotspot is (0, 0), the (0, 0) bit of the cursor's image will be located over the mouse's location. The remainder of the cursor will appear to the south and east.

The color of each pixel in the cursor's image is determined from the corresponding bits in `plane1` and `plane2` (`p1` and `p2`):

```
p1 = 0, p2 = 0  => transparent
p1 = 0, p2 = 1  => color1
p1 = 1, p2 = 0  => color2
p1 = 1, p2 = 1  => color3
```

The colors for the cursor are matched as closely as possible to the selection of cursor colors that the screentype supports. If the screentype allows only two colors for the cursor, then the pixels that would have been `color3` will be `color1`. The `gray` and `bw` values control the color on gray-scale and monochrome displays, according to the same rule used in `PaintOp.FromRGB`.

```
PROCEDURE FromRaw(READONLY r: Raw): T;
```

Return a cursor that looks like `r` on all screens.

If the screentype does not support `r`'s colors or size, `FromRaw` will clip or convert colors as necessary. On a screentype that does not allow user-defined cursors, the cursor returned by `FromRaw` will behave like `DontCare`.

```
PROCEDURE FromName(READONLY names: ARRAY OF TEXT): T;
```

Return the first available cursor of those named in the array `names`.

The entries of `names` are cursor names as specified in the `ScrnCursor` interface, possibly containing wild card characters. On any particular screentype, `FromName(names)` iterates through `names` in order and returns an arbitrary match from the first name that matches anything. If no name has any matches, it returns `DontCare`.

Standard X screentypes support the cursors named in *X Window System* by Scheifler et. al. [5] Appendix B. Therefore, for example,

```
FromName(ARRAY OF TEXT{"XC_Arrow"})
```

returns a cursor that behaves like the X arrow cursor on X screentypes, and like `DontCare` on screentypes that have no cursor named `XC_Arrow`.

```
END Cursor.
```

7.3 The Pixmap interface

A `Pixmap.T` is a screen-independent specification of a pixmap. Many procedures interpret `Pixmap.Ts` as textures, by tiling the plane with translated copies of the pixmap. There are three predefined pixmaps:

The locking level is `LL.sup <= VBT.mu` for all of the procedures in this interface.

```
INTERFACE Pixmap;

TYPE T = RECORD pm: INTEGER END; Predefined = [0..2];

CONST
  Solid = T{0};
  Empty = T{1};
  Gray = T{2};
```

`Solid` represents a pixmap of all ones. `Empty` represents a pixmap of all zeros. `Gray` represents a checkerboard of ones and zeros.

The domains of these pixmaps may vary from screentype to screentype, but they will always be non-empty.

When used on a screentype `st`, they will have type `st.bits` (see the `PaintOp` interface).

```
TYPE Raw <: ROOT;
```

A `Pixmap.Raw` represents a pixmap as a packed array of pixels. The `ScrnPixmap` interface reveals the representation.

```
PROCEDURE FromBitmap(bits: Raw): T;
```

Return a pixmap that looks like `bits` on all screens.

`FromBitmap` causes a checked runtime error if the depth of `bits` is not one. On a screentype `st`, it will have type `st.bits`.

```
END Pixmap.
```

7.4 The Font interface

A `Font.T` is a screen-independent specification of a typeface. There is one predefined `Font.T`, which yields the built-in font of the screentype.

The locking level is `LL.sup <= VBT.mu` for all of the procedures in this interface.

```
INTERFACE Font;
TYPE T = RECORD fnt: INTEGER END; Predefined = [0..0];
CONST BuiltIn = T{0};
PROCEDURE FromName(READONLY names: ARRAY OF TEXT): T;
```

Return the first available font of those named in the array `names`.

The entries of `names` are font names as specified in the `ScrnFont` interface, possibly containing wild card characters. On any particular screentype, `FromName(names)` iterates through `names` in order and returns an arbitrary match from the first name that matches anything. If no name has any matches, it returns the built-in font.

Standard X screentypes give fonts long "names" that encode their properties, so with X it is almost always desirable to include wild-card characters in the names. For example,

```
FromName(
 ARRAY OF TEXT{"-*-times-medium-r-*-*-*-10?-*"})
```

will return a font that, on an X server containing the standard fonts, is some Times Roman medium-weight unslanted font sized 10 to 10.9 points, and behaves like `Font.BuiltIn` on any screentye that doesn't have a font whose name matches the pattern.

```
END Font.
```

7.5 The Palette interface

The `Palette` interface allows you to implement your own screen-independent resources by registering a closure to produce an appropriate screen-dependent resource for any given screentype.

```
INTERFACE Palette;

IMPORT VBT, ScreenType, PaintOp, Cursor, Pixmap, Font,
  ScrnPaintOp, ScrnCursor, ScrnPixmap, ScrnFont;
```

Translating a screen-independent resource into its screen-dependent form is called *resolving* the resource. Here are the closure types for resolving resources:

```
TYPE
  OpClosure = OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    apply(st: VBT.ScreenType): ScrnPaintOp.T;
  END;

  CursorClosure = OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    apply(st: VBT.ScreenType): ScrnCursor.T;
  END;

  PixmapClosure = OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    apply(st: VBT.ScreenType): ScrnPixmap.T;
  END;

  FontClosure = OBJECT METHODS
    <* LL.sup <= VBT.mu *>
    apply(st: VBT.ScreenType): ScrnFont.T;
  END;
```

When an `apply` method is called, `st # NIL`. If the method returns `NIL`, then some default screen-dependent resource will be used; for example, the built-in font or the transparent painting operation.

The following procedures produce screen-independent resources from closures:

```
PROCEDURE FromOpClosure(cl: OpClosure): PaintOp.T;
<* LL.sup <= VBT.mu *>
```

Return a `PaintOp.T` that behaves like `cl.apply(st)` on `st`.

```
PROCEDURE FromCursorClosure
  (cl: CursorClosure): Cursor.T; <* LL.sup <= VBT.mu *>
```

Return a `Cursor.T` that behaves like `cl.apply(st)` on `st`.

```
PROCEDURE FromPixmapClosure
```

```
  (cl: PixmapClosure): Pixmap.T; <* LL.sup <= VBT.mu *>
```

Return a `Pixmap.T` *that behaves like* `cl.apply(st)` *on* `st`*.*

```
PROCEDURE FromFontClosure(cl: FontClosure): Font.T;
<* LL.sup <= VBT.mu *>
```

Return a `Font.T` *that behaves like* `cl.apply(st)` *on* `st`*.*

If your apply method that resolves a resource needs to resolve some other resource, you should use one of the following procedures to do so. In all cases, `st` must be non-`NIL`.

```
PROCEDURE ResolveOp(st: VBT.ScreenType; op: PaintOp.T)
  : ScrnPaintOp.T;
```

Resolve `op` *for* `st`*.*

```
PROCEDURE ResolveCursor(st: VBT.ScreenType;
  cursor: Cursor.T): ScrnCursor.T;
```

Resolve `cursor` *for* `st`*.*

```
PROCEDURE ResolvePixmap(st: VBT.ScreenType;
  pixmap: Pixmap.T): ScrnPixmap.T;
```

Resolve `pixmap` *for* `st`*.*

```
PROCEDURE ResolveFont(st: VBT.ScreenType; font: Font.T)
  : ScrnFont.T;
```

Resolve `font` *for* `st`*.*

If you create a cycle of screen-independent resources each of which tries to resolve the next resource in the cycle, then the program will deadlock.

To implement screen-independent resources, every screentype includes a *palette*, which is a table of screen-dependent resources appropriate for that screentype. Most clients don't need to worry about the palette, but if you are implementing a `VBT` class that translates to some other window system—like X or Microsoft Windows—here is the procedure for building the palette in the screentype for a top-level window:

```
PROCEDURE Init(st: VBT.ScreenType);
<* LL.sup = VBT.mu.v *>
```

Initialize `st`*'s palette, if it is not already initialized, by resolving all screen-independent resources for* `st` *and storing the results.*

```
END Palette.
```

7.6 The ScreenType interface

A `ScreenType.T` represents a class of screens that have a common pixel depth, a common set of operations on the pixels, and common repositories for cursors, pixmaps, and fonts.

When the screentype of a `VBT` changes, any screen-dependent resources for the old screentype become useless. The application must use the new screentype's *oracles* to look up resources that are valid for the new screentype. This is all handled automatically if you use screen-independent resources that are predefined or defined by somebody else. But you will need to use this interface if you are implementing your own screen-independent resources.

```
INTERFACE ScreenType;

IMPORT ScrnCursor, VBT, ScrnColorMap, ScrnFont,
  ScrnPaintOp, ScrnPixmap;

TYPE T = VBT.ScreenType;

REVEAL VBT.ScreenType <: Public;

TYPE
  Public = VBT.ScreenTypePublic OBJECT (*CONST*)
    bg, fg: ScrnPaintOp.Pixel;
    bits: T;
    op: ScrnPaintOp.Oracle;
    cursor: ScrnCursor.Oracle;
    pixmap: ScrnPixmap.Oracle;
    font: ScrnFont.Oracle;
    cmap: ScrnColorMap.Oracle;
  END;
```

For a screentype `st`, the values `st.bg` and `st.fg` are the pixel values that represent the user's default background and foreground colors on `st`. If the screen is color-mapped, these are appropriate for the default colormap. For applications doing simple painting, `bg` is logical white and `fg` is logical black. Depending on the screen and user preferences, the actual colors that the user sees might be different.

The screentype `st.bits` is the screentype for 1-bit deep pixmap sources for painting on screens of type `st`. It is guaranteed that `st.bits.bits=st.bits`, `st.bits.fg=1`, and `st.bits.bg=0`.

The oracles `st.op`, `st.font`, `st.cursor`, and `st.pixmap` contains methods that provide screen-dependent resources appropriate for `st`—for example, `st.font` has a method that will look up fonts by name.

If `st.cmap # NIL`, `st` is a color-mapped screen, which means that the color of a pixel is determined by looking up its value in a table. The color map can be either readonly or writable.

```
END ScreenType.
```

7.7 Screen-dependent painting operations

```
INTERFACE ScrnPaintOp;

IMPORT TrestleComm, PaintOp;
```

A `ScrnPaintOp.T` is a painting operation that is valid for some particular screentype.

If `op` is a `ScrnPaintOp.T` valid for screentype `st`, then `op` maps a source pixel `s` and destination pixel `d` to a result pixel `op(d, s)`. It will be used in a painting operation that sets `d := op(d, s)`. Both `d` and `op(d, s)` have type `st`, and `s` either has type `st` or `st.bits`. (The type `st.bits` is the screentype for one-bit deep sources that can be used with `st`.) For example, in a copy operation, `s` has type `st`, while in painting a bitmap, `s` has type `st.bits`.

A `ScrnPaintOp.Oracle` is meaningful only as the `op` field of some screentype `st`. It provides methods to generate `ScreenPaintOp.T`s that are valid for `st`.

A *tint* is a paintop that is independent of `s`. If `op` is a tint, we write `op(d)` instead of `op(d, s)`. (Even in the case of a tint, the type of `s` must be `st.bits`; otherwise the result of applying the tint is undefined.)

7.7.1 Obtaining handles from the oracle

```
TYPE
  Pixel = INTEGER;
  Oracle = Private OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      opaque(pix: Pixel): T
        RAISES {Failure, TrestleComm.Failure};
      bgfg(bg, fg: T): T
        RAISES {Failure, TrestleComm.Failure};
      swap(p,q: Pixel): T
        RAISES {Failure, TrestleComm.Failure};
      transparent(): T
        RAISES {Failure, TrestleComm.Failure};
      copy(): T
        RAISES {Failure, TrestleComm.Failure};
      builtIn(op: PaintOp.Predefined): T;
    END;
    Private <: ROOT;

EXCEPTION Failure;
```

For a screentype `st`, the field `st.op` is an `Oracle` whose methods satisfy the following specifications:

The method call

```
op := st.op.opaque(pix)
```

sets op to a tint such that op(p) = pix for any p. The method call

```
op := st.op.bgfg(bg, fg)
```

sets op to a tint such that op(p, 0) = bg(p) and op(p, 1) = fg(p), for any p, if bg and fg are tints. The method call

```
op := st.op.swap(p, q)
```

sets op to a tint such that op(p)=q, op(q)=p, and for any x, op(op(x))=x. The method call

```
op := st.op.transparent()
```

sets op to a tint such that op(p) = p for any p. The method call

```
op := st.op.copy()
```

sets op to a painting operation such that op(p, q) = q for any p and q. The method call

```
st.op.builtIn(op)
```

returns the operation valid for st that corresponds to the predefined screen-independent operation PaintOp.T{op}.

The exception Failure is raised if the screentype cannot provide the requested painting operation. For all the methods, LL.sup <= VBT.mu.

```
TYPE
  PlaneWiseOracle = Oracle OBJECT
    METHODS <* LL.sup <= VBT.mu *>
      planewise(
          READONLY mask: ARRAY OF BOOLEAN;
          op1, op2: T): T
        RAISES {Failure, TrestleComm.Failure};
  END;
```

If a screentype's op oracle is a PlaneWiseOracle (which you can test with TYPECASE), then you can use its planewise method to define painting operations by their effects on each bit position of the destination pixel. Let p[i] denote bit i of pixel p. Assuming NUMBER(mask) = st.depth, the method call

```
op := st.op.planewise(mask, bitOps)
```

sets op so that for d and s of screentype st and i in [0..st.depth-1],

```
IF mask[i] THEN
   op(d, s)[i] = op1(d[i], s[i])
ELSE
   op(d, s)[i] = op2(d[i], s[i])
END
```

The method may raise `Failure` if it does not support a particular combination of `op1`, `op2`, and `mask`.

The convenience procedure `ConstructPlanewiseOp` can be used to construct a painting operation from an array of boolean functions represented by the enumeration by `BitOp`:

```
TYPE
  BitOp = {
    Zero,           (* 0 *)
    And,            (* dest AND src *)
    NotAnd,         (* (NOT dest) AND src *)
    Src,            (* src *)
    AndNot,         (* dest and (NOT src) *)
    Dest,           (* dest *)
    Xor,            (* dest XOR src *)
    Or,             (* dest OR src *)
    Nor,            (* (NOT dest) AND (NOT src) *)
    Equal,          (* dest XOR (NOT src) *)
    Invert,         (* NOT dest *)
    NotOr,          (* (NOT dest) OR src *)
    NotSrc,         (* NOT src *)
    OrNot,          (* dest OR (NOT src) *)
    Nand,           (* (NOT dest) OR (NOT src) *)
    One};           (* 1 *)

PROCEDURE ConstructPlanewiseOp(
  pwo: PlaneWiseOracle;
  READONLY bitOps: ARRAY OF BitOp): T
RAISES {Failure, TrestleComm.Failure};
<* LL.sup <= VBT.mu *>
```

Return the painting operation that applies `bitOp[i]` to plane `i` of the source and destination.

If `NUMBER(bitOps) = st.depth` then `ConstructPlanewiseOp` uses `pwo` to construct and return an operation `op` such that for `s` and `d` of screentype `st` and `i` in `[0 .. st.depth-1]`,

```
op(d, s)[i] = bitOps[i](d[i], s[i])
```

The procedure may raise `Failure` if the screentype does not support a particular array `bitOps`.

7.7.2 The handle object

```
TYPE
  T <: Public;
  Public = OBJECT id: INTEGER; pix: Pixel := -1 END;
```

If p is a T, then p.id is an identifier whose interpretation depends on the screentype. If p was created by a call st.op.opaque(pix), then p.pix = pix; otherwise p.pix = -1.

```
END ScrnPaintOp.
```

7.8 Screen-dependent cursors

A ScrnCursor.T is a handle on a cursor shape that is valid for some particular screentype, called the *owner* of the handle. Some handles have names; others are anonymous. A named handle is valid forever. The cursor referenced by an anonymous handle will be garbage-collected when all handles to it have been dropped.

```
INTERFACE ScrnCursor;

IMPORT Point, ScrnPixmap, TrestleComm, Cursor;

EXCEPTION Failure;

VAR DontCare: T;

TYPE Raw = Cursor.Raw;
```

See the Cursor interface for the raw representation of a cursor shape as a pair of bitmaps, color information, and hotspot offset.

7.8.1 Obtaining handles from the oracle

```
TYPE
  Oracle = Private OBJECT (*CONST*)
    width, height: INTEGER;
  METHODS
    <* LL.sup <= VBT.mu *>
    load(READONLY r: Raw; nm: TEXT := NIL): T
      RAISES {TrestleComm.Failure};
    list(pat: TEXT; maxResults: CARDINAL := 1)
      : REF ARRAY OF TEXT
      RAISES {TrestleComm.Failure};
    lookup(name: TEXT): T RAISES {TrestleComm.Failure};
    builtIn(cs: Cursor.Predefined): T;
  END;
  Private <: ROOT;
```

For a screentype st, the field st.cursor is an Oracle that produces cursors owned by st:

The integers st.cursor.width and st.cursor.height are the dimensions in pixels of the largest cursor image that the screentype st supports. Larger images will be cropped; smaller images will be padded.

The method call

```
st.cursor.load(r, nm)
```

allocates and returns a cursor handle `c` owned by `st` whose contents are equal to `r`. If `nm # NIL`, `c` receives the name `nm`, and any cursor handle owned by `st` that previously had the name `nm` becomes anonymous.

The method call

```
st.cursor.list(pat, maxResults)
```

returns the names of all cursors owned by `st` that match the pattern `pat`. The list of results may be truncated to length `maxResults`. A `*` matches any number of characters and a `?` matches a single character.

The method call

```
st.cursor.lookup(name)
```

return the cursor handle owned by `st` with the given name, or `NIL` if no cursor has this name.

The method call

```
st.cursor.builtIn(cs)
```

returns the screen-dependent cursor valid for `st` that corresponds to the predefined screen-independent cursor `Cursor.T{cs}`.

The locking level for all methods is `LL.sup <= VBT.mu`.

7.8.2 The handle object

```
TYPE
  T <: Public;
  Public = OBJECT (*CONST*)
    id: INTEGER
  METHODS
    <* LL.sup <= VBT.mu *>
    localize(): Raw
      RAISES {TrestleComm.Failure, Failure};
    unload() RAISES {TrestleComm.Failure};
  END;
```

If `cs` is a `ScrnCursor.T`, then `cs.id` is an identifier whose interpretation depends on the screentype that owns `cs`. The method call `cs.localize()` returns a raw cursor equal to the one on which `cs` is a handle, and the method call `cs.unload()` causes `cs` to become anonymous.

```
END ScrnCursor.
```

7.9 Screen-dependent pixmaps

A `ScrnPixmap.T` is a handle on a rectangular array of pixels that is valid for use on a particular screentype, called the *owner* of the handle. Some handles have names; others are anonymous. A named handle is valid forever; the pixmap referenced by an anonymous handle will be garbage-collected when all handles to it have been dropped.

```
INTERFACE ScrnPixmap;

IMPORT Point, Rect, Word, TrestleComm, Pixmap;

EXCEPTION Failure;

TYPE Raw = Pixmap.Raw;
```

The raw representation of a pixmap is revealed at the end of this interface.

7.9.1 Obtaining handles from the oracle

```
TYPE
  Oracle = Private OBJECT
  METHODS
    <* LL.sup <= VBT.mu *>
    load(READONLY r: Raw; nm: TEXT := NIL): T
      RAISES {TrestleComm.Failure};
    list(pat: TEXT; maxResults: CARDINAL := 1)
      : REF ARRAY OF TEXT RAISES {TrestleComm.Failure};
    lookup(name: TEXT): T RAISES {TrestleComm.Failure};
    builtIn(pm: Pixmap.Predefined): T;
  END;
  Private <: ROOT;
```

For a screentype `st`, the field `st.pixmap` is an `Oracle` that produces pixmaps owned by `st`.

The method call `st.pixmap.load(r, nm)` allocates and returns a pixmap handle `p` owned by `st` whose contents are equal to `r`. The depth of `r` must either be `1` or `st.depth`, otherwise there is a checked runtime error. If `nm # NIL`, `p` receives the name `nm`, and any pixmap handle owned by `st` that previously had the name `nm` becomes anonymous.

The method call `st.pixmap.list(pat, maxResults)` returns the names of all pixmaps owned by `st` that match the pattern `pat`. The list of results may be truncated to length `maxResults`. A `*` matches any number of characters and a `?` matches any single character.

The method call `st.pixmap.lookup(name)` return the pixmap with the given name, or `NIL` if no pixmap has this name.

The method call `st.pixmap.builtIn(pm)` returns the screen-dependent pixmap valid for `st` that corresponds to the predefined screen-independent `Pixmap.T{pm}`.

The locking level for all methods is `LL.sup <= VBT.mu`.

7.9.2 The handle object

```
TYPE
  T <: Public;
  Public = OBJECT (*CONST*)
    id: INTEGER;
    depth: INTEGER;
    bounds: Rect.T
  METHODS
    <* LL.sup <= VBT.mu *>
    localize(READONLY rect: Rect.T): Raw
      RAISES {TrestleComm.Failure};
    unload() RAISES {TrestleComm.Failure};
    free() RAISES {TrestleComm.Failure}
  END;
```

If `pm` is a `ScrnPixmap.T`, then `pm.id` is an identifier whose interpretation depends on the screentype that owns `pm`. The field `pm.depth` is the number of bits in each pixel of `pm`, and `pm.bounds` is the rectangular extent of `pm`.

The method call `pm.localize(rect)` returns a raw pixmap equal to a rectangualr subpixmap of the one on which `pm` is a handle. The bounds of the raw pixmap returned by `localize` is `Rect.Meet(rect, pm.bounds)`.

The method call `pm.unload()` causes `pm` to become anonymous.

Pixmaps consume large amounts of memory. The method call `pm.free()` releases the memory associated with the pixmap. You must make sure that all `VBT`s using `pm` have finished painting before you free it. After a call to `free`, the pixmap bounds and contents are arbitrary.

7.9.3 The raw representation

A raw pixmap allows the client to directly locate and modify the bits of the pixmap. The following procedure produces a new raw pixmap:

```
PROCEDURE NewRaw(dpth: INTEGER;
  READONLY bnds: Rect.T): Raw;
<* LL arbitrary *>
```

Allocate and return a raw pixmap with the given depth and bounds.

The initial contents of the pixmap returned by `NewRaw` are undefined.

Here is the representation of a raw pixmap:

```
REVEAL Pixmap.Raw <: Raw_Public;

TYPE
  Raw_Public = OBJECT
    depth: INTEGER;
```

```
    bounds: Rect.T;
    pixels: REF ARRAY OF Word.T;
    offset: INTEGER;
    bitsPerPixel: INTEGER;
    wordsPerRow: INTEGER;
    pixelOrder: ByteOrder;
    westRounded: INTEGER;
  METHODS
    get(READONLY pt: Point.T): Pixel;
    set(READONLY pt: Point.T; pix: Pixel);
    sub(READONLY rect: Rect.T): Raw;
  END;

  Pixel = Word.T;
  ByteOrder = {MSBFirst, LSBFirst};
```

The methods provide the easiest way to operate on a raw pixmap, and we will explain them first. Let `pm` be a `ScrnPixmap.Raw`, then:

The method call

```
pm.get(pt)
```

returns the pixel value at the point `pt` in the pixmap. The result is undefined if `pt` is not in `pm.bounds`.

The method call

```
pm.set(pt, pix)
```

sets the pixel value at the point `pt` of the pixmap `pm` to the value `pix`. It is a noop if `pt` is not in `pm.bounds`.

The method call

```
pm.sub(rect)
```

returns a pixmap whose bounds are `Rect.Meet(rect, pm.bounds)` and whose contents are shared with `pm`'s.

It is also possible to bypass the methods and access the data in the raw pixmap directly. Here is the specification for the internal layout of pixels in a raw pixmap:

A value `pm` of type `Pixmap.Raw` is a rectangular subregion of a larger rectangular pixmap, which we shall call the *surround*. The surround is a word-aligned pixmap, stored in raster-scan order by rows. Pixels do not cross word boundaries. More precisely, the westmost pixel in each row of the surround is always a pixel whose `h`-coordinate is a multiple of `pixelsPerWord` (which is equal to `Word.Size DIV pm.bitsPerPixel`). The eastmost pixel in each row of the surround is always a pixel whose `h`-coordinate modulo `pixelsPerWord` is congruent to `pixelsPerWord-1`. Hence, the number of pixels in each row of the surround is a multiple of `pixelsPerWord`. The value `pm.wordsPerRow` is the number of words that are needed to store one row of the surround.

The value `pm.bitsPerPixel` might be greater than `pm.depth`; for example, a twelve-bit deep pixmap might be stored with sixteen bits per pixel.

The pixels of the surround are stored in the array `pm.pixels`. Each row is represented in `pm.wordsPerRow` adjacent words; the first of these words stores the westmost `pixelsPerWord` pixels of the row, the following word stores the adjacent `pixelsPerWord` pixels, and so on until the last word, which stores the eastmost `pixelsPerWord` pixels.

The order in which pixels are packed into words is indicated by `pm.pixelOrder`. In this discussion, bit `0` is the least significant bit and bit `Word.Size - 1` is the most significant bit of a word.

If `pm.pixelOrder = LSBFirst`, the bits of the pixels are as follows (where `bpp` is `pm.bitsPerPixel`):

pixel `0`: bits `0..bpp-1`
pixel `1`: bits `bpp..2*bpp-1`
`...`
pixel `i`: bits `i*bpp..(i+1)*bpp-1`

If `pm.pixelOrder = MSBFirst`, the pixels are stored in reverse order, so that pixel `i` occupies the same bits as pixel `pixelsPerWord-i-1` occupies for `LSBFirst`.

A `Word.Extract` of the bits indicated above, from the correct word, gives the pixel's value. If the word size does not contain an integral number of pixels, the unused bits in the word have undefined values.

The pixmap `pm` itself is a rectangular region selected from the surround; the value `pm.bounds`, of type `Rect.T`, specifies the domain of `pm`. The value `pm.offset` specifies where in `pm.pixels` the words containing the pixels of `pm` can be found. In particular, the northwestern-most bit of `pm`, the bit with coordinates

`h = pm.bounds.west` and `v = pm.bounds.north`,

is stored in word `pm.pixels[pm.offset]`. The pixel is the `(pm.bounds.west MOD pixelsPerWord)`'th pixel of the word. Its bits can be found by the earlier formulas.

The general formula for the word containing the pixel with position `h, v` is

```
pm.pixels[
 (v - pm.bounds.north) * pm.wordsPerRow +
 (h - pm.westRounded) DIV pixelsPerWord) + pm.offset].
```

Here is another useful formula. The surround rectangle must be at least wide enough to contain the subrectangle `pm.bounds`, even after we have rounded the west edge of `pm.bounds` westward to the next word boundary and rounded the east edge of `pm.bounds` eastward to the next word boundary. As a result, we have the inequality:

```
pm.wordsPerRow >=
  ((pm.bounds.east - 1) DIV pixelsPerWord) -
  (pm.bounds.west DIV pixelsPerWord) + 1
```

Finally, the value `pm.westRounded` is provided for convenience; it is equal to

```
bounds.west - (bounds.west MOD pixelsPerWord),
```

that is, the western boundary moved west to the nearest word boundary.

```
END ScrnPixmap.
```

7.10 Screen-dependent fonts

A `ScrnFont.T` is a handle on a typeface that is valid for some particular screen-type, called the *owner* of the handle. All handles have names, which are highly conventionalized strings encoding the size, style, and other properties of the typeface.

```
INTERFACE ScrnFont;

IMPORT ScrnPixmap, Rect, TrestleComm, Font;

EXCEPTION Failure;
```

7.10.1 Obtaining handles from the oracle

```
TYPE
  Oracle = Private OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      list(pat: TEXT; maxResults := 1):
        REF ARRAY OF TEXT RAISES {TrestleComm.Failure};
      match(
        family: TEXT;
        pointSize: INTEGER := 120;
        slant: Slant := Slant.Roman;
        maxResults: CARDINAL := 1;
        weightName: TEXT := AnyMatch;
        version: TEXT := "";
        foundry: TEXT := AnyMatch;
        width: TEXT := AnyMatch;
        pixelsize: INTEGER := AnyValue;
        hres, vres: INTEGER := ScreenTypeResolution;
        spacing: Spacing := Spacing.Any;
        averageWidth: INTEGER := AnyValue;
        charsetRegistry: TEXT := "ISO8859";
        charsetEncoding: TEXT := "1")
      : REF ARRAY OF TEXT RAISES {TrestleComm.Failure};
      lookup(name: TEXT): T
```

```
      RAISES {Failure, TrestleComm.Failure};
    builtIn(f: Font.Predefined): T;
END;
Private <: ROOT;
```

For a screentype `st`, the field `st.font` is an `Oracle` that produces font handles owned by `st`.

The method call

```
st.font.list(pat, maxResults)
```

returns the names of all fonts owned by `st` that match the pattern `pat`. The list of results may be truncated to length `maxResults`. A `*` matches any number of characters and a `?` matches a single character.

The arguments to the `match` method specify various font attributes, as explained below. The method call

```
st.font.match(...)
```

returns the names of all font handles owned by `st` that match the specifications. The list of results may be truncated to the length `maxResults`. If no fonts match the specifications, the result will be either `NIL` or an empty array. Passing `AnyMatch` for a text attribute, or `AnyValue` for an integer attribute, allows any value for that attribute. For text attributes, partial text matches are also possible: a `*` matches any number of characters and `?` matches a single character.

The method call

```
st.font.lookup(name)
```

returns the font handle owned by `st` with the given name. Generally `name` should be one of the names returned by the `list` or `match` method.

The method call

```
st.font.builtIn(f)
```

returns the screen-dependent font valid for `st` that corresponds to the predefined screen-independent font `Font.T{f}`.

The locking level for all methods is `LL.sup <= VBT.mu`.

7.10.2 Font attributes

The arguments to a font oracle list method specify font attributes whose full specifications are the "X Logical Font Description Conventions Version 1.3", an MIT X Consortium Standard which can be found in Part IV of *X Window System* by Scheifler and Gettys [5]. Here they are described in brief.

The argument `family` specifies the family of the typeface. To find out what fonts your X server has, run the `xlsfonts` program. Most servers support the families `Courier`, `Helvetica`, and `Times`, among others.

The argument `pointsize` is ten times the font's size in points; e.g., 120 for a standard 12-point font.

The argument `slant` is an element of the following enumeration type:

```
TYPE
  Slant = {Roman, Italic, Oblique, ReverseItalic,
    ReverseOblique, Other, Any};
```

whose elements have the following interpretations:

`Roman`: Upright letters in a roman style.

`Italic`: Clockwise slanted letters in an italic style.

`Oblique`: Clockwise slanted letters in a roman style.

`ReverseItalic`: Counter clockwise slanted letters in an italic style.

`ReverseOblique`: Counter clockwise slanted letters in a roman style.

`Other`: None of the above

`Any`: Any of the above (including `Other`).

The argument `weightName` is the foundry's name for the font's weight; e.g., `Bold`, `DemiBold`, or `Medium`.

The argument `version` specifies the version of the *X Logical Font Description Conventions* that describes the format of a font's name. If the argument is omitted, Version 1.3 is assumed. (Version 1.3 is the only version as these words are written.)

The argument `foundry` specifies the X registered name for the font's foundry, e.g., `Adobe`, `B&H`, `Bitstream`, `DEC`.

The argument `width` specifies the foundry's name for the font's width; e.g., `Normal` or `Condensed`.

The argument `pixelsize` specifies the size of the font in pixels. The size in points depends on the vertical resolution of the device: A pixelsize of 20 could represent a 20-point font at 75 pixels per inch or a 10-point font at 150 pixels per inch.

The arguments `hres` and `vres` specify the horizontal and vertical screen resolution for which the font is designed, in pixels per inch.

The argument `spacing` is an element of the following enumeration:

```
TYPE Spacing =
  {Proportional, Monospaced, CharCell, Any};
```

whose elements have the following meaning:

`Proportional`: Character widths vary.

`Monospaced`: Character widths are constant.

`CharCell`: Font is self-clearing, as defined in the `VBT` interface.

`Any`: Any of the above.

The argument `averageWidth` specifies the un-weighted arithmetic mean of the widths of all glyphs in the font, measured in tenths of a pixel.

The arguments `charsetRegistry` and `charsetEncoding` are the X names of the font's character set and encoding scheme; e.g., `ISO8859` and `1` for ISO Latin-1 fonts. See Appendix G of [5].

```
CONST
  AnyMatch = "*";
  AnyValue = -1;
  ScreenTypeResolution = -2;
```

Passing `AnyMatch` as an argument to the `list` method matches any text value for the corresponding attribute, and `AnyValue` matches any integer value. Passing `ScreenTypeResolution` for `hres` or `vres` matches fonts whose horizontal and vertical resolutions agree with the screentype that owns the font.

7.10.3 Registering fonts

Some screentypes allow the client to register fonts. The client registers the font's strike (bits) and metrics (description) with the `StrikeOracle`. The name of the font is implied by the attributes in the metrics, so the `list` and `lookup` methods will find client-registered fonts.

```
TYPE
  StrikeOracle = Oracle OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      load(strike: Strike; metrics: Metrics): T
        RAISES {Failure, TrestleComm.Failure};
    END;
```

The method call `st.font.load(strike, metrics)` creates a font owned by `st` with the given strike and metrics and returns a handle to it.

The `metrics` argument must define all of the initial fields of the font metrics record: `family`, `pointSize`, ..., `isAscii`, and `defaultChar`. The values `minBounds` and `maxBounds` must be provided if `charMetrics` is `NIL`; otherwise if `printWidth` is `AnyValue`, the `load` method will compute them from `charMetrics`. If any of the remaining fields have the value `AnyValue`, the `load` method will compute them.

7.10.4 The handle object

```
TYPE
  T <: Public;
  Public = OBJECT (*CONST*)
    id: INTEGER;
    metrics: Metrics
  END;
```

```
TYPE StrikeFont = T OBJECT
  METHODS <* LL.sup <= VBT.mu *>
    strike(): Strike RAISES {TrestleComm.Failure}
  END;

TYPE Strike = OBJECT
  METHODS <* LL.sup <= VBT.mu *>
    glyph(ch: INTEGER): ScrnPixmap.T;
  END;
```

If `f` is a `ScrnFont.T`, then `f.id` is an identifier whose interpretation depends on the screentype that owns `f` and `f.metrics` are the metrics for `f`. If in addition `f` is a `StrikeFont`, then `f.strike()` returns `f`'s strike. The screentype of the strike's pixmaps will be the screentype that owns `f`.

If `str` is a `Strike`, then `str.glyph(ch)` is the pixmap for the character `ch`. This will be empty except for characters in the range `[m.firstChar..m.lastChar]`, where `m` is the metrics (see below) for the font of which `str` is the strike.

```
PROCEDURE BoundingBox(txt: TEXT; fnt: T): Rect.T;
<* LL arbitrary *>
```

Return the smallest rectangle that contains the bounding boxes of the characters of `txt` if `txt` were painted in the font `fnt` with `txt`'s reference point at the origin.

```
PROCEDURE BoundingBoxSub(
  READONLY txt: ARRAY OF CHAR;
  fnt: T): Rect.T;
<* LL arbitrary *>
```

Like `BoundingBox` but takes an array instead of a `TEXT`.

```
PROCEDURE TextWidth(txt: TEXT; fnt: T): INTEGER;
<* LL arbitrary *>
```

Return the sum of the printing widths of the characters in `txt` in the font `fnt`.

7.10.5 The raw representation

```
TYPE
  CharMetric = RECORD
    printWidth: INTEGER;
    boundingBox: Rect.T;
  END;
  CharMetrics = REF ARRAY OF CharMetric;
```

The `printWidth` of a character is the displacement to the next character's reference point.

The `boundingBox` of a character is the smallest rectangle with sides parallel to the axes that contains the glyph of the character placed with its reference point at (0,0).

```
TYPE
  Metrics = OBJECT (*CONST*)
    family: TEXT;
    pointSize: INTEGER;
    slant: Slant;
    weightName: TEXT;
    version: TEXT;
    foundry: TEXT;
    width: TEXT;
    pixelsize: INTEGER;
    hres, vres: INTEGER;
    spacing: Spacing;
    averageWidth: INTEGER;
    charsetRegistry: TEXT;
    charsetEncoding: TEXT;
    firstChar, lastChar: INTEGER;
    charMetrics: CharMetrics;
    selfClearing: BOOLEAN;
    rightKerning, leftKerning: BOOLEAN;
    isAscii: BOOLEAN;
    defaultChar: INTEGER;
    minBounds, maxBounds: CharMetric;
  METHODS <* LL arbitrary *>
    intProp(name: TEXT; ch: INTEGER := -1): INTEGER
      RAISES {Failure};
    textProp(name: TEXT; ch: INTEGER := -1): TEXT
      RAISES {Failure};
  END;
```

The fields from `family` to `charSetEncoding` in the `Metrics` object specify the attributes that were defined for the `lookup` method. A value of `*` or `Any` in one of these fields means that the corresponding attribute is unknown.

The integers `firstChar` and `lastChar` are the indices of the first and last characters defined in the font.

The array `charMetrics` specifies the metrics of the individual characters. The metrics for character `ch` are in `charMetrics[ch-firstChar]`. If all characters have the same `printWidth` and `boundingBox`, then these values are stored in `minBounds` and `maxBounds` and the `charMetrics` field is `NIL`.

The flag `selfClearing` indicates whether the font is self-clearing, as defined in the `VBT` interface, and the two kerning flags indicate the present of right and left kerning in the font.

The flag `isAscii` indicates that character codes 32-126 (base 10) have their normal ASCII meanings.

The integer `defaultChar` is the code for the recommended character to display in the place of a character that isn't defined for the font.

The rectangles `minBounds.boundingBox` and `maxBounds.boundingBox` contain the meet and join, respectively, of the bounding boxes of all characters in the font when they are positioned with their reference points at (0, 0). The values `minBounds.printWidth` and `maxBounds.printWidth` are the minimum and maximum printing widths for all characters in the font.

The method call `m.intProp(nm)` returns the integer value of the font attribute named `nm`, or raises `Failure` if this attribute is not defined for `m`. The method call `m.intProp(nm, ORD(ch))` returns the integer value of the font attribute named `nm` for the character `ch`, or raises `Failure` if this attribute is not defined for `(m, ch)`. The `textProp` method is similar.

The set of attributes returned by the metrics methods depend on the font. Fonts that are owned by X screentypes support the attributes defined in Part IV of *X Window System* (*op. cit.*); we recommend that other fonts support them too. (To read an X font attribute whose type is an X atom, use the `textProp` method, which returns the name of the atom.)

```
END ScrnFont.
```

7.11 Color maps

A `ScrnColorMap.T` is a handle on a colormap that is valid for some particular screentype, called the *owner* of the handle. Some handles have names; others are anonymous. A named handle is valid forever. The colormap referenced by an anonymous handle will be garbage-collected when all handles to it have been dropped.

Every colormap has a *depth*; the pixel values defined by the color map are in the range `[0..(2^depth)-1]`. Every color-mapped screentype defines a set of *preferred* colors that cover the spectrum reasonably densely. Some preferred colors are designated as *stable*.

Clients can allocate pixels out of a color map as read-only shared entries or as writable exclusive entries. The implementation maintains reference counts on the read-only entries so that an entry can be freed when it is no longer allocated to any client.

```
INTERFACE ScrnColorMap;

IMPORT TrestleComm;
```

7.11.1 Obtaining handles from the oracle

```
TYPE
  Oracle = Private OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      standard(): T RAISES {TrestleComm.Failure};
```

```
      new(name: TEXT := NIL; preLoaded := TRUE): T
        RAISES {TrestleComm.Failure, Failure};
      lookup(name: TEXT): T
        RAISES {TrestleComm.Failure};
      list(pat: TEXT; maxResults: CARDINAL := 1)
        : REF ARRAY OF TEXT RAISES {TrestleComm.Failure}
    END;
  Private <: ROOT;

EXCEPTION Failure;
```

Every color-mapped screentype `st` contains a field `st.cmap` of type `Oracle`, which hands out colormaps owned by `st`:

The method call

```
st.cmap.standard()
```

returns the default colormap owned by `st`. This is the colormap that a top-level window will initially have when it is rescreened to `st`. Initially, the stable colors are allocated read-only with a reference count of one.

The method call

```
st.cmap.new(name, preLoaded)
```

creates and returns a new colormap owned by `st` with the given name. If `preLoaded` is true, the stable colors are initially allocated read-only; otherwise nothing is allocated initially.

The method call

```
st.cmap.lookup(name)
```

returns the colormap owned by `st` with the given name, or `NIL` if no colormap has this name.

The method call

```
st.cmap.list(pat, maxResults)
```

returns the names of colormaps owned by `st` that match the pattern `pat`. The list of results may be truncated to length `maxResults`. A `*` matches any number of characters and a `?` matches any single character.

7.11.2 The handle object

```
TYPE
  T <: Public;
  Public = OBJECT (*CONST*)
      depth: INTEGER;
      readOnly: BOOLEAN;
      ramp: Ramp;
    METHODS
```

```
    <* LL.sup <= VBT.mu *>
    fromRGB(rgb: RGB; mode := Mode.Normal): Pixel
      RAISES {Failure, TrestleComm.Failure};
    read(VAR res: ARRAY OF Entry)
      RAISES {TrestleComm.Failure};
    write(READONLY new: ARRAY OF Entry)
      RAISES {Failure, TrestleComm.Failure};
    new(dim: CARDINAL): Cube RAISES
      {Failure, TrestleComm.Failure};
    free(READONLY cb: Cube)
      RAISES {TrestleComm.Failure};
  END;
Mode = {Stable, Normal, Accurate};
Ramp = RECORD
  base: INTEGER;
  last, mult: ARRAY Primary OF INTEGER;
END;
Primary = {Red, Green, Blue};
Cube = RECORD lo, hi: Pixel END;
Pixel = INTEGER;
RGB = RECORD r, g, b: REAL END;
Entry = RECORD pix: Pixel; rgb: RGB END;
```

The field `cm.depth` is the depth of `cm`, and `cm.readOnly` is `TRUE` if `cm` cannot be written. The field `cm.ramp` defines a three dimensional lattice of colors preallocated in `cm`, as follows.

If `cm.ramp.base` is `-1`, the lattice of preallocated colors is empty.

If `cm.ramp.base` is not `-1`, then the pixel value

```
base + r*mult[Red] + g*mult[Green] + b*mult[Blue]
```

represents the color `(r/last[Red], g/last[Green], b/last[Blue])`, for `r` in the range `[0..last[Red]]`, `g` in the range `[0..last[Green]]`, and `b` in the range `[0..last[Blue]]`.

An `RGB` represents the color with the given blend of red, green, and blue. Each of the numbers is in the range `[0.0..1.0]`; thus the triple `(0.0, 0.0, 0.0)` specifies black. In case of a gray scale display, only the `r` component is relevant.

The method call

```
cm.fromRGB(rgb, mode)
```

extends the read-only portion of `cm` with a new entry whose value is near `rgb` and returns the pixel of the new entry. If the read-only portion of `cm` already contains an entry whose value is near `rgb`, that entry's pixel is returned. The `mode` argument controls how near the new entry's value will be to `rgb`, as follows. If `mode` is `Stable`, the new entry's color is the nearest stable color to `rgb`. If `mode` is `Normal`, the new entry's color is the nearest preferred color to `rgb`. If `mode` is `Accurate`, the new

entry's color is the nearest color to `rgb` that the hardware supports. The method raises `Failure` if a new entry is required but the colormap is full.

For each entry `e` in the array `res`, the method call

```
cm.read(res)
```

sets `e.rgb` to the color in `cm` of the pixel `e.pixel`.

The method call

```
cm.write(new)
```

changes the value of `cm` at `p` to be `rgb`, for each pair `(p, rgb)` in the array `new`, assuming all these pixels are writable. Otherwise the method raises `Failure`. The array `new` must be sorted.

The method call

```
cm.new(dim)
```

extends the writable portion of `cm` with a set of 2^{dim} new entries whose pixels form a cube, and returns the cube. The method raises `Failure` if the free entries of the colormap do not contain a cube of the given dimension.

A `Cube cb` represents a set of pixels by the following rule: a pixel `p` is in `cb` if `Word.And(lo, pix) = lo` and `Word.Or(hi, pix) = hi`.

The method call `cm.free(cb)` deallocates from the writable portion of `cm` each entry whose pixel is in the cube `cb`, assuming all of these pixels are allocated.

```
END ScrnColorMap.
```

8 Geometry interfaces

Most programs that use windows need to perform geometric calculations with integer lattice points. Such calculations can easily become obscure and error-prone. This section provides a set of geometry interfaces that help make them easier to read and write.

The interfaces are named `Axis`, `Point`, `Interval`, `Rect`, `Region`, `Path`, and `Trapezoid`. The locking level is arbitrary for all procedures in these interfaces.

8.1 The Axis Interface

`Axis.T.Hor` and `Axis.T.Ver` are Trestle's names for the horizontal and vertical axes. `Axis.Other` exchanges `Hor` and `Ver`.

```
INTERFACE Axis;

TYPE T = {Hor, Ver};

CONST Other = ARRAY T OF T {T.Ver, T.Hor};

END Axis.
```

8.2 The Point interface

A `Point.T` is a pair of integers representing a position in the plane. If `pt` is a point, then `pt.h` is the distance of `pt` to the right of the coordinate origin, and `pt.v` is the distance of `pt` below the coordinate origin. That is, the `hv` coordinate system is related to the Cartesian coordinate system by the equation `(h, v) = (x, -y)`.

```
INTERFACE Point; IMPORT Axis;

TYPE T = RECORD h, v: INTEGER END;

CONST Origin = T{0, 0};

PROCEDURE Add(READONLY p, q: T): T;
```
Return `T{p.h + q.h, p.v + q.v}`.

```
PROCEDURE Sub(READONLY p, q: T): T;
```
Return `T{p.h - q.h, p.v - q.v}`.

```
PROCEDURE Minus (READONLY p: T): T;
```
Return `T{-p.h, -p.v}`

```
PROCEDURE Mul(READONLY p: T; n: INTEGER): T;
```
Return `T{p.h * n, p.v * n}`.

```
PROCEDURE Div(READONLY p: T; n: INTEGER): T;
```
Return `T{p.h DIV n, p.v DIV n}`.

```
PROCEDURE Mod(READONLY p: T; n: INTEGER): T;
```
Return `T{p.h MOD n, p.v MOD n}`.

```
PROCEDURE Scale(READONLY p: T; num, den: INTEGER): T;
```
Return `Div(Mul(p, num), den)`.

```
PROCEDURE Min(READONLY p, q: T): T;
```
Return `T{MIN(p.h, q.h), MIN(p.v, q.v)}`.

```
PROCEDURE Max(READONLY p, q: T): T;
```
Return `T{MAX(p.h, q.h), MAX(p.v, q.v)}`.

```
PROCEDURE MoveH(READONLY p: T; dh: INTEGER): T;
```
Return `T{p.h+dh, p.v}`.

```
PROCEDURE MoveV(READONLY p: T; dv: INTEGER): T;
```
Return `T{p.h, p.v+dv}`.

```
PROCEDURE MoveHV(READONLY p: T; dh, dv: INTEGER): T;
```
Return `T{p.h+dh, p.v+dv}`.

```
PROCEDURE Transpose(READONLY p: T; ax := Axis.T.Ver): T;
```
If `ax = Hor` *then return* `p` *else return* `T{p.v, p.h}`.

For example, `Point.Transpose(pt, ax).h` is the `ax` component of `pt`.

```
PROCEDURE DistSquare(READONLY p, q: T): INTEGER;
```
Return the square of the Euclidean distance between `p` *and* `q`.

```
END Point.
```

8.3 The Interval interface

An `Interval.T` is a contiguous set of integers. An interval `a` contains an integer `n` if

```
a.lo <= n  AND  n < a.hi
```

We impose the restriction that if an interval contains no integers, then it must be equal as a record to `Interval.Empty`.

```
INTERFACE Interval;
TYPE T = RECORD lo, hi: INTEGER END;
```

```
CONST
  Empty = T{0, 0};
  Full  = T{FIRST(INTEGER), LAST(INTEGER)};
```

```
PROCEDURE FromBounds(lo, hi: INTEGER): T;
```
If `lo >= hi` then return `Empty`, else return `T{lo, hi}`.

```
PROCEDURE FromAbsBounds(n, m: INTEGER): T;
```
Return `FromBounds(MIN(n,m), MAX(n,m))`.

```
PROCEDURE FromBound(lo: INTEGER; s: CARDINAL): T;
```
Return `FromBounds(lo, lo+s)`.

```
PROCEDURE FromSize(s: CARDINAL): T;
```
Return `FromBounds(0, s)`.

```
PROCEDURE Move(READONLY a: T; n: INTEGER): T;
```
Return `FromBounds(a.lo+n, a.hi+n)`.

```
PROCEDURE Inset(READONLY a: T; n: INTEGER): T;
```
If `a` is empty then return `Empty`, else return `FromBounds(a.lo + n, a.hi - n)`.

```
PROCEDURE Change(READONLY a: T; dlo, dhi: INTEGER): T;
```
If `a` is empty then return `Empty`, else return `FromBounds(a.lo + dlo, a.hi + dhi)`.

```
PROCEDURE Join(READONLY a, b: T): T;
```
Return the smallest interval containing both `a` and `b`.

```
PROCEDURE Meet(READONLY a, b: T): T;
```
Return the largest interval contained in both of `a` and `b`.

```
PROCEDURE Project(READONLY a: T; n: INTEGER): INTEGER;
```
Return the element of `a` that is closest to `n`. This is a checked runtime error if `a` is empty.

```
PROCEDURE Mod(n: INTEGER; READONLY a: T): INTEGER;
```
Return the member of `a` whose distance from `n` is a multiple of `Size(a)`. This is a checked runtime error if `a` is empty.

```
PROCEDURE Size(READONLY a: T): CARDINAL;
```
Return `a.hi - a.lo`.

```
PROCEDURE Middle(READONLY a: T): INTEGER;
```

Return `(a.hi + a.lo) DIV 2`.

```
PROCEDURE Center(READONLY a: T; n: INTEGER): T;
```

If `a` *is empty then return* `Empty`*, else return* `b` *such that* `Size(b) = Size(a)` *and* `Middle(b) = n`.

```
PROCEDURE IsEmpty(READONLY a: T): BOOLEAN;
```

Return whether `a` *is empty.*

```
PROCEDURE Member(n: INTEGER; READONLY a: T): BOOLEAN;
```

Return whether `n` *is in* `a`.

```
PROCEDURE Overlap(READONLY a, b: T): BOOLEAN;
```

Return whether `a` *and* `b` *have any element in common.*

```
PROCEDURE Subset(READONLY a, b: T): BOOLEAN;
```

Return whether `a` *is contained in* `b`.

```
END Interval.
```

8.4 The Rect interface

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

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

We impose the restriction that if a rectangle contains no points, then it must be equal as a record to `Rect.Empty`.

```
INTERFACE Rect;

IMPORT Axis, Interval, Point;

TYPE T = RECORD west, east, north, south: INTEGER END;

CONST
  Empty = T{0,0,0,0};
  Full = T{FIRST(INTEGER), LAST(INTEGER),
           FIRST(INTEGER), LAST(INTEGER)};

PROCEDURE FromEdges(w, e, n, s: INTEGER): T;
```

If `w >= e` *or* `n >= s` *return* `Empty`*, else return* `T{w, e, n, s}`.

```
PROCEDURE FromAbsEdges(h1, h2, v1, v2: INTEGER): T;
```

Return

```
FromEdges(MIN(h1,h2), MAX(h1,h2),
          MIN(v1,v2), MAX(v1,v2))
```

```
PROCEDURE FromCorners(READONLY p, q: Point.T): T;
```

Return `FromAbsEdges(p.h, q.h, p.v, q.v)`.

```
PROCEDURE FromCorner(
  READONLY p: Point.T;
  hor, ver: CARDINAL): T;
```

Return `FromEdges(p.h, p.h+hor, p.v, p.v+ver)`.

```
PROCEDURE FromIntervals
  (READONLY hor, ver: Interval.T): T;
```

Return `FromEdges(hor.lo, hor.hi, ver.lo, ver.hi)`.

```
PROCEDURE FromPoint(READONLY p: Point.T): T;
```

Return the rectangle whose only element is `p`.

```
PROCEDURE FromSize(hor, ver: CARDINAL): T;
```

Return `FromCorner(Point.Origin, hor, ver)`.

```
PROCEDURE Add(READONLY r: T; READONLY p: Point.T): T;
```

Return

```
FromEdges(r.west+p.h,  r.east+p.h,
          r.north+p.v, r.south+p.v)
```

```
PROCEDURE Sub(READONLY r: T; READONLY p: Point.T): T;
```

Return `Add(r, Point.Minus(p))`.

```
PROCEDURE Change
  (READONLY r: T; dw,de,dn,ds: INTEGER): T;
```

If `r` *is empty return* `Empty`*, else return the rectangle* `FromEdges(r.west+dw, r.east+de, r.north+dn, r.south+ds)`.

```
PROCEDURE Inset(READONLY r: T; n: INTEGER): T;
```

Return `Change(r, n, -n, n, -n)`.

```
PROCEDURE Transpose(READONLY r: T; ax := Axis.T.Ver): T;
```

If `r` *is empty or if* `ax = Axis.Hor`*, then return* `r`*, else return* `T{r.north, r.south, r.west, r.east}`.

```
PROCEDURE Join(READONLY r, s: T): T;
```

Return the smallest rectangle containing both `r` *and* `s`.

```
PROCEDURE Meet(READONLY r, s: T): T;
```

Return the largest rectangle contained in both `r` *and* `s`.

```
PROCEDURE HorSize(READONLY r: T): CARDINAL;
```

Return `r.east - r.west`.

```
PROCEDURE VerSize(READONLY r: T): CARDINAL;
```

Return `r.south - r.north`.

```
PROCEDURE Middle(READONLY r: T): Point.T;
```

Return `Point.T{(r.west+r.east) DIV 2, (r.north+r.south) DIV 2}`.

```
PROCEDURE Center(READONLY r: T; READONLY p: Point.T): T;
```

If `r` *is empty then return* `Empty` *else return a rectangle* `s` *such that* `Congruent(r, s)` *and* `Middle(s) = p`.

```
PROCEDURE NorthWest(READONLY r: T): Point.T;
```

Return `Point.T{r.west,r.north}`.

```
PROCEDURE NorthEast(READONLY r: T): Point.T;
```

Return `Point.T{r.east,r.north}`.

```
PROCEDURE SouthWest(READONLY r: T): Point.T;
```

Return `Point.T{r.west,r.south}`.

```
PROCEDURE SouthEast(READONLY r: T): Point.T;
```

Return `Point.T{r.east,r.south}`.

```
PROCEDURE Project(READONLY r: T;
  READONLY p: Point.T): Point.T;
```

Return the element of `r` *that is closest to* `p`. *This is a checked runtime error if* `r` *is empty.*

```
TYPE Partition = ARRAY [0..4] OF T;

PROCEDURE Factor(
  READONLY r, s: T;
  VAR (*out*) f: Partition;
  dh, dv: INTEGER) ;
```

Partition `r` *into* `5` *pieces* `f[0]..f[4]` *where* `f[2] = Meet(r,s)`, *and the other rectangles in* `f` *partition the set difference* `r-s`.

The order of `f` is such that if `i<j` then `f[i]` translated by any positive multiple of `(dh,dv)` doesn't intersect `f[j]`. (Only the signs of `dh` and `dv` affect the order, not their magnitude.)

```
PROCEDURE Mod(READONLY p: Point.T;
  READONLY r: T): Point.T;
```

Return the element of `r` whose distance from `p` in each axis is a multiple of the size of `r` in that axis. This is a checked runtime error if `r` is empty.

```
PROCEDURE IsEmpty(READONLY r: T): BOOLEAN;
```

Return whether `r` is empty.

```
PROCEDURE Member(READONLY p: Point.T;
  READONLY r: T): BOOLEAN;
```

Return whether `p` is in `r`.

```
PROCEDURE Overlap(READONLY r, s: T): BOOLEAN;
```

Return whether `r` and `s` have any element in common.

```
PROCEDURE Subset(READONLY r, s: T): BOOLEAN;
```

Return whether `r` is contained in `s`.

```
PROCEDURE Congruent(READONLY r, s: T): BOOLEAN;
```

Return whether `r` and `s` are congruent, that is, whether they have the same height and width.

```
END Rect.
```

8.5 The Region interface

A `Region.T` represents a set of integer lattice points.

```
INTERFACE Region;

IMPORT Rect, Point, Axis;

TYPE
  T = RECORD r: Rect.T; p: P := NIL END;

  P <: REFANY;
```

If `rg` is a region, then `rg.r` is the smallest rectangle containing all points in `rg`, and `rg.p` is the private representation of the region as a sorted array of disjoint rectangles.

```
CONST
  Empty = T{Rect.Empty, NIL};
```

```
  Full = T{Rect.Full, NIL};
```

```
PROCEDURE FromRect(READONLY r: Rect.T): T;
```
Return the region containing the same points as `r`.

```
PROCEDURE FromRects(READONLY ra: ARRAY OF Rect.T): T;
```
Return the region containing all points in any rectangle of `ra`.

```
PROCEDURE ToRects(READONLY rg: T): REF ARRAY OF Rect.T;
```
Returns a list of disjoint rectangles that partition `rg`.

The call `ToRects(Empty)` produces an array of length zero.

```
PROCEDURE FromPoint(READONLY p: Point.T): T;
```
Return the region containing exactly the point `p`.

```
PROCEDURE BoundingBox(READONLY rg: T): Rect.T;
```
Return the smallest rectangle containing all the points of `rg`*; this is equivalent to* `rg.r`.

```
PROCEDURE Add(READONLY rg: T; READONLY p: Point.T): T;
```
Return the translation of `rg` *by* `p`.

That is, `Add(rg, p)` contains `pt` if and only if `rg` contains `Point.Sub(pt, p)`.

```
PROCEDURE Sub(READONLY rg: T; READONLY p: Point.T): T;
```
Return `Add(rg, Point.Minus(p))`.

```
PROCEDURE AddHV(READONLY rg: T; dh, dv: INTEGER): T;
```
Return `Add(rg, Point.T{dh,dv})`.

```
PROCEDURE Inset(READONLY rg: T; n: INTEGER): T;
```
Return the region inset into `rg` *by* `n`.

That is, if `n` is non-negative, `Inset(rg, n)` contains a point `pt` if all points within distance `n` of `pt` are contained in `rg`. If `n` is non-positive, `Inset(rg, n)` contains a point `pt` if some point within distance `-n` of `pt` is in `rg`. For the purposes of this definition, points `p` and `q` are "within distance `n`" if both `ABS(p.h-q.h)` and `ABS(p.v-q.v)` are at most `n`. (If `n` is zero, both definitions give `Inset(rg, n) = rg`.)

```
PROCEDURE PlaceAxis(READONLY rg: T;
  n: INTEGER; hv: Axis.T): T;
```
Return the retraction of `rg` *by* `n` *along the* `hv` *axis.*

That is, let `rect` equal `Rect.FromSize(1, ABS(n))` if `hv` is `Axis.T.Ver` or `Rect.FromSize(ABS(n), 1)` if `hv` is `Axis.T.Hor`. If `n` is non-negative, then

`PlaceAxis(rg, n, hv)` contains a point `pt` if the rectangle `Rect.Add(pt, rect)` is contained in `rg`. If `n` is negative, then `PlaceAxis(rg, n, hv)` contains a point `pt` if `Rect.Add(pt, rect)` contains some point in `rg`.

```
PROCEDURE Place(READONLY rg: T; h, v: INTEGER): T;
```

Return the retraction of `rg` by `h` along the horizontal axis and by `v` along the vertical axis.

More precisely, `Place(rg, h, v)` is defined by the expression

```
PlaceAxis(PlaceAxis(rg, h, Axis.T.Hor), v, Axis.T.Ver)
```

```
PROCEDURE Join(READONLY rg, rgP: T): T;
```

Return the union of the points in `rg` and `rgP`.

```
PROCEDURE JoinRect(READONLY r: Rect.T;
  READONLY rg: T): T;
```

Return the union of the points in `r` and `rg`.

```
PROCEDURE JoinRegions(READONLY rg: REF ARRAY OF T): T;
```

Return the union of all the regions in `rg`.

```
PROCEDURE Meet(READONLY rg, rgP: T): T;
```

Return the intersection of `rg` and `rgP`.

```
PROCEDURE MeetRect(READONLY r: Rect.T;
  READONLY rg: T): T;
```

Return the intersection of the points in `r` and `rg`.

```
PROCEDURE Difference(READONLY rg, rgP: T): T;
```

Return the set of points in `rg` and not in `rgP`.

```
PROCEDURE SymmetricDifference(READONLY rg, rgP: T): T;
```

Return the set of points in exactly one of `rg` and `rgP`.

```
PROCEDURE MaxSubset(READONLY r: Rect.T;
  READONLY rg: T): Rect.T;
```

Return a large rectangular subset of `rg` containing `r`, or return `Empty` if `r` is not a subset of `rg`.

```
PROCEDURE Equal(READONLY rg, rgP: T): BOOLEAN;
```

Return whether `rg` and `rgP` contain the same points.

```
PROCEDURE IsEmpty(READONLY rg: T): BOOLEAN;
```

Return whether `rg` is empty.

```
PROCEDURE IsRect(READONLY rg: T): BOOLEAN;
```

Return whether `rg` *is a rectangle, that is, whether it contains all the points in its bounding box.*

```
PROCEDURE Member(READONLY p: Point.T;
  READONLY rg: T): BOOLEAN;
```

Return whether `p` *is in* `rg`*.*

```
PROCEDURE SubsetRect(READONLY r: Rect.T;
  READONLY rg: T): BOOLEAN;
```

Return whether `r` *is contained in* `rg`*.*

```
PROCEDURE Subset(READONLY rg, rgP: T): BOOLEAN;
```

Return whether `rg` *is contained in* `rgP`*.*

```
PROCEDURE OverlapRect(READONLY r: Rect.T;
  READONLY rg: T): BOOLEAN;
```

Return whether `r` *and* `rg` *have any point in common.*

```
PROCEDURE Overlap(READONLY rg, rgP: T): BOOLEAN;
```

Return whether `rg` *and* `rgP` *have any point in common.*

```
END Region.
```

8.6 The Path interface

A `Path.T` is a sequence of straight and curved line segments, suitable for stroking or filling.

A *segment* is a directed arc in the Cartesian plane determined by two cubic polynomials `h(t)`, `v(t)`, where `t` ranges over the interval of real numbers `[0, 1]`. The segment is said to *start* at `(h(0), v(0))` and *end* at `(h(1), v(1))`. If `h` and `v` are linear functions of `t`, then the segment is *linear*: it consists of a line segment. If `h` and `v` are constant functions of `t`, then the segment is *degenerate*: it consists of a single point.

The segments of a path are grouped into contiguous *subpaths*, which can be *open* or *closed*. Within a subpath, each segment starts where the previous segment ends. In a closed subpath, the last segment ends where the first segment starts. (This may also happen for an open subpath, but this coincidence does not make the subpath closed.)

The *current point* of a path is the endpoint of the last segment of its last subpath, assuming this subpath is open. If the path is empty or if the last subpath is closed, the current point is undefined.

```
INTERFACE Path;
```

```
IMPORT Point;

TYPE T <: ROOT;
```

The call `NEW(Path.T)` creates an empty path.

```
PROCEDURE Reset(path: T);
```

Set `path` to be empty.

```
PROCEDURE MoveTo(path: T; READONLY p: Point.T);
```

Extend `path` with a new degenerate segment that starts and ends at `p`. This begins a new subpath.

```
PROCEDURE LineTo(path: T; READONLY p: Point.T);
```

Extend `path` with a linear segment that starts at its current point and ends at `p`.

```
PROCEDURE CurveTo(path: T; READONLY q, r, s: Point.T);
```

Extend `path` with a curved segment that starts at its current point and ends at `s`.

`CurveTo` adds a curve that starts from the current point of `path` in the direction of `q`, and ends at `s` coming from the direction of `r`. More precisely, let `p` be the current point of `path` and let `h(t)` and `v(t)` be the cubic polynomials such that

```
(h(0), v(0)) = p
(h(1), v(1)) = s
(h'(0), v'(0)) = 3 * (q - p)
(h'(1), v'(1)) = 3 * (s - r)
```

(where the primes denote differentiation with respect to `t`). Then `CurveTo` adds the segment `(h(t), v(t))` for `t` between zero and one. This is called the *Bezier* arc determined by `p`, `q`, `r`, and `s`.

```
PROCEDURE Close(path: T);
```

Add a linear segment to create a closed loop in `path`.

More precisely, let `p` be the current point of `path`, and let `q` be last point of `path` that was added by a call to `MoveTo` (thus `q` is the startpoint of the first segment of the last subpath of `path`). `Close` adds a linear segment from `p` to `q` and marks the sequence of segments from `q` to the end of the path as a closed subpath.

```
PROCEDURE IsEmpty(p: T): BOOLEAN;
```

Returns `TRUE` if `p` is empty.

```
PROCEDURE IsClosed(p: T): BOOLEAN;
```

Returns `TRUE` if `p` is empty or the last subpath of `p` is closed.

```
PROCEDURE CurrentPoint(p: T): Point.T;
```

Returns the current point of `p`.

`LineTo`, `CurveTo`, `Close`, and `CurrentPoint` are checked runtime errors if the path has no current point.

```
EXCEPTION Malformed;
```

The `Malformed` exception is raised when a procedure detects a malformed path.

```
PROCEDURE Translate(p: T; READONLY delta: Point.T): T
  RAISES {Malformed};
```

The result of translating `p` *by* `delta`.

```
TYPE
  MapObject = OBJECT METHODS
    move(READONLY pt: Point.T);
    line(READONLY pt1, pt2: Point.T);
    close(READONLY pt1, pt2: Point.T);
    curve(READONLY pt1, pt2, pt3, pt4: Point.T)
  END;

PROCEDURE Map(path: T; map: MapObject)
  RAISES {Malformed};
```

Apply the appropriate method of `map` *to each segment of* `path`.

That is, for each segment `s` of `path`, in order, `Map` excecutes the following:

```
IF s is a linear segment (p, q) THEN
  IF s was generated by MoveTo THEN
     (* p = q *)
     map.move(p)
  ELSIF s was generated by LineTo THEN
    map.line(p, q)
  ELSE (* s was generated by Close *)
    map.close(p, q)
  END
ELSE (* s is a curved segment (p, q, r, s) *)
  map.curve(p, q, r, s)
END
```

`Map` raises the exception if it is passed a malformed path.

```
PROCEDURE Copy(p: T): T;
```

Returns a newly allocated path with the same contents as `p`.

```
PROCEDURE Flatten(p: T): T RAISES {Malformed};
```

Return a path like `p` *but with curved segments replaced by polygonal approximations.*

```
END Path.
```

8.7 The Trapezoid interface

A `Trapezoid.T` represents a set of points lying in a quadrilateral whose north and south edges are horizontal and whose west and east edges have arbitrary non-horizontal slopes. For example, a diagonal line can be represented as a tall skinny trapezoid.

```
INTERFACE Trapezoid;

IMPORT Point;

TYPE
  T = RECORD
    vlo, vhi: INTEGER;
    m1, m2: Rational;
    p1, p2: Point.T;
  END;
  Rational = RECORD n, d: INTEGER END;
```

For a trapezoid `tr`,

- `tr.vlo` and `tr.vhi` are the `v` coordinates of its north and south edges, respectively;
- `tr.m1` and `tr.m2` are the slopes of its west and east edges, respectively, as `(delta v) / (delta h)`. A denominator of zero represents an infinite slope; i.e., a vertical edge. A numerator of zero is illegal.
- `tr.p1` and `tr.p2` are points on the infinite lines that extend the west and east edges, respectively.

Trapezoids are closed on the north and west edges, open on the south and east edges, closed on the northwest corner, and open on the other corners.

A `Rational q` represents the rational number `q.n/q.d`.

```
END Trapezoid.
```

9 Implementing your own splits

This section defines the information needed to implement new VBT classes, especially split classes and filter classes. Most VBT leaf classes can get by with the information in the VBT interface.

Events that flow down the tree of VBTs, like mouse clicks and repaint events, are relayed via the methods described in the VBT interface. To relay the event, the parent method recursively activates the appropriate child method. However, the parent should not activate the child method directly; it should use one of the procedures in this interface to activate the child method indirectly.

A typical down method of a VBT v has the form v.method(args) and has locking level VBT.mu or VBT.mu.v, as explained in the VBT interface.

Information also flows up the tree of VBTs; for example, painting commands and commands to set the cursor shape and cage. This information is also relayed via methods, which we call "up" methods. For example, when a child ch of a parent p changes its cursor, Trestle notifies the parent by calling the method p.setcursor(ch). This method is expected to read the child's cursor and take appropriate action based on the class of the split.

A typical up method call has the form parent.method(child, args) and has locking level LL.sup = child.

Notice that the up methods come from the parent, not the child. This is convenient, since it is the parent that defines the class of split. However, it means that if a VBT's parent is NIL, then there are no up methods, so that painting on it (for example) is a noop. This produces a wrinkle at the address space boundary: the VBT that we call the root of the tree actually has a parent whose only purpose is to supply up methods for communicating across the address space boundary.

The procedures in the split interface for inserting, deleting, and enumerating children are also implemented via methods, which we call "split" methods. For example, Split.Succ(v, ch) is implemented by calling v.succ(ch).

9.1 The VBTClass interface

The VBTClass interface specifies the up methods, the split methods, and the wrapper procedures by which a parent activates a child's down methods.

In general, to implement a split or filter you override the down methods, up methods, and split methods of the parent. However, usually you will be able to inherit the majority of the methods from existing classes, and only have to override a few of them. We mention several groups of methods that in most cases you will want to inherit rather than reimplement.

The two down methods

```
VBT.Split.mouse
VBT.Split.position
```

together with the two up methods

```
VBT.Split.setcage
VBT.Split.setcursor
```

conspire to implement the mouse-cage semantics described in the VBT interface for delivering mouse clicks and cursor positions and for setting the cursor shape. They work for any VBT.Split, and there is almost never any reason to override them. As a far-fetched example of when you would override them, imagine a filter that converts shifted left button clicks to right button clicks.

Although you probably won't want to override these methods, you will have to help them a bit. They cache the results of the locate method, and therefore require that you call VBTClass.LocateChanged whenever the geometry of your split changes in a way that affects the locate method.

The up methods

```
VBT.Split.acquire
VBT.Split.release
VBT.Split.put
VBT.Split.forge
VBT.Split.readUp
VBT.Split.writeUp
```

implement the event-time semantics described in the VBT interface. They simply recurse up the tree of VBTs. At the root the recursive calls reach a VBT in which these methods are overridden to make the appropriate X calls. There is rarely any reason to override these methods. As an example of when you might want to override them, imagine keeping track of which VBT in your application last held the keyboard focus. You could do this by introducing a filter whose acquire method recorded the information before recursing on the parent.

Keystrokes and miscellaneous codes can skip levels of the tree when they are delivered. For example, associated with each top-level window is a filter much like the one just described, which keeps track of which of its decendants are selection owners. This filter forwards keystrokes and lost codes directly to the appropriate owner, bypassing the intermediate windows in the tree.

The up methods

```
VBT.Split.paintbatch
VBT.Split.capture
VBT.Split.sync
```

implement painting, painting synchronization, and screen capture. The sync and capture methods recurse up the tree in the obvious way. The paintbatch method also recurses up the tree, but in a less obvious way.

It would be too inefficient to call a method for every painting command; therefore the class-independent painting code groups painting commands into batches and hands them to the method a batch at a time. For example, the paintbatch method of a ZSplit clips the batch of painting commands to the visible portion of the child's domain and then executes the clipped operations on itself.

Painting on the vast majority of VBTs can be implemented simply by clipping to their domain and then relaying the painting to their parent. To speed up this common case, every VBT has a *short-circuit* bit. If this bit is set then Trestle doesn't call the VBT's `paintbatch` method at all; it just clips to the VBT's domain and paints on its parent. Typically the only VBTs whose short-circuit bits are not set are the root VBT and those `ZSplit` children that are overlapped by other children or that extend outside the parent's domain.

If the short-circuit bits are set on all the VBTs from `v` to the root, then the class-independent painting code will relay batches of painting commands from `v` to the root without activating any methods. The `paintbatch` method at the root translates the batch of painting commands into the appropriate X operations.

The default method `VBT.Split.paintbatch` sets the short-circuit bit and recurses on the parent. In the unlikely event that you want to override this method, the interfaces `Batch`, `BatchUtil`, and `PaintPrivate` define the representation of painting commands in batches. You could for example overriding the paintbatch method to implement a class of VBT that paints into a raw pixmap in your address space.

To speed up painting, Trestle does not rely on garbage collection for paintbatches: you must free them explicitly.

You almost never need to implement the split methods `succ`, `pred`, `move`, `nth`, `index`, and `locate`; on the other hand you must be careful to inherit them from the right place. There are two main subtypes of `VBT.Split`, filters and "proper" splits, and they have different suites of split methods. The implementations of the split methods for filters are

```
Filter.T.succ
Filter.T.pred
Filter.T.move
Filter.T.nth
Filter.T.index
Filter.T.locate
```

These are all quite trivial procedures, since a filter has at most one child. If you declare a split as a subtype of `Filter.T`, you inherit these methods automatically.

Most proper splits are subtypes of `ProperSplit.T`, which keeps the children in a doubly-linked list. For example, `ZSplits`, `HVSplits`, `TSplits`, and `PackSplits` are all subtypes of `ProperSplit.T`. The methods

```
ProperSplit.T.succ
ProperSplit.T.pred
ProperSplit.T.move
ProperSplit.T.nth
ProperSplit.T.index
ProperSplit.T.locate
```

implement the split methods using the doubly-linked list. If you declare a split as a subtype of `ProperSplit.T`, you inherit these methods automatically.

```
INTERFACE VBTClass;

IMPORT VBT, Trestle, Axis, Point, Rect, Region,
  ScrnCursor, ScrnPixmap, Cursor, Batch;
```

Before we get to the up methods and the split methods, there is more to be revealed about `VBT`s in general:

```
REVEAL
  VBT.Prefix <: Prefix;

TYPE Prefix =
  MUTEX OBJECT <* LL >= {VBT.mu, SELF} *>
    parent: VBT.Split := NIL;
    upRef: ROOT := NIL;
    domain: Rect.T := Rect.Empty;
    st: VBT.ScreenType := NIL;
  METHODS <* LL.sup = SELF *>
    getcursor(): ScrnCursor.T;
    <* LL.sup = VBT.mu *>
    axisOrder(): Axis.T;
  END;
```

From `VBT.Prefix <: Prefix` it follows `VBT.T <: Prefix`; hence every `VBT` is a `MUTEX` object, and has the above fields and methods. The complete revelation for the type `VBT.T` is private to Trestle.

The fields `v.parent`, `v.domain`, and `v.st` record `v`'s parent, domain, and screentype.

The object `v.upRef` is used by the methods of `v.parent` to store information specific to the child `v`. For example, if `v.parent` is a `ZSplit`, then `v.upRef` contains a region representing the visible part of `v`, pointers to the children before and after `v`, and other information. In a filter, `v.upRef` is usually `NIL`, since when there is only one child, all the state can be stored in data fields directly in the parent object.

If `v.parent` is `NIL`, then so is `v.upRef`.

The locking level comment on the data fields means that in order to write one of the fields `v.parent`, `v.upRef`, `v.domain`, or `v.st`, a thread must have both `VBT.mu` and `v` locked. Consequently, in order to read one of the fields, a thread must have either `VBT.mu` (or a share of `VBT.mu`) or `v` locked. Thus the fields can be read either by up methods or by down methods.

The call `v.getcursor()` returns the cursor that should be displayed over `v`; that is, the cursor that was called `GetCursor(v)` in the `VBT` interface. It is almost never necessary to override the `getcursor` method, since leaves and splits have suitable default methods.

The `axisOrder` method determines whether it is preferable to fix a `VBT`'s height first or its width first. For example, a horizontal packsplit would rather have its width fixed before its range of heights is queried, since its height depends on its width. In general, if `v`'s size range in axis `ax` affects its size range in the other axis (and not vice-versa), then `v.axisOrder()` should return `ax`. The default is to return `Axis.T.Hor`.

Next we come to the specifications of the split methods and the up methods:

```
REVEAL VBT.Split <: Public;

TYPE Public = VBT.Leaf OBJECT
  METHODS

   (* The split methods *)

    <* LL >= {VBT.mu, SELF, ch} *>
    beChild(ch: VBT.T);
    <* LL.sup = VBT.mu *>
    replace(ch, new: VBT.T);
    insert(pred, new: VBT.T);
    move(pred, ch: VBT.T);
    locate(READONLY pt: Point.T;
      VAR (*OUT*) r: Rect.T): VBT.T;
    <* LL >= {VBT.mu} *>
    succ(ch: VBT.T): VBT.T;
    pred(ch: VBT.T): VBT.T;
    nth(n: CARDINAL): VBT.T;
    index(ch: VBT.T): CARDINAL;

   (* The up methods *)

    <* LL.sup = ch *>
    setcage(ch: VBT.T);
    setcursor(ch: VBT.T);
    paintbatch(ch: VBT.T; b: Batch.T);
    sync(ch: VBT.T);
    capture(ch: VBT.T; READONLY rect: Rect.T;
      VAR (*out*) br: Region.T) : ScrnPixmap.T;
    screenOf(ch: VBT.T; READONLY pt: Point.T)
      : Trestle.ScreenOfRec;
    <* LL.sup < SELF AND LL >= {ch, VBT.mu.ch} *>
    newShape(ch: VBT.T);
    <* LL.sup = ch *>
    acquire(ch: VBT.T; w: VBT.T; s: VBT.Selection;
      ts: VBT.TimeStamp) RAISES {VBT.Error};
    release(ch: VBT.T; w: VBT.T; s: VBT.Selection);
```

```
    put(ch: VBT.T; w: VBT.T; s: VBT.Selection;
      ts: VBT.TimeStamp; type: VBT.MiscCodeType;
      READONLY detail := VBT.NullDetail)
      RAISES {VBT.Error};
    forge(ch: VBT.T; w: VBT.T; type: VBT.MiscCodeType;
      READONLY detail := VBT.NullDetail)
      RAISES {VBT.Error};
    <* LL.sup <= VBT.mu *>
    readUp(ch: VBT.T; w: VBT.T; s: VBT.Selection;
      ts: VBT.TimeStamp; tc: CARDINAL) : VBT.Value
      RAISES {VBT.Error};
    writeUp(ch: VBT.T; w: VBT.T; s: VBT.Selection;
      ts: VBT.TimeStamp; val: VBT.Value; tc: CARDINAL)
      RAISES {VBT.Error};
  END;
```

Notice that a `VBT.Split` is a subtype of a `VBT.Leaf`. That is, every `VBT.Split` is also a `VBT.Leaf`, and therefore the painting operations in the `VBT` interface can be applied to splits. This fact is revealed here rather than in the `VBT` interface to prevent clients of `VBT` from accidentally painting on splits. To do so is almost certainly a mistake—it is the responsibility of the split's implementation to paint on the parent as necessary to keep its screen up to date.

9.1.1 Specifications of the split methods

The first group of methods implement the behavior in the `Split` interface:

The method call `v.beChild(ch)` initializes `ch.upRef` as appropriate for a child of `v`. The method can assume that `ch` is non-nil and has the same screentype as `v`. When the method is called, `LL >= {VBT.mu, v, ch}`.

When declaring a subtype `ST` of a split type `S`, the `beChild` method for `ST` will ordinarily call `S.beChild(v, ch)`, which in turn will call `S`'s supertype's `beChild` method, and so on. Only one of the methods should allocate the `upRef`, but all of them may initialize different parts of it. Two rules make this work. First, the type of the `upRef` for children of `ST` splits should be a subtype of the type of the `upRef` for children of `S` splits. Second, if a `beChild` method finds `ch.upRef` is `NIL` and `NIL` is not appropriate for the type, the method should allocate `ch.upRef`; otherwise it should narrow `ch.upRef` to the appropriate type and initialize it.

For example, `HVSplit.T` is a subtype of `ProperSplit.T`. Hidden in the `HVSplit` module is a type `HVSplit.Child`, which represents the per-child information needed by an `HVSplit`. The type `HVSplit.Child` is a subtype of `ProperSplit.Child`. The method `HVSplit.beChild(hv, ch)` allocates a new `HVSplit.Child`, stores it in `ch.upRef`, initializes the part of it that is specific to `HVSplit`, and then calls `ProperSplit.beChild(hv, ch)`, which initializes

the part of `ch.upRef` that is common to all proper splits, and then calls its supertype's `beChild` method, and so on.

The chain of calls eventually ends with a call to `VBT.Split.beChild`, which causes an error if `ch` is not detached or if `ch`'s screentype differs from `v`, and otherwise sets `ch.parent` to `v` and marks `v` for redisplay.

The method call `v.replace(ch, new)` simply implements the operation `Split.Replace(v, ch, new)`, and the call `v.replace(ch, NIL)` implements `Split.Delete(v, ch)`. Before calling the method, the generic code in `Split` marks `v` for redisplay, checks that `ch` is a child of `v` and that `new` is detached, and rescreens `new` to the screentype of `v`.

Similarly, the method call `v.insert(pred, new)` implements the operation `Split.Insert(v, pred, new)`. Before calling the method, the generic code in `Split` marks `v` for redisplay, checks that `pred` is `NIL` or a child of `v` and that `new` is detached, and rescreens `new` to the screentype of `v`. A split that can only contain a limited number of children may detach and discard the previous child to implement `insert`.

The call `v.move(pred, ch)` implements `Split.Move(v, pred, ch)`. Before calling the method, the generic code verifies that `pred` and `ch` are children of `v` (or `NIL`, in the case of `pred`), and avoids the call if `pred = ch` or `v.succ(pred) = ch`.

When the `replace`, `insert`, or `move` method is called, `LL.sup = VBT.mu`. The default methods are equal to `NIL`; so every split class must arrange to override these methods, usually by inheriting them from `Filter` or from `ProperSplit`.

The method calls `v.succ(ch)`, `v.pred(ch)`, `v.nth(n)`, and `v.index(ch)` implement the corresponding operations in the `Split` interface. In all cases, `LL >= {VBT.mu}`.

The default method `VBT.Split.succ` is `NIL`; so every split class must arrange to override the method, usually by inheriting them from `Filter` or from `ProperSplit`. The default methods `VBT.Split.pred`, `VBT.Split.nth`, and `VBT.Split.index` are implemented by repeatedly calling the `succ` method.

The method call `v.locate(pt, r)` returns the child of `v` that controls the position `pt`, or `NIL` if there is no such child. The method also sets `r` to a rectangle containing `pt` such that for all points `q` in the meet of `r` and `domain(v)`, `v.locate(q, ...)` would return the same result as `v.locate(pt, ...)`. The split implementation is expected to make `r` as large as possible, so that clients can avoid calling `locate` unnecessarily. When the method is called, `pt` will be in `domain(v)`. When the locate method is called, `LL.sup = VBT.mu`.

If `v` inherits the `mouse`, `position`, `setcursor`, or `setcage` methods from `VBT.Split`, then you must call `LocateChanged(v)` whenever any operation on the split invalidates a rectangle-child pair returned previously by `v.locate`:

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

Clear any cached results of the `locate` method.

The default method `VBT.Split.locate(v, pt, r)` enumerates `v`'s children in `succ` order and returns the first child `ch` whose domain contains `pt`. It sets `r` to a maximal rectangle that lies inside the domain of `ch` and outside the domains of all preceding children. If no child contains `pt`, it returns `NIL` and sets `r` to a maximal rectangle that lies inside the domain of `v` and outside the domains of all its children. This is suitable if the children don't overlap or if whenever two children overlap, the top one appears earlier in `succ` order.

9.1.2 Specifications of the up methods

So much for the split methods; here now are the specifications of the up methods. In all cases, `ch` is a child of `v`.

The method call `v.setcage(ch)` is called by the system whenever `ch`'s cage is changed. It is called with `LL.sup = ch`. The default method implements the behavior described in the `VBT` interface.

The method call `v.setcursor(ch)` is called by the system whenever the result of `ch.getcursor()` might have changed. It is called with `LL.sup = ch`. The default method implements the behavior described in the `VBT` interface.

The method call `v.paintbatch(ch, b)` is called to paint the batch `b` of painting commands on `v`'s child `ch`. The procedure can assume that the batch is not empty and that its clipping rectangle is a subset of `ch`'s domain. It is responsible for ensuring that `b` is eventually freed, which can be achieved by calling passing `b` to `Batch.Free` or by passing `b` to another paintbatch method, which will inherit the obligation to free the batch. A `paintbatch` method is allowed to modify the batch. The default method clips the batch to `ch`'s domain, paints the batch on the parent, and sets `ch`'s shortcircuit bit. The method is called with `LL.sup = ch`.

The method call `v.sync(ch)` implements `VBT.Sync(ch)`. When the method is called, `ch`'s batch will have been forced. The default method simply applies `VBT.Sync` to the parent. When the method is called, `ch`'s batch is `NIL` and `LL.sup = ch`.

The method call `v.capture(ch, r, br)` implements `VBT.Capture(ch, r, br)`. The default method recurses on the parent. When the method is called, `ch`'s batch is `NIL`, `r` is a subset of `ch`'s domain, and `LL.sup = ch`.

The method call `v.screenOf(ch, pt)` implements `Trestle.ScreenOf(ch, pt)`. The default method recurses on the parent. When the method is called, `LL.sup = ch`.

The method call `v.newShape(ch)` signals that `ch`'s size range, preferred size, or axis order may have changed. The default recurses on the parent. When the method is called, `LL.sup < v AND LL >= {ch, VBT.mu.ch}`.

The remaining methods implement event-time operations for a descendent (not necessarily a direct child) of the window `v`. In all cases, `ch` is a child of `v` and `w` is a descendant of `ch`.

The `acquire`, `release`, `put`, and `forge` methods implement the corresponding procedures from the `VBT` interface. For example, `v.put(ch, w, s, ts, cd)` im-

plements `VBT.Put(w, s, ts, cd.type, cd.detail)`. When these methods are called, `LL.sup = ch`.

Similarly, the `readUp` and `writeUp` methods implement the procedures `VBT.Read` and `VBT.Write`. When these methods are called, `LL.sup <= VBT.mu`.

9.1.3 Getting and setting the state of a VBT

```
PROCEDURE Cage(v: VBT.T): VBT.Cage; <* LL >= {v} *>
```

Return v's cage.

```
TYPE
  VBTCageType = {Gone, Everywhere, Rectangle};

PROCEDURE CageType(v: VBT.T): VBTCageType;
<* LL >= {v} *>
```

Return v's cage's type.

`CageType(v)` returns `Gone` if `Cage(v) = VBT.GoneCage`, `Everywhere` if `Cage(v) = VBT.EverywhereCage`, and `Rectangle` otherwise. It is more efficient than `Cage`.

```
PROCEDURE GetCursor(v: VBT.T): Cursor.T;
<* LL >= {v} *>
```

Return `cursor(v)`.

```
PROCEDURE SetShortCircuit(v: VBT.T); <* LL >= {v} *>
```

Set the short-circuit property of `v`.

```
PROCEDURE ClearShortCircuit(v: VBT.T); <* LL >= {v} *>
```

Clear the short-ciruit propery of `v`.

If `v`'s short-circuit property is on, painting on `v` will be implemented by clipping to its domain and painting on its parent.

The next three procedures are equivalent to the corresponding procedures in `VBT`, except they have a different locking level:

```
PROCEDURE PutProp(v: VBT.T; ref: REFANY);
<* LL >= {v} *>

PROCEDURE GetProp(v: VBT.T; tc: INTEGER): REFANY;
<* LL >= {v} *>

PROCEDURE RemProp(v: VBT.T; tc: INTEGER);
<* LL >= {v} *>
```

In implementing a split it is sometimes necessary to read a child's bad region; in which case the following procedure is useful:

```
PROCEDURE GetBadRegion(v: VBT.T): Region.T;
<* LL >= {v} *>
```

Return v's bad region; that is, the join of `bad(v)` *and* `exposed(v)`*.*

For the convenience of split implementors, every `VBT` has a "newshape" bit which is set by a call to `VBT.NewShape`. For example, the redisplay or shape method of a split can test these bits to determine which of its children have new shapes.

```
PROCEDURE HasNewShape(v: VBT.T): BOOLEAN;
<* LL.sup < v *>
```

Return the value of `v`*'s newshape bit.*

```
PROCEDURE ClearNewShape(v: VBT.T); <* LL.sup < v *>
```

Clear `v`*'s* `newshape` *bit.*

9.1.4 Procedures for activating the down methods of a VBT

```
PROCEDURE Reshape(
    v: VBT.T;
    READONLY new, saved: Rect.T);
<* LL.sup >= VBT.mu.v AND LL.sup <= VBT.mu *>
```

Prepare for and call `v`*'s* `reshape` *method.*

That is, `Reshape` changes `v.domain` and then schedules a call to

```
v.reshape(VBT.ReshapeRec{v.domain, new, saved})
```

It should always be called instead of a direct call to the method, since it establishes essential internal invariants before calling the method. The bits in the `saved` argument must remain valid until the method returns. It is all right for `saved` to be larger than `v`'s old domain; `Reshape` will clip it to `v`'s old domain before calling the method. It is illegal to reshape a detached `VBT` to have a non-empty domain.

For example, the `reshape` method of `BorderedVBT` uses `VBTClass.Reshape` to reshape its child.

```
PROCEDURE Rescreen(v: VBT.T; st: VBT.ScreenType);
<* LL.sup >= VBT.mu.v AND LL.sup <= VBT.mu *>
```

Prepare for and call `v`*'s* `rescreen` *method.*

That is, `Rescreen` executes

```
prev := v.domain;
v.domain := Rect.Empty;
v.st := st;
v.rescreen(VBT.RescreenRec{prev, st}).
```

For example, to determine how large a menu `m` would be if it were inserted into a `ZSplit` `z`, you can't simply call `GetShapes(m)`, since in general the screentype of `m`

could be different from the screentype of `z`, and the shape can depend on the screentype. But you can call `VBTClass.Rescreen(m, z.st)` followed by `GetShapes(m)`.

```
PROCEDURE Repaint(v: VBT.T; READONLY badR: Region.T);
<* LL.sup >= VBT.mu.v AND LL.sup <= VBT.mu *>
```

Join `badR` into `v`'s bad region and then prepare for and call `v`'s repaint method.

```
PROCEDURE Position(v: VBT.T;
  READONLY cd: VBT.PositionRec);
<* LL.sup = VBT.mu *>
```

Prepare for and call `v`'s `position` method.

```
PROCEDURE Key(v: VBT.T; READONLY cd: VBT.KeyRec);
<* LL.sup = VBT.mu *>
```

Prepare for and call `v`'s `key` method.

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

Prepare for and call `v`'s `mouse` method.

```
PROCEDURE Misc(v: VBT.T; READONLY cd: VBT.MiscRec);
<* LL.sup = VBT.mu *>
```

Prepare for and call `v`'s `misc` method.

The following two procedures schedule calls to the down methods without making the calls synchronously. They are useful when you hold too many locks to call a down method directly. For example, when a `ZSplit` child scrolls bits that are obscured, the locking level of the `paintbatch` method precludes calling the `repaint` method directly; but a call can be scheduled with `ForceRepaint`.

```
PROCEDURE ForceEscape(v: VBT.T); <* LL.sup >= {v} *>
```

Enqueue a cage escape to `gone` for delivery to `v`.

```
PROCEDURE ForceRepaint(v: VBT.T;
  READONLY rgn: Region.T; deliver := TRUE);
<* LL.sup >= {v} *>
```

Join `rgn` into `v`'s bad region, and possibly schedule a call to `v`'s repaint method.

`VBTClass.ForceRepaint` is like `VBT.ForceRepaint`, except that it has a different locking level, and if `deliver` is `FALSE` then no thread will be forked to deliver the bad region—in this case the caller has the obligation to deliver the bad region soon, either by calling `ForceRepaint` with `deliver = TRUE`, or by calling `Repaint`.

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

If `v` is marked for redisplay, then unmark it and prepare for and call `v.redisplay()`.

```
PROCEDURE GetShape(v: VBT.T; ax: Axis.T; n: CARDINAL;
  clearNewShape := TRUE): VBT.SizeRange;
<* LL.sup >= VBT.mu.v AND LL.sup <= VBT.mu *>
```

Prepare for and call `v`'s `shape` method.

`GetShape` causes a checked runtime error if the result of the shape method is invalid. If `clearNewShape` is `TRUE`, `GetShape` calls `ClearNewShape(v)` before it calls the method.

```
PROCEDURE GetShapes(v: VBT.T; clearNewShape := TRUE):
  ARRAY Axis.T OF VBT.SizeRange;
<* LL.sup >= VBT.mu.v AND LL.sup <= VBT.mu *>
```

Return the shapes of `v` in both axes.

`GetShapes` calls the shape method of `v` in each axis, using the order determined by `v.axisOrder()`, and returns the array of the resulting size ranges. If `clearNewShape` is `TRUE`, `GetShapes` calls `ClearNewShape(v)` before it calls the method.

`GetShapes` is convenient if both the height and width preferences of the child can be accomodated—for example, when inserting a top level window or `ZSplit` child.

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

Set `v.parent` and `v.upRef` to `NIL`; set `v`'s domain to empty, enqueue a reshape to empty, and clear `v`'s shortcircuit bit.

9.1.5 Procedures for activating the up methods of a VBT

The following six procedures are like the corresponding procedures in the `VBT` interface, except that they have a different locking level:

```
PROCEDURE SetCage(v: VBT.T; READONLY cg: VBT.Cage);
<* LL.sup = v *>

PROCEDURE SetCursor(v: VBT.T; cs: Cursor.T);
<* LL.sup = v *>

PROCEDURE Acquire(
    v: VBT.T;
    s: VBT.Selection;
    t: VBT.TimeStamp)
  RAISES {VBT.Error}; <* LL.sup = v *>

PROCEDURE Release(v: VBT.T; s: VBT.Selection);
<* LL.sup = v *>

PROCEDURE Put(
    v: VBT.T;
    s: VBT.Selection;
    t: VBT.TimeStamp;
```

```
    type: VBT.MiscCodeType;
    READONLY detail := VBT.NullDetail)
  RAISES {VBT.Error};
<* LL.sup = v *>

PROCEDURE Forge(
    v: VBT.T;
    type: VBT.MiscCodeType;
    READONLY detail := VBT.NullDetail)
  RAISES {VBT.Error};
<* LL.sup = v *>
```

Finally, here is a procedure for executing a batch of painting commands on a `VBT`:

```
PROCEDURE PaintBatch(v: VBT.T; VAR b: Batch.T);
<* LL.sup < v *>
```

Execute the batch `b` of painting commands on `v`, free `b`, and set `b` to `NIL`.

The interpretation of `b` is described in the `Batch` and `PaintPrivate` interfaces. If `b.clipped` is erroneously set to `TRUE`, then `PaintBatch` may execute the batched painting commands without clipping them to `b.clip`, but it will not paint outside `v`'s domain.

```
END VBTClass.
```

9.2 The FilterClass interface

The `FilterClass` interface reveals the representation of a filter. If you are implementing a subtype of `Filter.T`, you can import `FilterClass` to gain access to the child field.

```
INTERFACE FilterClass;

IMPORT Filter, Split, VBT;

REVEAL Filter.T <: Public;

TYPE Public =
  Filter.Public OBJECT <* LL >= {SELF, VBT.mu} *>
    ch: VBT.T
  END;
```

A filter `f` is a split with the single child `f.ch`, or with no children if `f.ch=NIL`.

The `beChild` method initializes `ch` and calls `Split.T.beChild`. The `succ`, `pred`, `nth`, `index`, and `locate` methods use the `ch` field in the obvious way. The `misc`, `key`, `read`, `write`, `reshape`, `shape`, and `axisOrder` methods forward to the child.

```
END FilterClass.
```

9.3 The ProperSplit interface

A `ProperSplit.T` is a type of `VBT.Split` that contains a circularly-linked list of its children. All of Trestle's built-in splits that are not filters are subclasses of `ProperSplit`.

```
INTERFACE ProperSplit;

IMPORT VBT, VBTClass, Split;

TYPE
  T <: Public;
  Public = VBT.Split OBJECT
    <* LL >= {SELF, VBT.mu} *>
    lastChild: Child := NIL
  END;
  Child = OBJECT
    <* LL >= {SELF.ch.parent, VBT.mu} *>
    pred, succ: Child := NIL;
    ch: VBT.T
  END;
```

If `ch` is a child of a `ProperSplit.T`, then `ch.upRef` must be of type `ProperSplit.Child`, and `ch.upRef.ch` must equal `ch`. The `succ` and `pred` links represent a doubly-linked list of the children. The `succ` links are circular; the `pred` links are linear. The parent's `lastChild` field is is `NIL` if there are no children; otherwise it points to the last child in `succ` order.

The locking level comments imply that to write any of the links, a thread must have both `VBT.mu` and the parent locked.

If `v` is a `T`, the call `v.beChild(ch)` sets `ch.upref` to `NEW(Child)` if it is `NIL`. In any case it sets `ch.upref.ch := ch` and calls `VBT.Split.beChild(v, ch)`.

The following procedures are useful for implementing subtypes of `ProperSplit.T`:

```
PROCEDURE Insert(v: T; pred: Child; newch: VBT.T);
<* LL >= {VBT.mu, v, newch} *>
```

Insert `newch` as a new child after `pred`, and mark `v` for redisplay.

The child `newch` must be detached and of the appropriate screentype. It can be `NIL` to indicate insertion at the head of the list. `Insert` calls the `beChild` method of `newCh`.

```
PROCEDURE PreInsert(v: T; pred, ch: VBT.T): Child
  RAISES {Split.NotAChild}; <* LL.sup = VBT.mu *>
```

Rescreen `ch` to have `v`'s screentype (if necessary), cause a checked runtime error if `ch` is attached, raise `Split.NotAChild` if `pred` is non-nil and not a child of `v`, and finally return `pred.upRef`, or `NIL` if `pred` is `NIL`.

```
PROCEDURE Move(v: T; pred, ch: Child);
<* LL >= {VBT.mu, v} *>
```

Move `ch` *in the list of children so that it follows* `pred` *and mark* `v` *for redisplay.*

```
PROCEDURE Delete(v: T; ch: Child);
<* LL >= {VBT.mu} AND LL.sup < v  *>
```

Remove `ch` *from the list of children, detach* `ch.ch`*, and mark* `v` *for redisplay.*

```
END ProperSplit.
```

10 Implementing your own painting procedures

10.1 The Batch interface

A `Batch.T` is a data structure containing a sequence of `VBT` painting commands. Batches are untraced: they must be explicitly allocated and freed using the procedures in this interface.

```
INTERFACE Batch;

IMPORT Word;

TYPE T <: ADDRESS;

PROCEDURE New(len: INTEGER := -1): T;
```
Allocate a batch containing at least `len` `Word.Ts`.

If `len = -1`, the number of `Word.T`s in the result will be `VBTTuning.BatchSize`. Initially the clip and scroll source are `Rect.Empty`.

```
PROCEDURE Free(VAR ba: T);
```
Return `ba` to the free list and set `ba := NIL`.

`Free(ba)` is a checked runtime error if `ba` is `NIL`.

```
END Batch.
```

10.2 The BatchUtil interface

This interface provides operations to clip and translate a batch of painting commands. It is useful to those who are implementing window classes with customized painting behavior.

Don't apply these procedures to a batch whose contents are concurrently being read or written.

```
INTERFACE BatchUtil;

IMPORT Batch, Rect, Point, PaintPrivate;

PROCEDURE GetLength(ba: Batch.T): CARDINAL;
```
Return the number of `Word.Ts` in use in `ba`.

```
PROCEDURE Copy(ba: Batch.T): Batch.T;
```
Allocate and return a new batch initialized with a copy of `ba`.

Every entry in a batch has a clipping rectangle; there is also a clipping rectangle for the batch as a whole. The effective clipping rectangle for a painting operation is the intersection of its clipping rectangle with its batch's clipping rectangle.

```
PROCEDURE GetClip(ba: Batch.T): Rect.T;
```
Return ba's clipping rectangle.

```
TYPE ClipState = {Unclipped, Clipped, Tight};

PROCEDURE GetClipState(ba: Batch.T): ClipState;
```
Return ba's clipping state.

If `GetClipState(ba)` is `Clipped` then the clipping rectangle of every painting operation in `ba` is a subset of `GetClip(ba)`. If `GetClipState(ba)` is `Tight` then `GetClip(ba)` is equal to the join of the clipping rectangles of the painting operations in `ba`. If `GetClipState(ba)` is `Unclipped`, there is no particular relationship between `ba`'s clipping rectangle and the clipping rectangles of the entries in `ba`.

```
PROCEDURE Meet(ba: Batch.T; READONLY clip: Rect.T);
```
Set ba's clipping rectangle to `Rect.Meet(GetClip(ba), clip)`.

If the assignment is non-trivial, this will change the clip state of `ba` to be `Unclipped`.

```
PROCEDURE Clip(ba: Batch.T);
```
Apply ba's clipping rectangle to each operation.

That is, if `GetClipState(ba)` is `Unclipped`, then for each painting operation in `ba`, `Clip` replaces the clipping rectangle of the operation with the meet of the rectangle and `GetClip(ba)`, and sets the clipstate of `ba` to `Clipped`.

```
PROCEDURE Tighten(ba: Batch.T);
```
Achieve `ba.clipped = Tight` without changing the effect of ba.

That is, `Tighten(ba)` is equivalent to `Clip(ba)` followed by assigning to `ba`'s clipping rectangle the join of the resulting clipping rectangles of the entries in `ba`.

```
PROCEDURE Translate(ba: Batch.T;
  READONLY delta: Point.T);
```
Translate ba by `delta`.

That is, for each painting operation in `ba`, translate the target of the painting operation by `delta`. This always involves translating the clipping rectangle of the operation by `delta`. It also adds `delta` to the `delta` components of all textures and to the reference point of `TextComs`. It adjusts the `p1`, `p2`, `vlo`, and `vhi` fields of `TrapComs`. The relative displacement of a scrolling command is not affected; that is, both the source and target of the scroll are translated by `delta`. The clipping rectangle of the batch is also translated.

```
PROCEDURE ByteSwap(ba: Batch.T);
```
Convert all text painting operations in ba to have the same byteorder as `PaintPrivate.HostByteOrder`.

```
PROCEDURE Succ(ba: Batch.T;
  cptr: PaintPrivate.CommandPtr)
  : PaintPrivate.CommandPtr;
```

Return the pointer to the entry in `ba` *that follows the one pointed to by* `cptr`*.*

`Succ(ba, NIL)` returns the first entry in `ba`; `Succ(ba, cptr) = NIL` when `cptr` is the last entry in `ba`. To visit each entry in the batch `ba`, use a loop like this:

```
cptr := BatchUtil.Succ(ba, NIL);
WHILE cptr # NIL DO
  CASE cptr.command OF ... END;
  cptr := BatchUtil.Succ(ba, cptr)
END
```

The `PaintPrivate` interface explains the format of the entries.

```
END BatchUtil.
```

10.3 The PaintPrivate interface

This interface defines the layout of entries in paint batches.

```
INTERFACE PaintPrivate;

IMPORT Rect, Point, Trapezoid, Word;

TYPE
  PaintOp = INTEGER;
  Pixmap = INTEGER;
  Font = INTEGER;
```

In a paint batch, `PaintOps`, `Pixmaps`, and `Fonts` are represented by integers in a screentype-dependent way. During rescreening an old batch might find its way to a screen of the wrong type, causing garbage to be painted; but the garbage will be painted over with the correct pixels promptly.

```
TYPE
  PaintCommand = {RepeatCom, TintCom, TextureCom,
    PixmapCom, ScrollCom, TrapCom, TextCom,
    ExtensionCom};
  PackedCommand = BITS 32 FOR PaintCommand;
  FixedSzCommand =
    [PaintCommand.RepeatCom..PaintCommand.TrapCom];
  ByteOrder = {MSBFirst, LSBFirst};
  PackedByteOrder = BITS 32 FOR ByteOrder;

VAR (*CONST*)
  HostByteOrder: ByteOrder;
```

There are eight types of entries; each of which begins with a word containing a `PaintCommand` that indicates which type of entry it is.

Entries of type `TintCom`, `TextureCom`, `PixmapCom`, `ScrollCom`, `TrapCom`, and `TextCom` are used to implement the `VBT` operations `PaintTint`, `PaintTexture`, `PaintPixmap`, `Scroll`, `PaintTrapezoid`, and `PaintText/PaintSub`.

A `RepeatCom` entry in a batch indicates that the preceding entry is to be re-executed with its clipping rectangle changed to that of the `RepeatCom` entry. For example, these are used for implementing `PolyTint`, `PolyTexture`, and `PaintRegion`. There are some restrictions on where `RepeatCom` entries can occur.

`ExtensionCom` entries can be used to implement additional painting operations beyond those that are built into Trestle.

Some of the entries are fixed size; that is, the size of the entry is determined by their type. The following array gives the sizes of the fixed-size commands:

```
CONST
  WS = BYTESIZE(Word.T);
  ComSize =
    ARRAY FixedSzCommand OF INTEGER
    {(BYTESIZE(CommandRec) + WS-1) DIV WS,
     (BYTESIZE(TintRec) + WS-1) DIV WS,
     (BYTESIZE(PixmapRec) + WS-1) DIV WS,
     (BYTESIZE(PixmapRec) + WS-1) DIV WS,
     (BYTESIZE(ScrollRec) + WS-1) DIV WS,
     (BYTESIZE(TrapRec) + WS-1) DIV WS};
```

`ComSize[c]` equals the size in `Word.T`s of a paint batch entry for the command `c`.

```
TYPE
  CommandRec =
    RECORD command: PackedCommand; clip: Rect.T END;
  CommandPtr = UNTRACED REF CommandRec;
  RepeatPtr = CommandPtr;
```

We define a `Rec` and a `Ptr` type for each kind of batch entry.

Every batch entry is a "pseudo-subtype" of a `Command`, in the sense that its record type has `CommandRec` as a prefix.

A repeat command has no other fields besides the command identifier itself and the clipping rectangle. Hence a `RepeatPtr` is simply a pointer to a `CommandRec`.

All of the batch entries that are not repeat commands contain a `PaintOp`. They are all pseudo-subtypes of the following `Rec` and `Ptr` types:

```
  PaintRec = RECORD
    command: PackedCommand;
    clip: Rect.T;
    op: PaintOp
  END;
```

```
PaintPtr = UNTRACED REF PaintRec;
```

The following four entry types correspond to `PaintTint`, `PaintPixmap`, `Scroll`, and `PaintTrapezoid` operations.

```
TintRec = RECORD
  command: PackedCommand;
  clip: Rect.T;
  op: PaintOp
END;
TintPtr = UNTRACED REF TintRec;

PixmapRec = RECORD
  command: PackedCommand;
  clip: Rect.T;
  op: PaintOp;
  delta: Point.T;
  pm: Pixmap
END;
PixmapPtr = UNTRACED REF PixmapRec;
TexturePtr = PixmapPtr;

ScrollRec = RECORD
  command: PackedCommand;
  clip: Rect.T;
  op: PaintOp;
  delta: Point.T;
END;
ScrollPtr = UNTRACED REF ScrollRec;
```

It is illegal for a `ScrollRec` to be directly followed in a batch by a `Repeat` command.

```
TrapRec = RECORD
  command: PackedCommand;
  clip: Rect.T;
  op: PaintOp;
  delta: Point.T;
  pm: Pixmap;
  p1, p2: Point.T;
  m1, m2: Trapezoid.Rational;
END;
TrapPtr = UNTRACED REF TrapRec;
```

If `tr` is a `TrapRec`, then `tr.p1` and `tr.p2` are points that are on the extensions of the west and east edges of the trapezoid, and `tr.m1` and `tr.m2` are the slopes of the west and east edges. The slopes are given as `(delta v) / (delta h)`. A zero denominator represents an infinite slope; i.e., a vertical edge. A zero numerator is illegal.

The entries that are not fixed-size are pseudo-subtypes of `VarSzRec`, which contains a `size` field with the number of `Word.T`'s in the entire entry.

```
VarSzRec = RECORD
  command: PackedCommand;
  clip: Rect.T;
  op: PaintOp;
  szOfRec: INTEGER;
END;
VarSzPtr = UNTRACED REF VarSzRec;
```

`PaintText` and `PaintSub` operations result in the following entry type, in which `command` will equal `TextCom`:

```
TextRec = RECORD
  command: PackedCommand;
  clip: Rect.T;
  op: PaintOp;
  szOfRec: INTEGER;
  byteOrder: PackedByteOrder;
  clipped: BITS BITSIZE(Word.T) FOR BOOLEAN;
  refpt: Point.T;
  fnt: Font;
  txtsz, dlsz: INTEGER;

 (* dl: ARRAY [0..dlsz-1] OF VBT.Displacement *)
 (* chars: ARRAY [0..txtsz-1] OF CHAR *)

END;
TextPtr = UNTRACED REF TextRec;
```

In a `TextRec`, the boolean `clipped` must be set if `boundingbox(text)` is not a subset of the batch's `clip`. A `TextRec` can be directly followed in a batch by a `Repeat` only if `clipped` is `TRUE`. The `dl` and `chars` fields are declared in comments since Modula-3 does not allow a record to contain a variable-sized array; they must be accessed using address arithmetic. The `chars` field will be padded out so that the `TextRec` ends on a word boundary.

The `byteOrder` field defines the byteorder of the characters. (Since paint batches can be transported across address spaces and merged, the byte order could be different for different records in a paint batch.)

```
ExtensionRec = RECORD
  command: PackedCommand;
  clip: Rect.T;
  op: PaintOp;
  szOfRec: INTEGER;
```

```
    delta: Point.T;
    pm: Pixmap;
    fnt: Font;
    subCommand: INTEGER;

  (* extensionData: ARRAY OF CHAR *)

END;
ExtensionPtr = UNTRACED REF ExtensionRec;
```

An `ExtensionRec` can be used to implement painting operations that exploit rendering primitives that may be available on some particular implementation. Extension commands get a `PaintOp`, a `delta`, a `pm`, and a `fnt` "for free"; they can also put whatever data they need into the rest of the extension data part of the record. The field `szOfRec` is the number of `Word.Ts` in the extension record, including the extension data. When an `ExtensionRec` is translated, it's `clip` and `delta` fields are translated automatically; its extension data is unaffected.

```
PROCEDURE CommandLength(p: CommandPtr): INTEGER;
```

Return the length in words of the command entry `p`.

```
END PaintPrivate.
```

11 Miscellaneous interfaces

11.1 The VBTTuning interface

This interface defines values that can be changed to maximize Trestle's performance on particular systems.

```
INTERFACE VBTTuning;

IMPORT Word;

CONST
  BatchSize: CARDINAL = 325;
  BatchLatency: CARDINAL = 50000;
  HVParlim: CARDINAL = 100000;
  ZParlim: CARDINAL = 100000;
  ResumeLength: CARDINAL = 1;
  CombineLimit: CARDINAL
    = (BatchSize * ADRSIZE(Word.T)) DIV 2;
```

The value `BatchSize` is the number of `Word.T`'s in a standard painting batch.

The value `BatchLatency` is the number of microseconds before a paint batch is automatically forced.

The values `HVParlim` and `ZParlim` are the default minimum child areas (in pixels) for which `ZSplit` and `HVSplit` will fork separate repaint or reshaping threads.

`ResumeLength` is the size that a queue of paint batches must shrink to before a cross-address space filter will unblock a thread that painted into an overfull queue. It must be at least 1.

The value `CombineLimit` is the number of addressable units (e.g., bytes) in a batch beyond which Trestle will not consider combining another batch into it.

```
END VBTTuning.
```

11.2 The TrestleComm interface

```
INTERFACE TrestleComm;

EXCEPTION Failure;
```

Raised when communication to the window server fails.

```
END TrestleComm.
```

12 History and Acknowledgments

"There are lots of interesting problems in window systems", said Butler Lampson to Greg Nelson in April, 1984; and he was right. Nelson was enticed into the design meetings for the new window system for the Firefly multiprocessor at SRC. In 1984 most of the discussions were about what came to be called the event-time protocol, and besides Lampson and Nelson the main participants were Mark R. Brown, Jim Horning, and Lyle Ramshaw. Mark Brown and Greg Nelson wrote the first version of the VBT interface.

Mark Manasse joined SRC in 1985, and he and Nelson finished the design and implementation of the first version of Trestle (then called Trellis), which they shipped for use at SRC on December 31st, 1985.

Trestle evolved for five years, improving under feedback from the projects that built upon it, notably Luca Cardelli's Dialog Editor, Mark R. Brown's Ivy text editor, Marc H. Brown's FormsVBT system, Patrick Chan's session manager Rooms, and a number of applications built by Andrew Birrell. Bob Ayers's Facade system spurred the Trestle team into performance work that otherwise might never have been undertaken.

In 1990 and 1991, Steve Glassman, Mark Manasse, and Greg Nelson overhauled Trestle to make it into the portable Modula-3 X toolkit described in this reference manual. We are grateful to the Modula-3 export sites that used the alpha-test version of the system released in January 1991; special thanks for the helpful feedback from Dave Goldberg, Norman Ramsey, Jim Meehan, and Marc H. Brown. Finally, we thank Patrick Chan, James Mason, and Jim Horning, who carefully read the entire reference manual and made many helpful suggestions.

References

[1] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 Report (revised). Research Report 52, Digital Systems Research Center, November 1989.

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

[3] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison-Wesley, 1985.

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

[5] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System, 2nd ed.* Digital Press, 1990.

[6] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

Index

Acquire
    in VBTClass, 128
    in VBT, 32
Add
    in Point, 104
    in Rect, 108
    in Region, 111
AddChild
    in Split, 47
AddChildArray
    in Split, 46
AddHV
    in Region, 111
Adjust
    in HVSplit, 56
AllCeded
    in Trestle, 43
Altitude
    in ZSplit, 49
AnchorBtnVBT interface, 69–70
AnyMatch
    in ScrnFont, 97
AnyValue
    in ScrnFont, 97
Attach
    in Trestle, 41
AvailSize
    in HVSplit, 56
AwaitDelete
    in Trestle, 40
Axis interface, 104
AxisOf
    in HVSplit, 55
    in PackSplit, 58

bad region, 19, 21
batch (of painting commands), 29
Batch interface, 132
BatchUtil interface, 132–134
BeginGroup
    in VBT, 29
Bg
    in PaintOp, 75
BgBg
    in PaintOp, 76
BgFg
    in PaintOp, 76
BgSwap
    in PaintOp, 76
BgTransparent
    in PaintOp, 76
bitmaps, introduced, 4
BitOp
    in ScrnPaintOp, 87
BorderedVBT interface, 60–61
BoundingBox
    in Region, 111
    in ScrnFont, 98
BoundingBoxSub
    in ScrnFont, 98
BuiltIn
    in Font, 81
Button
    in VBT, 11
Buttons
    in VBT, 11
ButtonVBT interface, 65–66
BW
    in Cursor, 79
    in PaintOp, 77
ByteOrder
    in PaintPrivate, 134
    in ScrnPixmap, 92
ByteSwap
    in BatchUtil, 133

Cage
    in VBTClass, 125
    in VBT, 14
CageFromPosition
    in VBT, 15
CageFromRect

in VBT, 15
cages (for cursor tracking), 14
CageType
in VBTClass, 125
Capture
in Trestle, 43
in VBT, 30
CARDINAL
in VBTTuning, 139
Center
in Interval, 107
in Rect, 109
Ch
in ZSplit, 52
ChainedReshape
in ZSplit, 52
ChainReshapeControl
in ZSplit, 52
ChainSet
in ZSplit, 52
Change
in Interval, 106
CharMetric
in ScrnFont, 98
CharMetrics
in ScrnFont, 98
Child
in Filter, 60
in ProperSplit, 130
ClearNewShape
in VBTClass, 126
ClearShortCircuit
in VBTClass, 125
Clip
in BatchUtil, 133
ClipState
in BatchUtil, 133
Close
in Path, 114
ColorQuad
in PaintOp, 78
ColorScheme
in PaintOp, 78
CommandLength
in PaintPrivate, 138
CommandPtr
in PaintPrivate, 135
CommandRec
in PaintPrivate, 135
ComSize
in PaintPrivate, 135
Congruent
in Rect, 110
Connect
in Trestle, 44
Cons
in HVSplit, 55
in TSplit, 59
ConsArray
in HVSplit, 56
ConstructPlanewiseOp
in ScrnPaintOp, 87
coordinate system of a VBT, 2
coordinate system of screen, 1
coordinate translation from parent to child, 2
Copy
in BatchUtil, 132
in PaintOp, 75
in Path, 115
Cube
in ScrnColorMap, 102
CurrentPoint
in Path, 114
cursor, 1
Cursor interface, 79–80
cursor shape, how to change, 30
CursorClosure
in Palette, 82
CursorPosition
in VBT, 11
cursor tracking, 14
CurveTo
in Path, 114
cut buffer, 2

Decorate
in Trestle, 40

`Default`
in `BorderedVBT`, 61
`DefaultShape`
in `VBT`, 37
`DefaultSize`
in `HVBar`, 74
`Delete`
in `ProperSplit`, 131
in `Split`, 46
in `Trestle`, 40
`DeltaH`
in `VBT`, 27
`Detach`
in `VBTClass`, 128
`Difference`
in `Region`, 112
`Discard`
in `VBT`, 38
discard method, 38
`Displacement`
in `VBT`, 27
`DistSquare`
in `Point`, 105
`Div`
in `Point`, 105
`Domain`
in `VBT`, 8
`DontCare`
in `Cursor`, 79

`Empty`
in `Interval`, 106
in `Pixmap`, 80
in `Rect`, 107
in `Region`, 110
`EmptyCage`
in `VBT`, 14
`EndGroup`
in `VBT`, 29
`EndStyle`
in `VBT`, 24
`Entry`
in `ScrnColorMap`, 102
`Equal`
in `Region`, 112
`ErrorCode`
in `VBT`, 31
event time protocol, introduced, 4
`EverywhereCage`
in `VBT`, 14
exposed region, 19, 21
`ExtensionPtr`
in `PaintPrivate`, 138
`ExtensionRec`
in `PaintPrivate`, 137

`Factor`
in `Rect`, 109
`FeasibleRange`
in `HVSplit`, 56
`Fg`
in `PaintOp`, 75
`FgBg`
in `PaintOp`, 76
`FgFg`
in `PaintOp`, 76
`FgSwap`
in `PaintOp`, 76
`FgTransparent`
in `PaintOp`, 76
`Fill`
in `VBT`, 24
filter, 2
`Filter` interface, 60
`FilterClass` interface, 129
`Find`
in `HighlightVBT`, 63
`FixedSzCommand`
in `PaintPrivate`, 134
`Flatten`
in `Path`, 115
`Font`
in `PaintPrivate`, 134
`Font` interface, 81
`FontClosure`
in `Palette`, 82
fonts, introduced, 4
`ForceEscape`

in VBTClass, 127
ForceRepaint
in VBTClass, 127
in VBT, 20
Forge
in VBTClass, 129
in VBT, 34
Free
in Batch, 132
FromAbsBounds
in Interval, 106
FromAbsEdges
in Rect, 107
FromBitmap
in Pixmap, 81
FromBound
in Interval, 106
FromBounds
in Interval, 106
FromCorner
in Rect, 108
FromCorners
in Rect, 108
FromEdges
in Rect, 107
FromFontClosure
in Palette, 83
FromHV
in RigidVBT, 62
FromName
in Cursor, 80
in Font, 81
FromOpClosure
in Palette, 82
FromPoint
in Rect, 108
in Region, 111
FromRaw
in Cursor, 80
FromRect
in Region, 111
FromRects
in Region, 111
FromRef
in VBT, 35
FromRGB
in PaintOp, 77
FromSize
in Interval, 106
in Rect, 108
Full
in Rect, 107
in Region, 111

geometry interfaces, introduced, 4
Get
in AnchorBtnVBT, 70
in BorderedVBT, 61
in HighlightVBT, 64
in PackSplit, 58
in TextVBT, 72
in TextureVBT, 73
GetBadRegion
in VBTClass, 126
GetClip
in BatchUtil, 133
GetClipState
in BatchUtil, 133
GetCurrent
in TSplit, 59
GetCursor
in VBTClass, 125
GetDecoration
in Trestle, 41
GetDomain
in ZSplit, 50
GetFont
in TextVBT, 72
GetLength
in BatchUtil, 132
GetMiscCodeType
in VBT, 33
GetParent
in AnchorBtnVBT, 69
GetParentDomain
in ZSplit, 51
GetProp
in VBTClass, 125

in VBT, 38
GetQuad
in TextVBT, 72
GetScreens
in Trestle, 43
GetSelection
in VBT, 31
GetShape
in VBTClass, 128
GetShapes
in VBTClass, 128
GetTextRect
in TextVBT, 72
GoneCage
in VBT, 14
Gray
in Pixmap, 80

HasNewShape
in VBTClass, 126
HGap
in PackSplit, 58
HighlightVBT interface, 63–64
HorSize
in Rect, 109
HVBar interface, 74
HVSplit interface, 54–56

ICCCM, 1
Iconize
in Trestle, 41
Index
in Split, 46
Init
in Palette, 83
init method, rules for calling, 48
InOut
in VBT, 14
input or keyboard focus, 31
Insert
in ProperSplit, 130
in Split, 46
in ZSplit, 49
InsertAfter
in ZSplit, 48
InsertAt
in ZSplit, 49
Inset
in Interval, 106
in Rect, 108
in Region, 111
InsideCage
in VBT, 14
Install
in Trestle, 39
installing a top level window, 2
InstallOffscreen
in Trestle, 42
Interval interface, 105–107
Invert
in HighlightVBT, 64
IsActive
in AnchorBtnVBT, 70
IsClosed
in Path, 114
IsEmpty
in Interval, 107
in Path, 114
in Rect, 110
in Region, 112
IsMapped
in ZSplit, 50
IsMarked
in VBT, 18
IsRect
in Region, 113

Join
in Interval, 106
in Rect, 108
in Region, 112
JoinRect
in Region, 112
JoinRegions
in Region, 112
JoinStyle
in VBT, 24

Key
in VBTClass, 127
key method, 16
keyboard focus, 31
keyboard focus, introduced, 3
KeyRec
in VBT, 16
KeySym
in VBT, 17

Leaf
in VBT, 10
leaf VBT, introduced, 1
Lift
in ZSplit, 50
Line
in VBT, 25
LineTo
in Path, 114
LL (Locking Level), 8
Locate
in Split, 46
LocateChanged
in VBTClass, 123

MakeColorQuad
in PaintOp, 78
MakeColorScheme
in PaintOp, 78
Map
in Path, 115
in ZSplit, 50
MapObject
in Path, 115
Mark
in VBT, 18
marking for redisplay, 17
Max
in Point, 105
MaxSubset
in Region, 112
Meet
in BatchUtil, 133
in Interval, 106
in Rect, 109
in Region, 112
MeetRect
in Region, 112
Member
in Interval, 107
in Rect, 110
in Region, 113
MenuBar
in ButtonVBT, 66
MenuBtnVBT interface, 67–68
Metrics
in ScrnFont, 99
Middle
in Interval, 106
in Rect, 109
Min
in Point, 105
Misc
in VBTClass, 127
misc method, 32
MiscCodeDetail
in VBT, 33
MiscCodeType
in VBT, 33
MiscCodeTypeName
in VBT, 33
MiscRec
in VBT, 32
MMToPixels
in VBT, 10
Mod
in Interval, 106
in Point, 105
in Rect, 110
Mode
in PaintOp, 77
in ScrnColorMap, 102
Modifier
in VBT, 11
Modifiers
in VBT, 11
Mouse
in VBTClass, 127

mouse, 1
MouseRec
    in VBT, 12
mouse focus, 13
Move
    in Interval, 106
    in ProperSplit, 131
    in Split, 46
    in ZSplit, 49
MoveH
    in Point, 105
MoveHV
    in Point, 105
MoveNear
    in Trestle, 41
MoveTo
    in Path, 114
MoveV
    in Point, 105
Mul
    in Point, 104

New
    in AnchorBtnVBT, 69
    in Batch, 132
    in BorderedVBT, 61
    in ButtonVBT, 66
    in HVBar, 74
    in HVSplit, 55
    in HighlightVBT, 63
    in MenuBtnVBT, 67
    in PackSplit, 58
    in QuickBtnVBT, 67
    in RigidVBT, 62
    in TextVBT, 71
    in TextureVBT, 73
    in TranslateVBT, 65
    in ZSplit, 48
NewRaw
    in ScrnPixmap, 91
NewShape
    in VBT, 37
NorthEast
    in Rect, 109
NorthWest
    in Rect, 109
NotReady
    in Cursor, 79
Nth
    in Split, 45
NullDetail
    in VBT, 33
NumChildren
    in Split, 45

OpClosure
    in Palette, 82
Oracle
    in ScrnColorMap, 100
    in ScrnCursor, 88
    in ScrnFont, 94
    in ScrnPaintOp, 85
    in ScrnPixmap, 90
Origin
    in Point, 104
Other
    in Axis, 104
Outside
    in VBT, 15
Overlap
    in Interval, 107
    in Rect, 110
    in Region, 113
    in Trestle, 41
OverlapRect
    in Region, 113

PackedByteOrder
    in PaintPrivate, 134
PackedCommand
    in PaintPrivate, 134
PackSplit interface, 57–58
paint batch, 29
PaintBatch
    in VBTClass, 129
PaintCommand
    in PaintPrivate, 134
painting operation code, introduced, 4

PaintOp
in PaintPrivate, 134
PaintOp interface, 75–78
PaintPixmap
in VBT, 26
PaintPrivate interface, 134–138
PaintPtr
in PaintPrivate, 136
PaintRec
in PaintPrivate, 135
PaintRegion
in VBT, 24
PaintScrnPixmap
in VBT, 26
PaintSub
in VBT, 29
PaintText
in VBT, 27
PaintTexture
in VBT, 22
PaintTint
in VBT, 23
PaintTrapezoid
in VBT, 24
Pair
in PaintOp, 77
palette, 83
Palette interface, 82–83
Parent
in VBT, 10
parent VBT, introduced, 1
Partition
in Rect, 109
Path interface, 113–115
Pixel
in ScrnColorMap, 102
in ScrnPaintOp, 85
in ScrnPixmap, 92
Pixmap
in PaintPrivate, 134
Pixmap interface, 80–81
PixmapClosure
in Palette, 82
PixmapDomain
in VBT, 26
PixmapPtr
in PaintPrivate, 136
PixmapRec
in PaintPrivate, 136
pixmaps, introduced, 4
Place
in Region, 112
PlaceAxis
in Region, 111
PlaneWiseOracle
in ScrnPaintOp, 86
Point interface, 104–105
pointing device, 1
PolyTexture
in VBT, 23
PolyTint
in VBT, 23
Position
in VBTClass, 127
PositionRec
in VBT, 13
Pred
in Split, 45
Predefined
in Cursor, 79
in Font, 81
in PaintOp, 75
in Pixmap, 80
Prefix
in VBTClass, 120
in VBT, 7
PreInsert
in ProperSplit, 130
Primary
in ScrnColorMap, 102
Private
in HVSplit, 54
in PackSplit, 57
in ScrnColorMap, 101
in ScrnCursor, 88
in ScrnFont, 95
in ScrnPaintOp, 85
in ScrnPixmap, 90

in TSplit, 59
in ZSplit, 47
Proc
in ButtonVBT, 66
Project
in Interval, 106
in Rect, 109
proper split, 2
ProperSplit interface, 130–131
property set, of window, 37
Public
in AnchorBtnVBT, 69
in BorderedVBT, 61
in ButtonVBT, 65
in FilterClass, 129
in Filter, 60
in HVBar, 74
in HVSplit, 55
in HighlightVBT, 63
in PackSplit, 57
in ProperSplit, 130
in RigidVBT, 62
in ScreenType, 84
in ScrnColorMap, 101
in ScrnCursor, 89
in ScrnFont, 97
in ScrnPaintOp, 87
in ScrnPixmap, 91
in TSplit, 59
in TextVBT, 71
in TextureVBT, 72
in VBTClass, 121
in VBT, 7
in ZSplit, 47
Put
in TextVBT, 72
in VBTClass, 128
in VBT, 34
PutProp
in VBTClass, 125
in VBT, 38

QuickBtnVBT interface, 67

race conditions in the user interface, 3
Ramp
in ScrnColorMap, 102
Rational
in Trapezoid, 116
Raw
in Cursor, 79
in Pixmap, 81
in ScrnCursor, 88
in ScrnPixmap, 90
Read
in VBT, 35
read method, 36
reading a selection (introduction), 3
reading the screen, 30
Ready
in VBT, 35
Rect interface, 107–110
Redisplay
in VBTClass, 127
redisplay method, 17
Region interface, 110–113
Release
in VBTClass, 128
in VBT, 32
RemProp
in VBTClass, 125
in VBT, 38
Repaint
in VBTClass, 127
repaint method, 19
RepeatPtr
in PaintPrivate, 135
Replace
in Filter, 60
in Split, 46
Rescreen
in VBTClass, 126
rescreen method, 19
RescreenRec
in VBT, 19
Reset
in Path, 114
Reshape

in VBTClass, 126
reshape method, 18
ReshapeControl
in ZSplit, 51
ReshapeRec
in VBT, 18
ResolveCursor
in Palette, 83
ResolveFont
in Palette, 83
ResolveOp
in Palette, 83
ResolvePixmap
in Palette, 83
resources, introduced, 4
RGB
in Cursor, 79
in ScrnColorMap, 102
RigidVBT interface, 62

Scale
in Point, 105
ScaledReshape
in ZSplit, 53
Screen
in Trestle, 43
ScreenArray
in Trestle, 43
ScreenID
in Trestle, 42
in VBT, 11, 14
ScreenOf
in Trestle, 42
ScreenOfRec
in Trestle, 42
ScreenType
in VBT, 9
ScreenType interface, 84
ScreenTypeOf
in VBT, 10
ScreenTypePublic
in VBT, 9
ScreenTypeResolution
in ScrnFont, 97
screentypes, introduced, 5
ScrnColorMap interface, 100–103
ScrnCursor interface, 88–89
ScrnFont interface, 94–100
ScrnPaintOp interface, 85–88
ScrnPixmap interface, 90–94
Scroll
in VBT, 21
ScrollPtr
in PaintPrivate, 136
ScrollRec
in PaintPrivate, 136
Selection
in VBT, 31
SelectionName
in VBT, 31
selections, introduced, 2
Set
in AnchorBtnVBT, 69
in PackSplit, 58
in TextureVBT, 73
SetCage
in VBTClass, 128
in VBT, 15
SetColor
in BorderedVBT, 61
SetCurrent
in TSplit, 59
SetCursor
in VBTClass, 128
in VBT, 30
SetFont
in TextVBT, 72
SetParent
in AnchorBtnVBT, 69
SetRect
in HighlightVBT, 64
SetReshapeControl
in ZSplit, 51
SetShortCircuit
in VBTClass, 125
SetSize
in BorderedVBT, 61
SetTexture

in HighlightVBT, 64
Shape
in RigidVBT, 62
Size
in Interval, 106
SizeRange
in RigidVBT, 62
in VBT, 37
Slant
in ScrnFont, 96
Solid
in Pixmap, 80
source selection, 31
SouthEast
in Rect, 109
SouthWest
in Rect, 109
Spacing
in ScrnFont, 96
Split
in VBT, 10
Split interface, 45–47
split VBT, introduced, 1
Strike
in ScrnFont, 98
StrikeFont
in ScrnFont, 98
StrikeOracle
in ScrnFont, 97
Stroke
in VBT, 25
Sub
in Point, 104
in Rect, 108
in Region, 111
Subset
in Interval, 107
in Rect, 110
in Region, 113
SubsetRect
in Region, 113
Succ
in BatchUtil, 134
in Split, 45
Swap
in PaintOp, 75
SwapBg
in PaintOp, 76
SwapFg
in PaintOp, 76
SwapPair
in PaintOp, 78
SwapSwap
in PaintOp, 76
SwapTransparent
in PaintOp, 76
SymmetricDifference
in Region, 112
Sync
in VBT, 29

target selection, 31
TextItem
in MenuBtnVBT, 67
TextPointer
in Cursor, 79
TextPtr
in PaintPrivate, 137
TextRec
in PaintPrivate, 137
TexturePtr
in PaintPrivate, 136
textures, introduced, 4
TextureVBT interface, 72–73
TextVBT interface, 71–72
TextWidth
in ScrnFont, 98
TickTime
in Trestle, 44
Tighten
in BatchUtil, 133
time interval between events, 11
TimeStamp
in VBT, 11
TintPtr
in PaintPrivate, 136
TintRec
in PaintPrivate, 136

top level window, 2
`ToRects`
    in `Region`, 111
`Translate`
    in `BatchUtil`, 133
    in `Path`, 115
`TranslateVBT` interface, 65
`Transparent`
    in `PaintOp`, 75
`TransparentBg`
    in `PaintOp`, 76
`TransparentFg`
    in `PaintOp`, 76
`TransparentSwap`
    in `PaintOp`, 76
`TransparentTransparent`
    in `PaintOp`, 76
`Transpose`
    in `Point`, 105
    in `Rect`, 108
`Trapezoid` interface, 116
`TrapPtr`
    in `PaintPrivate`, 136
`TrapRec`
    in `PaintPrivate`, 136
Trestle abstraction, introduced, 1
`Trestle` interface, 39–44
`TrestleComm` interface, 139
`TSplit` interface, 59

`Unmap`
    in `ZSplit`, 50
`Unmark`
    in `VBT`, 18

`Value`
    in `VBT`, 35
`VarSzPtr`
    in `PaintPrivate`, 137
`VarSzRec`
    in `PaintPrivate`, 137
VBT abstraction, introduced, 1
`VBT` interface, 7–38
`VBTCageType`
    in `VBTClass`, 125
`VBTClass` interface, 120–129
`VBTTuning` interface, 139
`VerSize`
    in `Rect`, 109
`VGap`
    in `PackSplit`, 58

`WindingCondition`
    in `VBT`, 24
`Write`
    in `VBT`, 36
write method, 36
writing a selection (introduction), 3
`WS`
    in `PaintPrivate`, 135

Xlib, 1

`ZSplit` interface, 47–53