Functional Specification

Vista: SDD Display Window Package

September 24, 1980

Introduction

Vista, sometimes called "the New Window Package", replaces current SDD windows implementations and is a supported "development software" package. It is used by the Tools Environment & Debugger and will be the package of choice for private system developments, such as the Desktop Prototype.

Overview

Hardware and Operating System

Vista runs on Altos and Dolphins. The external interface presented to the client is the same on the two machines. Configurations supported:

Narrow Altos running Mesa

Wide Altos running Mesa (two banks only)

XMesa Altos running Mesa

Dolphins running Mesa (Alto display only)

Dolphins running Pilot

The Window Tree

Vista creates a tree of *windows*.

There is one *root window* (at level "zero") which is always equated to the visible bitmap and which supplies the background grey.

Any window may have child windows contained within it.

Child windows "obscure" their parent -- that is they are "above" their parent in the apparent "stack" of windows visible on the screen.

Sibling windows may overlap: the eldest sibling -- the one which appears first in the list -- is the sibling "on top" of the stack.

Vista contains routines for creating and destroying windows, for arranging them, and for displaying data within them.

A comparison with the current Tools Environment: The first level of the window tree contains what, are now called windows. The lower levels of the window tree contain what are now called subwindows. There is no longer any distinction between windows and subwindows. In the following discussion, all symbols come from the interface **Window** unless otherwise qualified.

Organization

Vista is organized as a collection of procedures. It is passive, responding only to calls from the client's program. It creates no processes. It allocates almost no storage. There is a discussion about starting the world, and Vista's interface to the bitmap in a section on UserTerminal at the end of this document.

Windows

Windows occupy [possibly overlapping] rectangular regions of the virtual bitmap. A window's location and size are defined in terms of its parent's location; the root window is always at bitmap location [0,0] even though its **box.place** may not be [0,0]. The **box.place** of **rootWindow** is the screen place of the bitmap origin.

Windows "overlap" other windows and may be manipulated even when they are "under" other windows. Windows are contained within their parents rectangular regions: if they are of a size and position that they would "stick out" of their parent, their display is "trimmed off" at their parent's edge.

Storage Allocation

Most storage allocation is provided by the client. On window creation, the client provides a handle to object storage space that he has obtained. This lets Vista be independent of and not fight with the client's allocation strategies.

All Vista object handles are of the form POINTER TO **Object**. Note: all Vista objects must be in the MDS.

Often a client wishes to associate some private data with each window instance. The client can easily do this by providing Vista with storage blocks larger than SIZE [**Object**].

Window Painting

Vista provides a variety of procedures which enable the client to display data in a window by whitening and blackening the bits in the window. These procedures include:

- Character and string paint procedures
- Blacken/whiten/grey box procedures
- Copy-this-array-of-bits procedures
- Brush-and-trajectory paint procedures which allow graphics curves to be easily drawn (not in initial release).

Every window object contains a client-supplied *repaint procedure* which, on demand, will repaint all or part of the window. This procedure is invoked, for example, when a window which was obscured by an overlapping window suddenly becomes more visible.

Scrolling

Recall that areas that "stick out" of a window's parent are trimmed for display purposes. Arbitrary scrolling can be implemented quite simply by imbedding a window (the one that paints the data to be scrolled) within another window (the "frame") and then just altering the position (y coordinate for vertical scrolling) of the former within the latter: routines are provided which will perform the appropriate **BITBLT**s to minimize the area to be painted.

Invalid Areas

Within a window as shown on the bitmap, sections of bits may become incorrect due to "external circumstances" -- e.g. because a window that was hiding them was just deleted. Vista accumulates these invalid areas and then calls the client's *repaint procedure* to fix them up. The client can indicate to Vista whether to keep extensive (and expensive) information about the exact areas that are invalid, or to keep briefer "bounding" information: the client will make this choice depending on how clever his repaint procedure is at repainting small sub-regions of the window.

Normally, when the client is called to paint the invalid area(s), there are no bits in the area that are black but should be white (the window package has possibly cleared the area to insure this) so the repaint procedure can use "or" functions when repainting. If the client knows that his repaint procedure always sets all of the bits in the area(s), he indicates this in the window object: this may save Vista from performing unnecessary clearings.

Features Not Present

Vista is exclusively a windows and bitmap package. It has nothing to do with the keyboard and the mouse.

Vista provides no scrollbars or menus. Scrollbars can be built by the client with windows and menus can be built by the client with "bitmap under" windows.

Accelerators

Since all window movement is done via explicit call [unlike the current Tools implementation, where the WindowObject and SubwindowObject can be directly manipulated by the client], Vista can maintain accelerator information within windows such as *totally visible* and *totally invisible* bits.

Definitions Files

Vista follows the Pilot naming conventions for definitions files. There are seven definitions files in the package:

Window contains all of the normal windows functions.

WindowFont contains the font object and handle and the font related routines, which are exactly **Initialize**, **CharWidth**, and **FontHeight**.

WindowOps contains private interfaces used by the implementation.

WindowCookie contains interfaces which will be used by Vista to support "cookie cutter" style iconic movement. This is not supported in the first release, and "cookies" wil not be extensively discussed in this specification.

UserTerminal and **UserTerminalOps** contain functions for manipulating and polling the hardware that Vista will run upon. See the last section of this document for a discussion of their use.

Event provides the capability of having procedures called upon certain events happening to the environment. See the Tajo Functional Specification for information regarding it. It is included in the Alto version of Vista since the implementation of **UserTerminal** depends upon it.

The Window Object

Record Definition

```
        Place: TYPE = UserTerminal.Coordinate;
        Dims: TYPE = RECORD [w, h: INTEGER];
        Box: TYPE = RECORD [place: Place, dims: Dims];
        LinkedBox: TYPE = RECORD [boxes: Boxes, box: Box];

        Boxes: TYPE = POINTER TO LinkedBox;
        BoxesCount: TYPE = {none, one, many};


        Handle: TYPE = POINTER TO Object;

Object: TYPE = RECORD [
-- "Minus Land" is the possibly-allocated space before the location
-- that the WindowHandle points to.  It is like a backwards variant space.
-- Booleans in the record body indicate whether the optional things are really
-- present.  If present, they are in the order shown, away from the window:
-- cookie: MinusLandCookieCutter,  "cookie cutter" location .. NIL for none
-- under: MinusLandBitmapUnder,  "bitmap under" location .. NIL for none
        parent: Handle _ NIL,                       -- its parent
        sibling: Handle _ NIL,                      -- the sibling chain
        child: Handle _ NIL,                        -- first child
        box: Box _ [[0,0],[0,0]],                   -- parent relative location and size
        display: PROCEDURE [Handle] _ NIL,
        invalidBoxes: Boxes _ NIL,
        cookieCutterVariant: BOOLEAN _ FALSE,       -- Is a "cookie cutter" in minus land?
        cookie: BOOLEAN _ FALSE,                    -- Is window a "cookie" now?
        underVariant: BOOLEAN _ FALSE,              -- Is a "bitmap under" location in minus land?
        underNow: BOOLEAN _ FALSE,                  -- Is there a current active "bitmap under"?
        underSomewhat: BOOLEAN _ FALSE,             -- Is bitmapunder incorrect for invisibles?
        boxesCount: BoxesCount _ one,               -- How many boxes to compute invalid for
        clearingNotRequired: BOOLEAN _ FALSE,       -- does display proc totally replace?
        invalidNow: PRIVATE BOOLEAN _ FALSE,        -- Is some part of window currently invalid?
        totallyInvisible: PRIVATE BOOLEAN _ FALSE,  -- Is window's area totally obscured?
        totallyVisible: PRIVATE BOOLEAN _ FALSE,    -- Is window's area totally visible?
        obscuredBySibling: PRIVATE BOOLEAN _ TRUE,  -- Does a sibling obscure me?
        obscuredByUncle: PRIVATE BOOLEAN _ TRUE,    -- Does an ancestor's sibling obscure me?
        sticksOut: PRIVATE BOOLEAN _ TRUE           -- Does it stick out past an ancestor? -- ];

-- records to define the MinusLand components so "SIZE" works --
MinusLandCookieCutter: TYPE = MACHINE DEPENDENT RECORD [LONG POINTER];
MinusLandBitmapUnder: TYPE = MACHINE DEPENDENT RECORD [
        pointer: LONG POINTER, -- to WordsForBitmapUnder [window] words
        underChanged: PROCEDURE [Handle, Box],
         -- ^ called when bitmap under changes if # NIL --
        mouseTransformer: PROCEDURE [Handle, Place] RETURNS [Handle, Place] ];
         -- ^ called to convert a mouse position if # NIL --
```

Client-Data Associated with a Window

Since the client handles all the storage allocation, it is simple for him to reserve private space associated with a window object for his private context. All he has to do is to allocate more than SIZE [**Object**] storage for the window.

In Mesa, the convenient thing for the client to do is to declare a record type which includes both the

Object and his own data:

```
MyWindowObject: TYPE = MACHINE DEPENDENT RECORD [
 window: Window.Object,
 context: MyContext ];
MyWindowHandle: TYPE = POINTER TO MyWindowObject;
```

Then, given a Vista **Handle**, he can simply reference his data via

```
mine: MyWindowHandle _ LOOPHOLE [windowHandle];
something: MyContext _ mine.context;
```

"Minus Land"

There are optional fields within the **Object**. This is to minimize storage requirements. If the client is utilizing the optional features associated with these fields, he needs to allocate an appropriate amount of storage and supply a **Handle** that points to the correct place within the allocated block.

Example: if the client wants a "bitmap under" to be maintained by Vista, he can set the bitmapUnder boolean in the **Object** and supply, as the object's storage pointer, something like

```
Storage.Node [ SIZE[Object] + SIZE[MinusLandBitmapUnder] ] +
 SIZE[MinusLandBitmapUnder]
```

The "Cookie" Window [not supported in initial release]

Vista allows a window to appear to be of irregular shape. This is implemented by defining a normal rectangular window and then arranging things so that part of the background "below" the window "shows through" it.

The implementation is arranged so that a user who does not utilize cookies is not burdened with the implementation cost.

To define a cookie, the user defines a window with a **MinusLandCookieCutter**. The **MinusLandCookieCutter** points to a block of memory which is a bitmap that defines where the cookie-window is to be transparent. This "cookie cutter" bitmap mask contains ones where the cookie is opaque and zeros where the cookie is transparent. The "words per line" of the cookie cutter bitmap is **(window.box.dims.w+15)/16**. The cookie cutter may be redefined with this procedure:

```
WindowCookie.SetCookieCutter: PROCEDURE [Handle, LONG POINTER]
```

Before actually inserting any cookie windows into the window tree, the client must give the window package some working space to manipulate cookies and cookie cutters:

```
WindowCookie.SetCookieWorkingSpaces: PROCEDURE [CARDINAL, LONG POINTER, LONG
POINTER]
```

This procedure supplies two blocks of memory and a word count of how big each block is. Cookies which contain more words [**((cookie.box.dims.w+15)/16)*cookie.box.dims.h**] than the supplied blocks don't act like cookies.

The "Bitmap Under"

The user may supply a block of memory that Vista will use to maintain the bits that would appear, in the bitmap, where this window is if this window, and those covering it up, did not exist.

This is useful primarily because Vista can then, when this window is removed from the tree, fix up the bitmap without calling the **display** procedure of all the windows that were (partially) hidden by this one. Thus menus can appear and disappear quickly.

Associated with the bitmapUnder is the **underChanged** procedure. This procedure, if supplied, is called whenever the bits in the bitmapUnder change due to another window's painting. This is useful in implementing a "magnifying glass" window or other window that does not really cover up the windows under it, but rather wishes to transform them in some way.

Also associated with the bitmapUnder is the **mouseTransformer** procedure. This procedure, if supplied, is called whenever a bitmap location is resolved (by the **PlaceToWindowAndPlace** procedure) to a position within the bitmapUnder window. The **mouseTransformer** gets a chance to pass the bitmap location into a window which it ostensibly covers up. This too is useful in implementing a "magnifying glass" window if you want the user to be able to point "through" the magnifier.

The client can associate a bitmap under with a window with this call:

```
SetBitmapUnder: PROCEDURE [
 window: Handle,
 pointer: LONG POINTER _ NIL,
 underChanged: PROCEDURE [Handle, Box] _ NIL,
 mouseTransformer: PROCEDURE [Handle, Place] RETURNS [Handle, Place] _ NIL ]
```

The user-supplied space for the bitmap under (supplied by the long pointer) must contain at least **WordsForBitmapUnder**[window] words. [For AltoMesa programs the formula is: **((window.box.dims.w+30)/16)*window.box.dims.h**. The reason for the strange formula: the left edge of the data stored in the bitmap under is kept in the same bit-within-word position as the window's position within the real bitmap so that grey patterns stay aligned.]

Window Painting and the Display Procedure

Invalid Areas

Vista attempts to keep track of what parts of the window need to be repainted. A window can be in need of a partial repaint because a window that formerly obscured it has been moved; several disjoint pieces of a window can need repainting if the window is scrolled when it is partially obscured.

Within the window object supplied by the client, the boxesCount field indicates how the system should keep track of the "invalid areas" of this window. An invalid area is a rectangular sub-region of the

window that currently has its bits in the bitmap incorrect. The client, within the field, indicates how clever Vista should be at keeping track of the invalid area:

```
BoxesCount: TYPE = {none, one, many};
```

window.boxesCount = none =>

> No information about the invalid region is maintained. The client's **display** procedure, when it is called, needs to repaint the entire window (since it has no information).

window.boxesCount = one =>

> One "invalid box" is maintained. As areas become invalid, the box's edges are expanded, as necessary, to encompass the newly invalid regions. The client's **display** procedure, when it is called, only needs to paint the area defined by **window.invalidBoxes.box**.

window.boxesCount = many =>

> Multiple invalid regions are maintained. As areas become invalid, they are added to the chain beginning at **window.invalidBoxes**. The boxes are maintained sorted by **place.y** value. Consolidation of areas is done as appropriate. The client's display procedure, when it is called, needs to paint the boxes in the list headed by **window.invalidBoxes**.

Painting on Demand

The client must supply a *display procedure* which can repaint the window whenever Vista thinks it is necessary. Vista will call this procedure:

```
display: PROCEDURE [Handle];
```

The area(s) that need repainting are supplied by the **invalidBoxes** field in the window record:

```
invalidBoxes = NIL =>
    the whole window needs to be repainted
ENDCASE =>
```

> the chain of boxes headed by **invalidBoxes** needs to be repainted -- if the client indicated **boxesCount=one** then this chain will contain exactly one element.

As an aid to the client, the procedure

```
EnumerateInvalidBoxes [
 window: Handle,
 proc: PROCEDURE [Handle, Box] RETURNS [Box] ]
```

will do the following: it will call the client's procedure with a box from the list of invalid boxes; when that procedure returns a box (the area it painted -- typically the box it was passed) it will chop the returned box out of the invalid list, consolidating as appropriate. The chopping will be done on each box in the list for which the returned box covers at least all of one edge; therefore, the length of the invalid box list will not grow during this process. Iterating on the first box remaining in the list will be kept up until the list is empty.

The painting is assumed to be deterministic: if Vista calls the **display** procedure to paint areas that are already valid, it will repaint what is currently on the screen.

The area to be repainted will be clean -- that is there are no bitmap bits in it that are currently black but should be white -- so that the client can "or" data into the window. [Unless the client has set the **clearingNotRequired** BOOLEAN in the window.]

Painting Not-On-Demand

Often the client will wish to repaint his window with new bits -- e.g. when the user has issued a command. *The client should not call his **display** procedure to effect this.* Rather he should update his data to reflect the newly-desired content and then call

```
InvalidateBox [window: Handle, box: Box, clarity: Clarity _ isDirty];
  -- If clarity=isClear, Vista believes that the box is all white
  -- and performs no clearing.

Clarity: TYPE = {isClear, isDirty};
```

to mark the window, or portion thereof, invalid, and then call

```
Validate [window]
```

to indicate to Vista that any invalid areas should be "validated" by calling the window's display procedure. A call on InvalidateBox followed by a call on Validate may result in no call to the **display** procedure if, for example, the invalidated areas stick out of the parent.

If the client has potentially invalidated areas in a subtree, the procedure

```
ValidateTree [window _ rootWindow]
```

is called indicate to Vista that any invalid areas should be "validated". This is necessary if, for instance, the client has removed some window from the tree.

Painting Calls

The client's display procedure calls the window package to perform its window painting. The calls typically instruct the window package to do something to a rectangular area within the window. The window package performs all the required clipping of the box at winow edges, doing partial paints to allow for overlapping windows, maintaining "bitmap unders", etc. Some calls take **BitBlt** operations or sources and perform the appropriate **BitBlt** function. A good understanding of how **BitBlt** works is indicated before using any of those procedures.

A set of procedures allows a box to be set to a particular shade:

```
DisplayWhite: PROCEDURE [window: Handle, box: Box]
DisplayBlack: PROCEDURE [window: Handle, box: Box]
DisplayInvert: PROCEDURE [window: Handle, box: Box]
DisplayGrey: PROCEDURE [window: Handle, box: Box]
DisplayShade: PROCEDURE [window: Handle, box: Box,  grey: GreyArray]
DisplayBBopShade: PROCEDURE [
  source:  BBsourcetype, op: BBoperation,
```

```
  window: Handle, box: Box,  grey: GreyArray]
GreyArray: TYPE = ARRAY [0..3] OF WORD
BBsourcetype: TYPE = BitBlt.BBsourcetype;
BBoperation: TYPE = BitBlt.BBoperation;
```

A procedure displays a character into a particular place in the window. See the section discussing **WindowFont** for an explanation of the **font** argument.

```
DisplayCharacter: PROCEDURE [
 window: Handle,
 char: CHARACTER,
 place: Place, -- the upper left corner
 font: WindowFont.Handle _ NIL,
 bbop: BBoperation _ paint -- or replace or invert or erase
 source: BBsourcetype _ block -- or compliment -- ]
 RETURNS [ Place -- the upper left corner of the next guy -- ]
```

A procedure displays a substring in the window. It is designed to be considerably faster than a **DisplayCharacter** call within a loop. It takes characters from the substring argument, **bumping its pointer** and returns when the substring is exhausted or when it fills the supplied line length or when it finds a return character:

```
DisplaySubstring: PROCEDURE [
  window: Handle,
  ss: String.SubString,
  place: Place,
  lineLength: INTEGER _ 0, -- wh.box.dims.w - place.x if zero
  font: WindowFont.Handle _ NIL,
  bbop: BBoperation _ paint
  source: BBsourcetype _ block ]
  RETURNS [Place]
```

So that the client need not always manufacture a **Substring**, the following procedure will display a string. No feedback is given to the client to indicate how much of the string got painted:

```
DisplayString: PROCEDURE [
  window: Handle,
  s: STRING,
  place: Place,
  lineLength: INTEGER _ 0, -- wh.box.dims.w - place.x if zero
  font: WindowFont.Handle _ NIL,
  bbop: BBoperation _ paint
  source: BBsourcetype _ block ]
  RETURNS [Place]
```

Three procedures copy a bitmap of the client's into the window. The second allows the user to specify a x-offset into his data (a y-offset can be simulated by additions to the pointer) The third allows the client to specify the exact **BitBlt** operation to be performed:

```
DisplayData: PROCEDURE [
 window: Handle,
 box: Box, -- window area ... supplies data's dimensions
 data: LONG POINTER TO UNSPECIFIED,
 wpl: CARDINAL, -- "words per line" in client's bitmap
 bbop: BBoperation _ paint ];

DisplayOffsetData: PROCEDURE [
 window: Handle,
 box: Box, -- window area ... supplies data's dimensions
 data: LONG POINTER TO UNSPECIFIED,
 wpl: CARDINAL, -- "words per line" in client's bitmap
```

```
    offset: INTEGER, -- the x offset ... typically in [0..15]
    bbop: BBoperation _ paint ];

  DisplayOffsetBBopData: PROCEDURE [
    source: BBsourcetype,
    op: BBoperation,
    window: Handle,
    box: Box, -- window area ... supplies data's dimensions
    data: LONG POINTER TO UNSPECIFIED,
    wpl: CARDINAL, -- "words per line" in client's bitmap
    offset: INTEGER, -- the x offset ... typically in [0..15] -- ];
```

Series of paints in a predefined area

This following procedure is designed to avoid much of the overhead of successive calls to one of the normal window display routines. It is not available in Tajo.

```
  Trajectory: PROCEDURE [
    window: Window.Handle,
    box: Window.Box _ Window.NullBox, -- area where painting might occur .. default means anywhere --
    proc: PROCEDURE [Window.Handle] RETURNS [Window.Box, INTEGER],
    --  ^^ returns box to paint and xoffset into source --
    source: LONG POINTER _ NIL,
    wpl: CARDINAL _ 1,
    bbop: BBoperation _ paint,
    bbsource: BBsourcetype _ block,
    missesChildren: BOOLEAN _ FALSE, -- currently unused --
    grey: Window.GreyArray _ [177777B,177777B,177777B,177777B] ]
```

The client calls the trajectory procedure and passes to it:

The window of interest.

The box where painting might occur. If the client lies about this, bad things will happen.

A procedure of his which will, when called, repeatedly return small areas within the window where painting should occur. Think of them as the "brush strokes"

A set of arguments which define the type of bitblt operation that should be performed on each small area.

As an example of usage, consider the graphics package drawing a curve. It will call Trajectory with the window, the box that circumscribes the curve (allowing for the finite brush width), and a long pointer (and wpl) to the brush. Then the procedure, when repeatedly called from within the Trajectory routine, will return the boxes that define successive brush positions -- that is the curve.

To end the trajectory, the client's proc should return Window.NullBox.

The client may wish to alter the brush shape along the trajectory. It can do this by defining the "source" bitmap as a wide one, with several different brush shapes in it, and then returning, together with the brush-box, the xoffset into the source bitmap.

The Window Tree

The following procedures do not force repainting, except as noted below. Instead, they merely mark window areas as invalid and return. This is so the client can perform several functions and then trigger a single repaint. To force the repaint, the client calls ValidateTree[].

Structure

The windows that are currently known to Vista form a tree. The **parent**, **child**, and **sibling** fields within each window object serve to link this tree together.

If a window has descendants, its **child** field points to its eldest [topmost] **child**, and successive children are linked via their **sibling** fields. The *sibling* **field** of the youngest child is NIL. If a window has no children, its **child** field is NIL. The **parent** field points to the window which has this window as a child. The **parent** field of the **rootWindow** is NIL.

Root Window Definition

The first window to be defined must be the root window of the window tree. It is defined by the client:

```
DefineRoot: PROCEDURE [window: Handle, grey: GreyArray]
```

Where the client is passing the storage for the **rootWindow**; the storage for the hardware bitmap should already have been allocated and Vista will locate it by calling **UserTerminal.GetBitBltTable**. Repeated calls are OK. It is the responsibility of the client to get the bitmap turned off before calling **DefineRoot** when an outward call to **GetBitBltTable** will raise the signal **BitmapIsDisconnected**. The hardware bitmap's height and width are taken from the **Handle**, and must satisfy the hardware's constraints.

The grey array is the "background" that Vista will use to paint the root window.

The handle of the root window is accessable via the procedure

```
Root: PROCEDURE RETURNS [Handle]
```

or the variable **rootWindow**.

Window Definition

All window creation is done by the client. The client allocates storage for window objects and initializes

them. The client builds subtrees of windows with the appropriate initialization of their **child**, **parent**, and **sibling** fields.

The client inserts a window or subtree of windows into Vista's window tree via this call:

InsertIntoTree: PROCEDURE [window: Handle]

Before making the call, the client has set many fields in the window(s) appropriately: Other values are defaulted, so it is recommended that a constructor be used to initialize the window to allow Mesa to default them to the correct values.

box: Defines the window's location (relative to its parent) and size

parent: Indicates where in the tree structure this subtree belongs. This subtree will be inserted as a child of window.parent.

sibling: Indicates where in the tree structure this subtree belongs. This subtree will be inserted as the immediately-older sibling of window.sibling. If window.sibling=NIL, this window will be the youngest child of window.parent. If window.sibling is non-NIL and not a child of window.parent, a client error has occurred.

child: Supplies a window subtree, built by the client. NIL if the client is inserting only a single window.

display: The client's repaint procedure.

boxesCount: As described above.

underVariant: Indicates "Minus Land" bitmapUnder info.

cookieCutterVariant: Indicates "Minus Land" cookie info.

The client forces all the windows just inserted to be painted by calling **ValidateTree** passing a window that contains all of the inserted windows. If an inserted window has a bitmap under and the new window is partially obscured (meaning that all the bits needed for the bitmap under are not available) then **ValidateTree** will be called on the parent of the inserted window to obtain those bits.

Window De-Definition

The client can remove a window, together with any window subtree "below" it, from Vista's window tree via

RemoveFromTree: PROCEDURE [Handle]

This procedure removes the window and all of its descendants from Vista's window tree without altering their **parent**, **child**, and **sibling** fields. Thus they remain a legal subtree and can be re-inserted with **InsertIntoTree**.

The client forces all the repainting of the windows exposed by calling **ValidateTree** passing a window that contained all of the removed windows.

Window Movement

Removing and Inserting Windows

The default or normal way for a client to alter the window tree, either to restructure it or to rearrange it on the screen, is to remove a window or subtree from the tree, alter it, and then insert it. Such a procedure is universally applicable; however there are a couple of special cases that are optomized by the windows package to minimize the window repainting that would occur if the remove followed by insert procedure was followed.

Stacking a Window

A common operation is to alter the ontop-ness of a set of windows. The windows package has a procedure which effects this operation and minimizes repaints:

```
Stack: PROCEDURE [window: Handle, newSibling: Handle, newParent: Handle _ NIL]
```

If **newParent** is not NIL, then **window** is moved to be a child of **newParent**. The sibling-list containing the window **window** so that **window** is now immediately above **newSibling** in the stack. Supplying **newSibling**=NIL puts **window** on the bottom of the sibling stack. Unless **window** is already on top, supplying **newSibling=window.parent.child** puts **window** on the top of the stack. If **window** is on top, the previous expression is a client error which is not guarded against.

Moving a Window: Scrolling

As discussed earlier, scrolling is performed by moving a window (the scrollee) around within its parent (the frame). Typically the scrollee with be very tall and the same width as the frame: thus the scrollee will completely obscure the frame's bits and the only function of the frame is to clip the scrollee. A procedure which effects moving a child and minimizes repaints:

```
Slide: PROCEDURE [window: Handle, newPlace: Place]
```

The function is called "Slide" to distinguish it from tree rearrangement. It is given a new [x, y] where the window should go. The procedure can be used for any child movement; typically it is used for scrolling, and then the new place is vertically aligned with the old. For example, given a *scrollee* window within a frame, one can scroll the data upward by:

```
Slide [ scrollee, Place [scrollee.box.place.x, scrollee.box.place.y-100] ]
```

Moving a Window Iconically: I & II

The user's moving a (typically small) window around on the screen is also a common tools and desktop function. This case is different that the one above because the "icon" being moved is typically small and

quick-to-paint. This makes the trade-offs in minimizing repaints different and results in this procedure call:

```
SlideIconically: PROCEDURE [window: Handle, newPlace: Place]
```

The above procedure moves the window "correctly" in that the window will, for example, slide "under" siblings as appropriate. This logic makes it relatively slow and it doesn't keep up with user mouse-movement very well. SlideIconically is not available in Tajo.

An alternative procedure is much faster in a restricted set of circumstances:

```
Float: PROCEDURE [
 window: Handle,
 temp: Handle,
 proc: PROCEDURE [window: Handle] RETURNS [place: Place done: BOOLEAN] ]
```

This procedure requires that the **window** be a bitmapUnder window. It also requires that the user supply a **temp** window, exactly the same size as **window**, not in the window tree, also with a bitmapUnder, for scratch storage. The procedure repeatedly calls the client's procedure and does a continuous move to the new place as long as the **done** boolean is FALSE. The window is forced to the top of the sibling stack before the move begins. A new place which would entail moving the window so it is not completely visible is a client error. **ValidateTree** is called to pick up the bits that need to be on the bitmap when the **window** is moved away.

"Growing" a Window

Altering the size of a window is also a common operation. Frequently, the old and new window bitmap areas are not disjoint. This requires a single call (instead of a remove and insert) to save painting:

```
SlideAndSize: PROCEDURE [ window: Handle, newBox: Box, gravity: Gravity _ nw ]

SlideAndSizeAndStack: PROCEDURE [
 window: Handle, newBox: Box, newSibling: Handle,
 newParent: Handle _ NIL, gravity: Gravity _ nw ]
```

These procedures minimize the amount of invalidation they must do -- that is they try and "save" as much of the current window content as they can. The gravity argument indicates what should happen to the window's content when the size changes (this applies both to the bits within the window and to the window's children:

```
Gravity: TYPE = {nil nw, n, ne, e, se, s, sw, w, c, xxx}
```

gravity=nw => contents stay in the upper left corner: this is the normal case.

gravity=center => contents stays in the middle (e.g. trimming occurs equally at all edges).

gravity=nil => the contents stay the same place **on the bitmap**.

gravity=xxx => no attempt is made to save the contents: it is all repainted.

ENDCASE => the contents stay attached to the indicated compass point, which is either a corner or the middle of a side: like nw but flushed differently.

Moving an area within a window

The client may need to move a piece of the window without repainting the whole thing. This occurs, for example, as editors handle inserts and deletes. Such an operation is tricky, since during the area move, the window is difficult to repaint since its "current" state is ill-defined. This is discussed below. The procedure which takes a box within a window and copies it to somewhere else in the window is:

```
DisplayShift: PROCEDURE [window: Handle, box: Box, place: Place]
```

To avoid difficulties with the client's **display** procedure, this call, although it may produce invalid areas within the window (bits that should be moved into visible areas of the window but are not available either due to being clipped or obscured), does not call the **display** procedure, but simply leaves the **window** marked invalid. It is the client's responsibility to call **ValidateTree[]** as soon as he has corrected his data structures to reflect the call.

Also, **DisplayShift** does not invalidate the areas where the box has been moved "from". If they should be repainted, invalidating them is the client's responsibility.

Fonts

The following section describes the **WindowFont** interface. The text painting procedures of the **Window** interface take as an argument a **Handle** on an object from **WindowFont**. These fonts are all in strike-format.

Object and Handle

Vista defines a **Object** and **Handle** which are mostly private to the implementation.

```
Handle: TYPE = POINTER TO Object;
Object: TYPE = RECORD [
  height: [0..7777B] _ NULL,
  width: PACKED ARRAY CHARACTER [0C..177C] OF [0..255] _ NULL,
    ... ,
  swapper: PROCEDURE [Handle] _ NIL,
  address: LONG POINTER,
     ...  ];
```

Initialization

Vista font routines deal only in .strike fonts. The client executes the following procedure to create an internal font of his choice:

```
Initialize: PROCEDURE [font: Handle]
```

Where the supplied **Handle** points to a font record, allocated by the client, which is at least SIZE[**Object**] words long. The fields the client is responsible for filling in *before* calling **Initialize** are **font.swapper** and **font.address**. The rest of the fields are READONLY, and are filled in by calling the **Initialize** procedure. The font must be swapped in at Initialize time.

Usage

The bits within the font object that define the character pictures are private to the implementation. The only public interfaces allow the client to determine the sizes of the characters in screen dots:

```
CharWidth: PROCEDURE [char: CHARACTER, font: Handle _ NIL] RETURNS [[0..LAST[INTEGER]]]

FontHeight: PROCEDURE [font: Handle _ NIL] RETURNS [[0..LAST[INTEGER]]]
```

A **font** argument of NIL for these routines, as well as for the text painting routines of the **Window** interface, means to use the **defaultFont**. The **defaultFont** is set by calling

```
SetDefault: PROCEDURE [font: Handle]
```

Using this defaulting mechanism before the **defaultFont** is set is a client error.

Swapping

Vista allows the data which defines the bitmap representation of the characters in the font to be swapped by the client.

Whenever Vista is about to utilize the bitmap data in a font and the **font.address** is NIL, it calls the clients swap procedure to fill in the **font.address** with a pointer to the font. Whenever Vista is done utilizing the bitmap data and the clients swap procedure is not NIL, it calls the clients swap procedure to (possibly) swap out the data.

Thus a client that wishes to swap should supply a swap procedure and should set the bitmap pointer to NIL on swap out. And a client that wants fonts to be permanently in memory should set the swap procedure to NIL.

For AltoMesa programs, if the font character pictures and the bitmap are both not in the MDS, they must be in the same bank. This is due to a restriction on BitBlt.

The Other Parts of Vista

The current implementation of Vista has dependencies on outside interfaces. This is the result of its evolution and has resulted in a need for clumsy mechanisms for starting and changing the environment that Vista executes in. At some later date, these dependencies on outside interfaces will be removed. The Mesa/Tools group will supply modules that may be used to support Vista through

Vista's interfaces, but the client will be able to provide his own implementation of dealing with bitmaps, etc.

The following steps are necessary to start up Vista as it is currently configured:

Initialize Vista with the root window object that will be used.

In the Alto world, the implementation modules for UserTerminal must be started.

Get the bitmap allocated.

Let Vista know the bitmap parameters and get the root window inserted into the tree.

This can be accomplished by

```
Window.DefineRoot[window: @<your root object>, grey:, bitmapExists: FALSE];
UserTerminalOps.StartUserTerminal[wantWaitScanLine: TRUE];
[] _ UserTerminalOps.SetBitmapBox[[0, 0], [608, 808]];
[] _ UserTerminal.SetState[off];
Window.DefineRoot[
    window: @<your root object>, grey: <your background grey>, bitmapExists: TRUE];
[] _ UserTerminal.SetState[on];
```

The call on **StartUserTerminal** need only be made if running under Alto/Mesa. The argument should be TRUE if calls on **UserTerminal.WaitForScanLine** are going to be utilized. The reason for inserting the root window into the the tree between allocating the bitmap (**SetState[off]**) and having the bitmap displayed (**SetState[on]**) is to prevent the unitialized bitmap from being flashed onto the screen before the background grey has been painted into it.

UserTerminalOps.SetBitmapBox is also only useful if running under Alto/Mesa and it changes Window.rootWindow.box to be a copy of the box (trimmed to fit the hardware limitations if necessary) passed in. There are some instances in the Alto/Mesa system, that it is necessary for the client to do the allocation of the memory for the bitmap. A call on **UserTerminalOps.SetBitmapSpace** should be inserted immediately after the call on **UserTerminalOps.SetBitmapBox**.

Before the bitmap can be deallocated a call on **DefineRoot** with **bitmapExists:** FALSE must be made to suppress any paints into any window while the bitmap is not there. After the bitmap has been reallocated, another call on **DefineRoot** must be made to notifiy Vista that it can once again paint into the windows.

UserTerminal

The interface **UserTerminal** describes the state of the user input/output devices (i.e. display bitmap, display cursor, keyboard, mouse, and keyset), and allows the client to manipulate them. This interface takes as fixed many of the characteristics of these devices and only allows variations such as the number of keys or the size and resolution of the display. This interface deals with many of the lowest level attributes of the terminal and, with a few exceptions, should not be of interest to Tajo clients. This section presents definitions and functions of general interest first and then more

primitive ones that should only be used with proper knowledge, if at all.

Clients can determine the physical attributes of the display via the following exported variables.

```
screenWidth: READONLY CARDINAL[0..32767];
screenHeight: READONLY CARDINAL[0..32767];
pixelsPerInch: READONLY CARDINAL;
```

The bitmap display is addressed by xy coordinates defined as follows.

```
Coordinate: TYPE = MACHINE DEPENDENT RECORD [x, y: INTEGER];
```

The state of the display is defined as:

```
State: TYPE = {on, off, disconnected};
```

on - The display is physically on and visible to the user (bitmap allocated).

off - The display is physically off and not visible to the user (bitmap allocated).

disconnected - The same as **off** with no allocated bitmap.

Clients may alter the state of the bitmap display by calling

```
SetState: PROCEDURE [new: State] RETURNS [old: State];
```

Tajo clients may not call UserTerminal.SetState directly. They should use TajoMisc.SetState. Clients may determine the current state of the bitmap display by calling

```
GetState: PROCEDURE RETURNS [state: State];
```

The bitmap display is capable of displaying black-on-white or white-on-black. Clients may determine or alter the current state of the background by using the following procedures.

```
GetBackground: PROCEDURE RETURNS [background: Background];
SetBackground: PROCEDURE [new: Background] RETURNS [old: Background];

Background: TYPE = {white, black};
```

Clients may momentarily *blink* (video reverse) the display by calling

```
BlinkDisplay: PROCEDURE;
```

Some displays have the capability to display a border around the outside of the active display region. Clients can determine if the display has this capability by interrogating the following exported variable.

```
hasBorder: READONLY BOOLEAN;
```

If the display has a border, then clients may set the pattern to be displayed in the border by calling

```
SetBorder: PROCEDURE [oddPairs, evenPairs: [0..377B]];
```

The following function is provided for clients who need to synchronize bitmap alteration with display refresh.

```
WaitForScanLine: PROCEDURE [scanLine: INTEGER];
```

> [Note: Some implementations of this interface will only implement waiting for scan line zero. In Alto/Mesa, this procedure is only available if the argument to StartUserTerminal is TRUE. The module UserTerminalsA must be made resident. Tajo clients need not worry about these details.].

The following procedure will return a bitblt table with the bitmap fields filled in for the current bitmap.

```
GetBitBltTable: PROCEDURE RETURNS [bbt: BitBlt.BBTable];
```

> It is worth noting that clients cannot directly get at the bitmap. The bitmap parameters are contained in the **bbt** returned above. For a complete description of a **BBTable** see the interface **BitBlt.**
>
> In the event that the bitmap is currently disconnected (deallocated), the following error is raised.
>
> ```
> BitmapIsDisconnected: ERROR;
> ```

The cursor manipulation routines supplied here should not be used by general Tajo clients (see **Cursor** for more comprehensive cursor functions).

The display cursor is defined by a 16x16 bit array as follows.

```
CursorArray: TYPE = ARRAY [0..16) OF WORD;
```

Clients can determine the current bit pattern for the cursor by calling

```
GetCursorPattern: PROCEDURE RETURNS [cursorPattern: CursorArray];
```

The cursor pattern is set by calling

```
SetCursorPattern: PROCEDURE [cursorPattern: CursorArray];
```

The keyboard and function keyset defined in this interface is *uninterpreted*. This means that up/down key transitions are noted by the state of the bits in the following unencoded array (see interfaces **KeyStations** and **Keys** for uninterpreted and interpreted key/bit assignments).

```
keyboard: READONLY LONG POINTER TO READONLY ARRAY OF WORD;
```

The coordinates of the mouse and cursor can be found by the following exported variables.

```
mouse: READONLY LONG POINTER TO READONLY Coordinate;
cursor: READONLY LONG POINTER TO Coordinate;
```

Clients can alter the coordinates of the current mouse position by calling

```
SetMousePosition: PROCEDURE [newMousePosition: Coordinate];
```

UserTerminalOps

Much of **UserTerminalOps** is private to the implementation of **UserTerminal** for the Alto/Mesa world. However, there are some parts that the client needs to initialize Vista. As discussed earlier, one procedure must be called before using any of the parts of UserTerminal:

```
StartUserTerminal: PROCEDURE [needWaitScanLine: BOOLEAN _ TRUE];
```

Vista uses **machineFlavor**, which is an exported variable from **UserTerminalOps**, to know what kind of machine it is running on.

```
machineFlavor: MachineFlavor;
MachineFlavor: TYPE = {standard, widebody, xmesa39, xmesa, dolphin, dorado, spare};
```

At any point that the machine type might change out from under Vista, for example upon restarting an image, the client must call **Window.NoticeMachineFlavor**.

To set how much of the screen is to be used for the bitmap, the next time it is allocated, clients call

```
SetBitmapBox: PROCEDURE [box: Window.Box] RETURNS [result: Window.Box];
```

Window.rootWindow.box contains the current parameters and the client may examine it directly. The bitmap may not be changed while allocated. If an attempt is made to do this, the following error is raised.

```
BitmapChangeWhileAllocated: ERROR;
```

If the client wishes to allocate his own memory for the bitmap instead of allowing UserTerminal to do it, then the following procedure may be called. It is also subject to raising **BitmapChangeWhileAllocated**, and it is the client's responsibility to make sure that the amount of memory available at location address, is sufficient to contain a bitmap the size specified in the call to **SetBitmapBox**.

```
SetBitmapSpace: PROCEDURE [address: LONG POINTER, words: CARDINAL];
```

The following convenience routines map between screen and bitmap coordinates.

```
ScreenPlaceToBitmapPlace: PROCEDURE [Window.Place] RETURNS [Window.Place];
BitmapPlaceToScreenPlace: PROCEDURE [Window.Place] RETURNS [Window.Place];
```