InterViews Reference Manual

Version 2.6 + Updates

January 1997

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

with updates made at
Department of Computer Science
Cal Poly University
San Luis Obispo, CA 93407

NAME

InterViews − user interface library

SYNOPSIS

#include <InterViews/*class*.h>
CC ... −lInterViews −lX −lm

DESCRIPTION

The InterViews library contains a collection of C++ classes for implementing user interfaces. All interactive objects are derived directly or indirectly from the **Interactor** class. All composite interactive objects are derived directly or indirectly from the **Scene** class, which itself is derived from **Interactor.** Subclasses include Box(3I) and Glue(3I) for tiling interactors, Tray(3I) for overlapping/aligning, Frame(3I) for decorating a single interactor, Deck(3I) for paging through a list of interactors (one visible at a time), Viewport(3I) for scrolling/panning around a larger interactor, Scroller(3I) for a simple scrollbar, Panner(3I) for 2-D scrolling and zooming, and Menu(3I) for simple pop-up menus. An instance of the World(3I) subclass is used to map interactors onto a display.

A number of classes are provided for graphics input and output. These include Canvas(3I) for display regions; Painter(3I) for drawing operations; Brush(3I), Color(3I), Font(3I), and Pattern(3I) for controlling graphics characteristics; Cursor(3I) for the screen appearance of a pointing device; Event(3I) for input events, and Sensor(3I) for expressing interest in a set of possible events; Rubberband(3I) objects for interactive dragging and placement.

The basic InterViews library includes the above classes. The *libgraphic*(3I) library provides an additional set of classes for defining structured graphics and storing graphical objects in files.

All InterViews header files include the following definitions from ‘‘<InterViews/defs.h>’’:

typedef int Coord;

Display coordinates are represented uniformly as 32-bit integers; however, the units depend on context because of the possibility of coordinate transformations (see Painter(3I)). The predefined variables **cm, inches, pixels,** and **points** should be used as units if no transformations are present (e.g., ‘‘Coord x = 3∗inches’’). This convention makes the coordinate values independent of screen resolution. The variables **inch** and **point** are also provided as synonyms for **inches** and **points,** respectively.

typedef unsigned boolean;

Boolean values are represented as unsigned integers. The constants **true** and **false** are predefined.

typedef unsigned Alignment;

Alignments are used to specify the physical relationship between objects such as components in a Tray(3I). The interpretation of an Alignment is context-specific. The constants **TopLeft, TopCenter, TopRight, CenterLeft, Center, CenterRight, BottomLeft, BottomCenter, BottomRight, Left, Right, Top, Bottom, HorizCenter,** and **VertCenter** are predefined.

typedef unsigned TextStyle;

A TextStyle specifies a combination of stylistic attributes that are applied to text drawing. The style is obtained by ORing together zero or more of the predefined constants **Boldface, Underlined,** and **Reversed.** The constant **Plain** specifies an unadorned style.

#define nil 0

The symbol **nil** represents a pointer to a non-existent object. Dereferencing **nil** is an error.

int min(int, int), int max(int, int)
float min(float, float), float max(float, float)
double min(double, double), double max(double, double)

Return the minimum or maximum of two integers, floats, or doubles.

int round(double)

Return the integer closest to its argument.

NAME

Adjuster – button-like interactors for incremental scrolling and zooming

SYNOPSIS

#include <InterViews/adjuster.h>

DESCRIPTION

Adjuster objects manipulate a perspective using a pushbutton-like interface. Adjusters also provide an auto-repeat capability. Depressing and holding down any mouse button on an adjuster for a specified time causes it to repeatedly perform its operation. The auto-repeat terminates when the button is released.

Predefined subclasses of Adjuster include **Zoomer** and **Mover**. Zoomers adjust the perspective's *curwidth* and *curheight* members to suggest magnification. The user clicks on a zoomer to magnify or reduce the contents of the interactor through its perspective. The **Enlarger** zoomer subclass halves the perspective values while the **Reducer** class doubles them. Movers provide an incremental movement of the perspective in one of four directions; left, right, down, or up. The user clicks on a mover to move the perspective.

PUBLIC OPERATIONS

Adjuster(Interactor*, int = NO_AUTOREPEAT)

Adjuster constructors take a pointer to the interactor whose perspective will be adjusted and a delay (in tenths of a second) after which auto-repeat will begin once the adjuster is ‘‘pressed’’.

virtual void Highlight()
virtual void UnHighlight()

These functions define how the adjuster draws itself when it is pressed. The adjuster is highlighted when it is pressed with any mouse button and unhighlighted when the button is released. The default is to invert the adjuster when highlighted.

PROTECTED OPERATIONS

virtual void AdjustView(Event&)

This function defines how the adjuster adjusts the interactor's perspective. It is called after the adjuster is pressed and released, or repeatedly during auto-repeat. Derived classes should redefine this function to implement the desired behavior.

DERIVED CLASSES

Enlarger(Interactor*)
Reducer(Interactor*)

Enlargers and reducers adjust an interactor's perspective to enlarge and reduce the visible area by factors of one half and two, respectively. This suggests to an interactor to enlarge or reduce its contents. Auto-repeat is disabled on these adjusters.

LeftMover(Interactor*, int delay = NO_AUTOREPEAT)
RightMover(Interactor*, int delay = NO_AUTOREPEAT)
DownMover(Interactor*, int delay = NO_AUTOREPEAT)
UpMover(Interactor*, int delay = NO_AUTOREPEAT)

These subclasses of **Mover** adjust the current origin of the interactor's perspective by an amount in the corresponding direction. If the shift key is down while the mover is pressed, the amount corresponds to the perspective's *lx* or *ly* member variables; otherwise, the *sx* or *sy* values are used.

SEE ALSO

Interactor(3I), Perspective(3I)

NAME

Banner − one line title bar

SYNOPSIS

#include <InterViews/banner.h>

DESCRIPTION

A banner is an interactor that draws a one line title bar with optional left-justified, centered, and right-justified headings. The member variable *highlight* can be set to cause drawing to be performed with the foreground and background colors reversed.

The natural size of a banner depends on the size of its headings. Banners are infinitely stretchable horizontally, but fixed size vertically.

PUBLIC OPERATIONS

Banner(char* left, char* middle, char* right)

The *left* string will be left-justified, the *middle* string centered, and *right* string right-justified in the banner. A header can be omitted by passing a nil string.

void Update()

The strings can be set through the *left*, *middle*, and *right* member variables. The Update operation should be called after one or more of the strings is changed.

SEE ALSO

Interactor(3I)

NAME

Bitmap − bit mapped data

SYNOPSIS

#include <InterViews/bitmap.h>

DESCRIPTION

A *Bitmap* is a two−dimensional array of boolean values. Bitmaps can be used in Painter operations or to define window manager icons for top−level Interactors. Bitmaps can be constructed from data contained in a file, from data supplied by the application program, or from a font character. Operations include geometric transformations, common bit transformations, and individual bit manipulations. Geometric transformations and painting operations are relative to an *origin*.

PUBLIC OPERATIONS

Bitmap(const char* filename)

Create a bitmap from the named file. The file format is that produced by the X bitmap(3X) program. The bitmap origin will be the *hot spot* specified by the file or the lower left corner if the file does not define a hot spot.

Bitmap(void* data, int width, int height, int x0 = 0, int y0 = 0)

Create a bitmap from the data specified. The data format is the same as that of a bitmap(3X) file. The coordinates *x0* and *y0* specify the bitmap origin. The data is transferred into an internal format, so subsequent changes in either the passed data or the bitmap will not affect the other. Passing a nil pointer to *data* is equivalent to passing an array with all values set to *false*.

Bitmap(Font* font, int character)

Create a bitmap from the given character of the font. The bitmap width and height will reflect the actual size of the character glyph and will not usually equal the width and height of the character cell as reported by operations on the Font. The bitmap origin is set to the character origin.

Bitmap(Bitmap*)

Create a copy of the given bitmap.

int Left()
int Right()
int Top()
int Bottom()
int Width()
int Height()

These operations return the size and bounds of the bitmap.

void Transform(Transformer*)
void Scale(float x, float y)
void Rotate(float angle)

Transform the bitmap in place. *Scale* and *Rotate* provide convenient interfaces to common transformations. Transformations are relative to the bitmap origin.

void FlipHorizontal()
void FlipVertical()
void Rotate90()
void Rotate180()
void Rotate270()
void Invert()

Perform an operation on the bitmap data. *FlipHorizontal* and *FlipVertical* reflect the data about its vertical or horizontal center respectively. The *Rotate* functions move the data in a counter−clockwise direction about the bitmap center. *Invert* toggles each bit in the bitmap. In each case, the bitmap origin is unchanged.

boolean Peek(int x, int y)

void Poke(boolean bit, int x, int y)
boolean Contains(int x, int y)

Operations for reading and writing individual bits in the bitmap. If the specified coordinates are not within the bitmap bounds, *Contains* and *Peek* return false, and *Poke* has no effect.

SEE ALSO

Painter(3I), Transformer(3I)

NAME

Border, HBorder, VBorder − visual separators

SYNOPSIS

#include <InterViews/border.h>

DESCRIPTION

Border is the base class for interactors that simply display a line of a given thickness, usually to separate other interactors in a scene. **HBorder** objects have a fixed height and are infinitely stretchable and shrinkable horizontally. **VBorder** objects have a fixed width and are infinitely stretchable and shrinkable vertically.

PUBLIC OPERATIONS

HBorder(int thickness = 1)
VBorder(int thickness = 1)

Construct a border with a given thickness (default is one pixel).

SEE ALSO

Interactor(3I)

NAME

Box, HBox, VBox − tile interactors in a box

SYNOPSIS

#include <InterViews/box.h>

DESCRIPTION

A box is a scene of interactors that are tiled side-by-side in the available space. Interactors are placed left-to-right in a horizontal box, and top-to-bottom in a vertical box.

A box will try to stretch or shrink the interactors inside it to fit the available space. The natural size of a box is the sum of its elements along the major axis, and the maximum along its minor axis. A box's stretchability (shrinkability) is the sum of its elements along its major axis and the minimum of its elements along its minor axis.

PUBLIC OPERATIONS

HBox(...)
VBox(...)

Create a new box. Zero to seven interactors may be passed as arguments; the interactors are inserted into the box.

void Align(Alignment)

Set the alignment mode that the box uses to place elements along the minor axis. The default alignment for an hbox is *Bottom*; other choices are *Top* and *Center*. The default alignment for a vbox is *Left*; other choices are *Right* and *Center*.

void Insert(Interactor*)

Append an interactor to the box. Components of an hbox (vbox) will appear left-to-right (top-to-bottom) in the order in which they are inserted.

void Change(Interactor*)

Notify the box that the given interactor's shape has changed. If change propagation is true, the box will modify its own shape to reflect the change and notify its parent. Regardless of propagation, the box will recompute the positions of the component interactors and update any that have changed.

void Remove(Interactor*)

Take an element of out a box. Remove does not cause any immediate change to the other components in the box; the Change operation must be called after one or more Removes to update the component positions.

SEE ALSO

Glue(3I), Interactor(3I), Scene(3I), Shape(3I)

NAME

Brush − line style and width for a painter

SYNOPSIS

#include <InterViews/brush.h>

DESCRIPTION

A **Brush** defines the line thickness and line style for Painter operations that draw lines, curves, and unfilled shapes. The effect of these operations is as if a line-segment equal in length to the brush's width were dragged along an infinitely thin path between the specified coordinates. At each point along the path the brush is angled perpendicular to the path. As a special case, a brush width of zero specifies a minimal-width line. Many devices can render minimal-width lines more quickly than wide lines, but the resultant display may be device-dependent.

The effect of the operations is further influenced by the brush style. A solid brush paints all pixels along the path with foreground color. A dashed brush defines alternating foreground and background segments, measured along the length of the path. Foreground segments are painted with foreground color. The filling of background segments is affected by the setting of the painter's FillBg flag: if FillBg is true, background segments are painted with background color; if the flag is false, background segments are left unpainted.

The global variable ‘‘single’’ points to a Brush for drawing minimal-width solid lines.

PUBLIC OPERATIONS

Brush(int pattern[], int count, int width = 1)

Construct a brush with the specified width and line style. *Pattern* contains an array of integers that specifies the length of successive foreground and background segments. Even-numbered array indices (starting from 0) specify the length of foreground segments; odd-numbered indices specify background segments. *Count* contains the number of entries in the array. A *count* of zero specifies a solid brush.

Brush(int pattern, int width = 1)

Define a brush with a given width and a style specified by a bit vector. The least significant 16 bits of *pattern* are interpreted as a bit pattern; one bits specify foreground segments and zero bits specify background segments. A *pattern* of 0xffff specifies a solid brush.

int Width()

Return the brush's width in pixels.

SEE ALSO

Painter(3I)

NAME

ButtonState, Button, TextButton, PushButton, RadioButton, CheckBox − interactive buttons

SYNOPSIS

#include <InterViews/button.h>

DESCRIPTION

Button is a class of interactors that can set the value of a piece of state. **ButtonState** is a class of objects that contain a value and a list of buttons that can set its value. A value is either an integer or an arbitrary pointer.

TextButton is a subclass of **Button** that has an associated text label to be displayed inside or to the right of the button. **PushButton, RadioButton,** and **CheckBox** are subclasses of **TextButton** that provide particular displays for buttons. Each kind of button will display itself differently for each of the five possible cases: ‘‘enabled’’, when the button is listening to input, ‘‘disabled’’, when the button is not listening, ‘‘chosen’’, when the button's value is equal to its button state's value, ‘‘hit’’, when a button that is not chosen first receives a DownEvent, and ‘‘same-hit’’, when a button that is chosen receives a DownEvent.

When a button receives a DownEvent, it calls the virtual Refresh() to update its display. It then reads input waiting for an UpEvent. If the UpEvent coordinates are also within the button, then it calls the virtual Press(). The default Press operation sets the associated state's value to the button's value.

A button can have a list of attached buttons. When a button is not chosen its attached buttons are ‘‘disabled’’, meaning they ignore all input events.

BUTTONSTATE OPERATIONS

ButtonState()
ButtonState(int)
ButtonState(void∗)

Define a button state optionally with an initial value.

void Attach(Button∗)

Add a button to the list associated with a button state. The button will be notified when the button state's value is modified.

void Detach(Button∗)

Remove a button from the list associated with a button state.

void GetValue(int&)
void GetValue(void∗&)

Get the value of a button.

void SetValue(int)
void SetValue(void∗)

Set the value of a button state and notify all the buttons associated with the state.

void operator=(ButtonState&)

Copy the value of one button state to another. The button list associated with the source button state is not copied.

BUTTON OPERATIONS

void PushButton(const char∗ text, ButtonState∗ s, int v)
void PushButton(const char∗ text, ButtonState∗ s, void∗ v)

Construct a button with *text* in the center and a polygon on the outside. The polygon is like a rectangle, but with rounded corners. When chosen, the entire button is drawn with foreground and background colors reversed.

void RadioButton(const char∗ text, ButtonState∗ s, int v)
void RadioButton(const char∗ text, ButtonState∗ s, void∗ v)

Construct a button with a circle on the left and *text* on the right. When hit, a second circle is drawn inside the first. When chosen, a filled circle is drawn inside the outer circle.

void CheckBox(const char* text, ButtonState* s, int on, int off)
void CheckBox(const char* text, ButtonState* s, void* on, void* off)

Construct a button with a rectangle on the left and *text* on the right. When hit, a second rectangle is drawn inside the first. When chosen, two diagonal lines are drawn connecting the opposite corners of the rectangle. When a checkbox is pressed the first time, it sets *s* to *off* if the value is *on* and sets *s* to *on* otherwise. Subsequent presses swap the values of the checkbox and *s*.

void Attach(Button*)

Add to this button's list of associated buttons. These buttons are enabled when the button is chosen and disabled when it is not chosen.

void Detach(Button*)

Remove a button from this button's list of associated buttons.

void Disable()

Stop listening to input events. Disabled buttons are typically "grayed out" by drawing a half-filled rectangle in the background.

void Enable()

Start listening to input events.

void Choose()

Make the button as chosen, enabling any associated buttons.

void Handle(Event&)

If the event type is DownEvent and the button is enabled, call Refresh and read events until an UpEvent occurs. If the button is the target of the UpEvent, then call Press.

virtual void Press()

Take the appropriate action for the button being pressed. The default is to set the associated state to the button's value, which will trigger all related buttons to update their display.

virtual void Refresh()

Update the button display.

void SetDimensions(int width, int height)

Specify the natural size of the button. The default size is a function of the size of the text. This routine is obsolete; use Interactor::Reshape instead.

void UnChoose()

Mark the button as not chosen, disabling any associated buttons.

SEE ALSO

Event(3I), Interactor(3I)

NAME

Canvas − region for graphics output

SYNOPSIS

#include <InterViews/canvas.h>

DESCRIPTION

A canvas is an area for graphics output associated with an interactor. Most interactors do not need to perform any operations on a canvas; they simply pass a canvas as an argument to graphics operations. In this case, it is not necessary to include ‘‘canvas.h’’.

PUBLIC OPERATIONS

Canvas(int width, int height)

Create an offscreen canvas with the given size and height.

void Clip(Coord, Coord, Coord, Coord)

Clip drawing to the canvas to the given rectangle.

void ClipOn()
void ClipOff()

Temporarily turn use of a canvas’ clipping rectangle on or off.

int Height()

Return the height of a canvas in pixels.

boolean IsClipped()

Return whether output to the canvas is currently clipped.

void NoClip()

Delete the clipping rectangle for with a canvas.

void SetBackground(Color*)

Associate a background color with the canvas.

CanvasStatus Status()

Return the current status of a canvas, which is one of CanvasMapped (assigned an area on the display), CanvasUnmapped (not assigned an area), and CanvasOffscreen (assigned an area in offscreen memory).

int Width()

Return the width of a canvas in pixels.

SEE ALSO

Interactor(3I), Painter(3I)

NAME

ChiefDeputy − buffer communication with remote object space

SYNOPSIS

#include <InterViews/chief.h>

DESCRIPTION

A chief deputy mutiplexes communication with a remote object space across a single connection.

PUBLIC OPERATIONS

ChiefDeputy(Connection∗)

Construct a chief deputy that sends messages over the given connection.

ObjectTag Tag()

The tag for a chief deputy corresponds to the remote object space itself, whose tag is by convention zero.

Connection∗ GetServer()

Return the connection associated with the remote object space.

void Alloc(void∗& dst, ObjectTag, int op, int len)

Allocate a message in the buffer.

int PackString(const char∗, void∗ msg)
int PackString(const char∗, int len, void∗ msg)

Put a string or substring into a message buffer. The buffer area is assumed to have already been allocated.

void Msg(ObjectTag, int op)
void StringMsg(ObjectTag, int op, const char∗)
void IntegerMsg(ObjectTag, int op, int val)

Short-hand for allocating a message and storing fields in the cases where a message has no parameters, a single string parameter, or a single integer parameter.

void Sync()

Force all messages to be sent to the remote object space.

int ReadReply(void∗ reply, int len)
void GetReply(void∗ reply, int len)
void GetString(const char∗& str, int len)

Read a reply from the object space, first flushing any messages in the buffer. ReadReply returns the actual size of the reply from the object space. GetReply will abort if the size is not the size requested. GetString will first read the size of a string and then the string itself.

SEE ALSO

ChiefDeputy(3I), ObjectSpace(3I)

NAME

Color − colors for a painter

SYNOPSIS

#include <InterViews/color.h>

DESCRIPTION

A **Color** defines a mix of rgb (red, green, and blue) intensities. The global variables ‘‘black’’ and ‘‘white’’ point to predefined colors of zero and full intensity output, respectively. Note that the displayed intensities of a color will depend on the characteristics of the workstation.

PUBLIC OPERATIONS

Color(const char∗ name)

Construct a color with intensities associated with *name*.

Color(int red, int green, int blue)

Construct a color with the specified red, green, and blue intensities. Zero represents no intensity; 65535 represents full intensity.

Color(int entry)

Construct a color with intensities associated with a color map entry.

void Intensities(int& red, int& green, int& blue)

Get the intensities used to create the color.

void DisplayIntensities(int& red, int& green, int& blue)

Get the intensities used to display the color.

int PixelValue()

Return the color’s index in its color map.

boolean Valid()

Return whether the color is meaningful. This call can be used after Color(name) to see if the name is recognized as a color.

SEE ALSO

Painter(3I)

NAME

Connection − interprocess communication

SYNOPSIS

#include <InterViews/connection.h>

DESCRIPTION

Connection provides a simple interface to interprocess communication. A general connection can be made between processes on the same or different machines. A local connection can only be made between processes on the same machine.

PUBLIC OPERATIONS

Connection()
˜Connection()

Construct a connection, initially not associated with any process.

Connection(int)

Construct a connection associated with the given Unix file descriptor.

void CreateService(const char∗ host, int port)
void CreateLocalService(const char∗)

Create a connection and bind it to a particular address. In the general case, the address is defined by the host name and Internet port number. In the local case, a string name is used.

boolean OpenService(const char∗ host, int port)
boolean OpenLocalService(const char∗)

Establish a connection with another process. It is assumed the process has already created the service. If no connection can be made, the open functions return false.

Connection∗ AcceptClient()

Create a new connection for communicating with a client. AcceptClient is called by servers that receive input on a general service connection.

int Pending()

Return the number of bytes waiting to be read from a connection.

int Read(void∗ buf, int n)

Read up to *n* bytes from the connection into *buf* Return the actual number of bytes read.

int Write(const void∗ buf, int n)

Write *n* bytes from *buf* to the connection. Return the actual number of bytes written.

int WritePad(const void∗ buf, int n, int pad)

Write *pad* bytes to the connection. The first *n* bytes are from *buf*, any remaining are zero. Return the actual number of bytes written.

int Descriptor()

Return the file descriptor associated with the connection.

void Close()

Close the connection. Subsequent operations, such as Read and Write, will return an error code of -1.

NAME

Control, ControlState − select and execute an action

SYNOPSIS

#include <InterViews/control.h>

DESCRIPTION

A control is a monoscene that allows an action to be selected and executed. The component of a control determines the appearance associated with the control. A control state is a subject that coordinates a group of related controls. The control state contains a pointer to the currently selected control (if any).

Controls are useful for grabbing input and moving over potentially selectable items. Examples of controls include popup menus, pulldown menus, and menu items. It is generally not necessary to call functions directly on a control.

PUBLIC OPERATIONS

Control(Interactor*)

Construct a new control with the given interactor as the component.

~Control

Destruct a control. Because a control is a monoscene, deleting it implies deleting its component interactor.

ControlState* State()
void SetState(ControlState*)

Get or set the control state associated with a control.

virtual void Handle(Event&)

Controls are initially inactive. A DownEvent causes a control to become active. If an active control receives an EnterEvent, it will set the current selection to itself. If an active control receives a LeaveEvent, it will set the current selection to *nil*.

virtual void Select()

Highlight the control and call the protected virtual functions Open and Grab. The base class Open does nothing, but subclasses may insert a new interactor such as a pulldown menu. Grab reads input and only passes through an event if its target is an active control.

virtual void Unselect()

Call the protected virtual function Close and then unhighlight the control.

virtual void Do()

Implement a selected action. When a control sees an UpEvent, it calls Do on the current selection.

SEE ALSO

Event(3I), Interactor(3I), Menu(3I), Scene(3I)

NAME

Cursor − pointing device image

SYNOPSIS

#include <InterViews/cursor.h>

DESCRIPTION

A cursor specifies the image that tracks a pointing device's position. Cursors are defined by two bitmaps and a ''hot spot''. A mask bitmap specifies which pixels are to be drawn, and a pattern bitmap specifies which pixels are in foreground color and which in background color. The hot spot specifies the location of the pointing device relative to the cursor's lower-left corner. A number of common cursors are predefined and can be referenced by global variables declared in the include file. These include ''arrow'' (pointing to upper left), ''ltextcursor'' (an 'I' text cursor with hot spot on the left), ''rtextcursor'' (hot spot on the right), ''lowerleft'' (an 'L'), ''lowerright'' (mirrored 'L'), ''upperleft'' (upside-down 'L'), ''upperright'' (upside-down and mirrored 'L'), ''crosshairs'', and ''hourglass''.

PUBLIC OPERATIONS

typedef int CursorPattern[cursorHeight];

Cursor(short x, short y, CursorPattern pat, CursorPattern mask, Color* f, Color* b)

Create a cursor with the given data and a hot spot at (*x*, *y*). Ones in *pat* specify pixels which should be displayed in the foreground color *f* and zeros specify pixels which should be in the background color *b*. Ones in *mask* specify which pixels should be visible. The first pattern element defines the topmost row of pixels, and the least significant bit of each element defines the rightmost column.

Cursor(Bitmap* pat, Bitmap* mask, Color* f, Color* b)

Create a cursor from the given bitmaps. The hot spot is set to the origin of the *pat* bitmap.

Cursor(Font*, int pat, int mask, Color* f, Color* b)

Create a cursor from font glyphs. *Pat* and *mask* are the font characters to use as the pattern and mask bitmaps, respectively. The hot spot is set to the character origin.

Cursor(int id, Color* f, Color* b)

Create a ''standard'' cursor using a window system specific identifier. Particular values of *id* are usually defined in an include file. For example, standard X11 cursors are defined in the file <X11/cursorfont.h>.

NAME

Deck − a stack of interactors on top of each other

SYNOPSIS

#include <InterViews/deck.h>

DESCRIPTION

A deck is a scene containing interactors that are conceptually stacked on top of each other. Only one interactor is visible at a time. A deck is useful to implement ‘‘paging’’ through a document, or where only one of several similar components should be visible at a time, as in applications using alternate tool pallets or menus with alternate entries.

A deck will try to stretch or shrink the visible interactor to fill the total space. The natural size of a deck is the maximum of its components.

A deck uses its *perspective* to specify which component is visible. The components are numbered in the order in which they are inserted into the deck, starting with one. A scroller or other interactor which operates on the perspective can be used to allow a user to specify the component to view. Either horizontal or vertical scrolling can be used.

PUBLIC OPERATIONS

Deck() Construct a new deck.

void Insert(Interactor*)

Add an interactor to the deck.

void Change(Interactor*)

Notify the deck that an interactor's shape has changed. If propagation is on, the deck recomputes its own shape and notifies its parent.

void Remove(Interactor*)

Take an interactor out of a deck.

void Flip(int count = 1)
void FlipTo(int position)
void Top()
void Bottom()

Perform operations on the perspective to specify the component to view. Flip specifies the component relative to the current component. A positive value of *count* specifies a component nearer to the ‘‘bottom’’ of the deck (the components inserted last), and a negative value specifies a component nearer to the ‘‘top’’. FlipTo specifies a component by its absolute position in the deck. A positive value of *position* specifies a position from the top, starting at 1. A negative value specifies a position from the bottom, starting at -1. Top and Bottom are equivalent to FlipTo(1) and FlipTo(-1) respectively.

SEE ALSO

Interactor(3I), Scene(3I), Shape(3I)

NAME

Deputy − pack and send remote messages

SYNOPSIS

#include <InterViews/deputy.h>

DESCRIPTION

A deputy object performs operations on remote objects. A caller performs a local call on the deputy, which packs the parameters into a message. Deputies place the message into a buffer managed by a ChiefDeputy(3I), which then take care of sending the message to the remote space.

PUBLIC OPERATIONS

Deputy()

Deputy(ChiefDeputy∗)

Construct a deputy, optionally specifying the chief deputy that will manage the message buffer. The chief deputy is accessible through a protected member *chief*.

void Sync()

Force all messages to be sent to the remote object space. Without doing a Sync, messages are buffered locally to reduce message-passing overhead.

ObjectTag Tag()

Return a tag to use in messages to identify the remote object. Because the client defines the tag, a deputy uses its address (this pointer) as its tag.

Connection∗ GetServer()

Return the connection associated with the remote object.

SEE ALSO

ChiefDeputy(3I), ObjectSpace(3I)

NAME

Dialog − popup dialog interactor

SYNOPSIS

#include <InterViews/dialog.h>

DESCRIPTION

A dialog is an interactor that handles input for a dialog box. The dialog layout is specified by a separate interactor passed to the dialog constructor. The dialog completes when the value of an associated button state becomes non-zero. This button state is typically set by one or more buttons in the component interactor.

PUBLIC OPERATIONS

Dialog(ButtonState*, Interactor*, Alignment = Center)

Construct a new dialog with the given button state and interactor. The alignment is used by the Popup operation.

virtual boolean Accept()

Set the dialog button state to zero and loop reading events until the button state's value becomes non-zero. Each event is passed to is target's Handle operation. Return true if the value is one, false otherwise.

virtual boolean Popup(Event&, boolean placed = true)

Insert the dialog into the world associated with the given event. If *placed* is true, align the dialog around the event's (x, y) coordinate. Otherwise, the dialog is inserted without specifying a position. Popup then calls the Accept operation and returns its value.

SEE ALSO

Button(3I), Interactor(3I), World(3I)

NAME

Event − input events

SYNOPSIS

#include <InterViews/event.h>

DESCRIPTION

Events represent user input. The *target* field of an event indicates the interactor whose current sensor is interested in the event. For keyboard and button events, the *timestamp* field indicates when the event occured and the (*x*, *y*) fields contain the coordinates of the input pointing device relative to the target interactor. The *eventType* field specifies what kind of event occurred. The following types of events are possible:

MotionEvent

The position of the pointing device changed.

DownEvent
UpEvent

The *button* field identifies a button that was pressed or released. The constants LEFTMOUSE, MIDDLEMOUSE, and RIGHTMOUSE identify up to three buttons on the pointing device. The fields *leftmouse*, *middlemouse*, and *rightmouse* are *true* if the corresponding button is currently down as well (so chording can be handled).

KeyEvent

A key is pressed. Since many keyboards do not support up events for arbitrary keys; no event is generated when the key is released. The *button* field contains the key code. The *keystring* field points to a string of length *len* that represents the ASCII mapping of the key. The fields *control*, *meta*, *shift*, and *shiftlock* are *true* if the corresponding key is currently down.

EnterEvent
LeaveEvent

The pointing device enters or leaves the canvas associated with the target interactor.

ChannelEvent

Data is available on the file descriptors identified in the *channel* field. The field is a bitset, with bit 0 set if data is ready on file descriptor 0, bit 1 set if data ready on file descriptor 1, etc.

TimerEvent

The time limit defined by the sensor has expired without any input events occurring.

FocusInEvent
FocusOutEvent

The canvas associated with the target interactor gains or loses focus for keyboard events.

PUBLIC OPERATIONS

void GetAbsolute(Coord& x, Coord& y)
void GetAbsolute(World*&, Coord& x, Coord& y)

GetAbsolute computes the event coordinates relative to the root scene and optionally assigns a pointer to that root scene.

SEE ALSO

Sensor(3I), World(3I)

NAME

FileBrowser − browse and/or select file names

SYNOPSIS

#include <InterViews/filebrowser.h>

DESCRIPTION

FileBrowser is a StringBrowser that displays file names. In addition to the standard StringBrowser constructor parameters, the FileBrowser constructor takes a directory name as an argument. The FileBrowser scans the corresponding directory, appending the file names to its list of strings. FileBrowser lets the user choose a file name, and the application queries the FileBrowser for the chosen file.

PUBLIC OPERATIONS
done)

FileBrowser(ButtonState*, const char* dir, int rows, int cols, boolean uniqueSel, int highlight, const char*

FileBrowser(const char* name, ButtonState*, const char*, int, int, boolean, int, const char*)

Create a new FileBrowser object. The ButtonState, *rows, cols, uniqueSel, highlight, done*, and *name* parameters have semantics identical to those of the corresponding StringBrowser parameters. The *dir* parameter specifies the directory to search for file names. Internally, FileBrowser prepends *dir* to relative path specifications in operations that require them, thus avoiding a chdir(2).

boolean IsADirectory(const char*)

Return whether or not the given relative path corresponds to a valid directory.

boolean SetDirectory(const char* newdir)

Change the search directory to *newdir* if that directory is accessible. The function returns true if successful.

const char* GetDirectory()

Return the name of the search directory.

const char* ValidDirectories(const char* path)

Given a path string, return the largest substring (starting from the beginning of *path*) that represents a valid path. The substring is returned in a static buffer and should be copied before use.

const char* Normalize(const char* path)

Return a normalized version of *path* with all spurious slashes, dots, and tildes interpreted. The normalized path is returned in a static buffer and should be copied before use.

const char* Path(int i)

Return the *i*th string as it appears in the FileBrowser. This function prepends the search directory to the string, normalizes the concatenation, and returns the result.

SEE ALSO

Interactor(2I), Button(3I), StringBrowser(3I)

NAME

FileChooser − dialog box for choosing a file name

SYNOPSIS

#include <InterViews/filechooser.h>

DESCRIPTION

FileChooser is a StringChooser that provides a complete dialog box interface to selecting a file name. The interface consists of two lines of descriptive text, a StringEditor for entering the file name explicitly, a FileBrowser, and two PushButtons, one for accepting and one for canceling the dialog.

PUBLIC OPERATIONS

Alignment)

FileChooser(const char* title, const char* subtitle, const char* dir, int rows, int cols, const char* acceptLabel,

FileChooser(const char* name, const char*, const char*, const char*, int, int, const char* accept, Alignment)

Construct a new FileChooser with the given *title* and *subtitle*, search directory *dir*, FileBrowser size specification *rows* and *cols*, label for the accept PushButton *acceptLabel*, and pop-up alignment. These constructors create all the elements of the dialog box and compose them into a default layout.

void SetTitle(const char*)

void SetSubtitle(const char*)

Change the text in title or the subtitle. The new text will appear immediately in the dialog when either function is called.

void SelectFile()

Highlight the text in the StringEditor.

PROTECTED OPERATIONS

FileChooser(ButtonState*, Alignment = Center)

Construct a new FileChooser with the given button state and pop-up alignment *without* creating FileBrowser, StringEditor, title, subtitle, or PushButton instances. This constructor should be called in subclasses requiring a specialized StringEditor or FileBrowser.

FileChooser(ButtonState*, const char* dir, int rows, int cols, Alignment)

Construct a new FileChooser (along with its StringEditor and its FileBrowser) with the given button state, search directory, FileBrowser size specification, and pop-up alignment *without* composing them into the default layout and without creating the title, subtitle, or PushButtons. This constructor should be used by FileChooser subclasses that define a specialized look and feel.

SEE ALSO

Button(3I), Dialog(3I), FileBrowser(3I), StringChooser(3I), StringEditor(3I)

NAME

Font − text font for a painter

SYNOPSIS

#include <InterViews/font.h>

DESCRIPTION

A font defines the representation of characters on the screen. The global variable ‘‘stdfont’’ is a small, fixed-width font that Painter(3I) operations use by default.

PUBLIC OPERATIONS

Font(const char* str)
Font(const char* str, int length)

Construct a font named by the given string or substring.

int Baseline()

Return the font’s baseline in pixels.

boolean FixedWidth()

Return whether the font contains fixed width characters.

int Height()

Return the font’s height in pixels.

int Index(const char* string, int offset, boolean between)
int Index(const char* string, int len, int offset, boolean between)

Return the index in a string or substring of the character that would be *offset* pixels from the left if the string were displayed. If *between* is *false*, the index of the character that contains *offset* is returned; otherwise the index of the character following the between-character space that is nearest *offset* is returned. In either case a negative offset will return an index of zero and an offset beyond the end of the string or substring will return an index equal to the length of the string.

int Width(const char* string)
int Width(const char* string, int len)

Return the width in pixels that the string or substring would have if it were displayed in this font.

boolean Valid()

Return whether the font is meaningful.

SEE ALSO

Painter(3I)

NAME

Frame, ShadowFrame, MarginFrame, ShowFrame, TitleFrame, BorderFrame − surrounding interactor

SYNOPSIS

#include <InterViews/frame.h>

DESCRIPTION

A frame is a monoscene subclass that outlines another interactor. A shadow frame draws an outline with a drop shadow. A margin frame surrounds its component with horizontal and vertical glue.

A show frame is an abstract class that changes its appearance in response to an EnterEvent or LeaveEvent. A title frame provides a banner that it displays in reverse video when it receives an EnterEvent. A border frame draws an outline using a solid pattern when it receives an EnterEvent and draws the outline using a gray pattern when it receives a LeaveEvent.

PUBLIC OPERATIONS

Frame(Interactor*, int width = 1)

Construct a frame. If the given interactor is not nil, it is inserted as the frame's component. The frame extends *width* pixels outside of the interactor.

ShadowFrame(Interactor*, int h = 1, int v = 1)

Construct a shadow frame. The *h* and *v* ShadowFrame parameters specify the width of the horizontal and vertical shadows, respectively. Positive values for these parameters specify shadows falling to the right and below the interactor, respectively; negative values specify shadows falling to the left and above.

MarginFrame(Interactor* = nil, int margin = 0)
MarginFrame(Interactor*, int margin, int shrink, int stretch)
MarginFrame(Interactor*, int hmargin, int vmargin)
MarginFrame(Interactor*, int hm, int hshr, int hstr, int vm, int vshr, int vstr)

Construct a margin frame. The first constructor puts rigid, fixed-sized glue around the component, while the second surrounds it with fixed-sized glue of the given shrinkability and stretchability. Independent horizontal and vertical margins are set with the third constructor. The last constructor allows full specification of the margin frame's horizontal and vertical margins, shrinkabilities, and stretchabilities.

virtual void ShowFrame::HandleInput(Event&)

ShowFrame objects listen for EnterEvent and LeaveEvent. When an EnterEvent is received, ShowFrame::Handle calls InsideFrame(true); when an LeaveEvent is received, it calls InsideFrame(false). For any other event, it calls HandleInput. ShowFrame::InsideFrame does nothing. ShowFrame::HandleInput calls Handle on the frame's component.

TitleFrame(Banner*, Interactor*, int width = 1)

Construct a title frame with the given banner. TitleFrame::InsideFrame highlights and unhighlights the banner.

BorderFrame(Interactor*, int width = 1)

Construct a border frame. BorderFrame::InsideFrame highlights and unhighlights the border.

PROTECTED OPERATIONS

Frame(Interactor*, int, int, int, int)

This constructor is used by subclasses to explicitly set the widths of each line in the outline. The integers specify the width of the left, bottom, right, and top lines, respectively.

SEE ALSO

Banner(3I), Event(3I), Glue(3I), Scene(3I)

NAME

Glue, HGlue, VGlue − open space between interactors

SYNOPSIS

#include <InterViews/glue.h>

DESCRIPTION

A glue object is an interactor that is used to define variable-size space between interactors in boxes. The **HGlue** subclass is for space in a horizontal box; **VGlue** for space in a vertical box. Along its minor axis (vertical for **HGlue**, horizontal for **VGlue**), glue has a natural size of zero and infinite stretchability.

PUBLIC OPERATIONS

HGlue(int natural = 0, int stretch = hfil)
VGlue(int natural = 0, int stretch = vfil)

Construct glue of a minimum size. The glue can be stretched arbitrarily, but cannot shrink.

HGlue(int natural, int shrink, int stretch)
VGlue(int natural, int shrink, int stretch)

Construct glue with a given natural size, shrinkability, and stretchability. If *shrink* is larger than *natural* the glue potentially can have ‘‘negative size’’, meaning interactors will overlap in a box.

SEE ALSO

Box(3I), Interactor(3I), Shape(3I)

NAME

Interactor − base class for interactive objects

SYNOPSIS

#include <InterViews/interactor.h>

DESCRIPTION

Interactor is the base class for all interactive objects. Every interactor has a *shape* member variable that defines the desired characteristics of screen space in terms of size, shrinkability, and stretchability. This information is used to allocate display space for the interactor and the interactor's *canvas* member variable is set to the actual space obtained. The lower left corner of the canvas is addressed by (0, 0); the upper right by the member variables (*xmax*, *ymax*).

The *input* member variable is the normal sensor for reading events. The *output* member variable is the standard painter for performing graphics operations. Interactors generally should not set *output*; it will either be inherited (and shared) from the interactor's parent or set by user customization attributes.

An interactor may optionally define the *perspective* member variable to represent the portion of total area that the interactor is displaying. Perspectives allow interactors to coordinate with other interactors, such as scrollers, that want to control the display area.

An interactor also may specify certain characteristics of the interactor's canvas, such as whether it is read-only or read/write, whether its contents should be saved when not visible. Top-level interactors may specify an associated string name and icon. Interactors also may specify the visual format and interpretation of the input pointing device (e.g., mouse cursor) when it is inside the interactor's canvas.

To be mapped to some portion of the display, an interactor must have been inserted into a scene, called its *parent*. The interactor will be mapped when its parent is mapped. The root scene for the display is an instance of World(3I) and is always mapped.

CONSTRUCTORS

Interactor()

Construct an interactor. The *input* sensor and *output* painters are initialized to *nil*.

Interactor(const char∗ name)

Construct an interactor associated with the string *name*. The string is used to customize the behavior of the interactor according to user preferences. Behavior can also be customized on a per-class basis using the subclass name. Although not explicitly documented, an instance of any object inheriting from interactor may be constructed with an additional argument (appearing first) containing the string name. For example, both ‘‘HGlue(natural, stretch)’’ and ‘‘HGlue("somename", natural, stretch)’’ are valid.

˜Interactor()

The base destructor automatically deletes the base fields, including *shape*, *canvas*, *input*, and *output*.

CONFIGURATION

void Align(Alignment, int w, int h, Coord& l, Coord& b)

Return the coordinates at which an object with the given width and height will have the given alignment within the interactor's canvas.

void SetClassName(const char∗)
const char∗ GetClassName()
void SetInstance(const char∗)
const char∗ GetInstance()

Set/get the class or instance name for an interactor. The names are used to determine user style preferences. SetClassName and SetInstance can only be performed by subclasses.

void Config(Scene∗ s)

Configure this interactor and its descendants according to user preferences. The scene *s* is assumed to be the interactor's parent and is used to inherit attribute values. This operation need not be called explicitly; it is called automatically when the ancestors of an interactor become known (e.g., when the interactor or an ancestor is inserted into a world).

Configuration involves a traversal of the interactor hierarchy. For each interactor in the hierarchy, the *output* painter is either inherited from its parent or copied from its parent if there are user preferences specific to the interactor for painter attributes such as colors, font, and brush. For example, suppose the user preference is ‘‘A*B*font:9x15’’ and the interactor hierarchy is ‘‘A.B.C.D’’ (each ‘‘.’’ representing a nesting level in the hierarchy). Interactors A and B will share the same output painter, C will copy B’s output and change the font to ‘‘9x15’’, and D will share C’s output.

After assigning the output painter, configuration is performed recursively on any children interactors. The final step at each node in the traversal is to call the virtual Reconfig operation.

virtual void Reconfig()
: Perform any configuration specific to a particular interactor. This operation should minimally compute the interactor’s shape based on the shape of its children and/or the characteristics of its output painter (e.g., font). It can also retrieve user preferences specific to this interactor’s class or instance name using GetAttribute.

const char* GetAttribute(const char*)
: Retrieve the value of a user preference with the given name. GetAttribute searches through the global PropertySheet(3I) looking for the most specific match to the current context. If called from Reconfig, the context contains this interactor’s class and instance names as well as those of all ancestors. If called outside the configuration traversal, the context is only the top-level class and instance (World’s const char* argument and the application’s name).

virtual void Reshape(Shape&)
Shape* GetShape()
: Set/get the shape of an interactor. Reshape is a a suggestion that an interactor’s shape should change to the given one. The default operation sets the interactor’s shape to the new shape and calls Scene::Change on the interactor’s parent. Suggested shape information may be lost when an interactor is configured; thus, it is best to avoid use of Reshape. The same affect can usually be achieved by putting the interactor in a box along with a particular shape of glue.

void SetCursor(Cursor*)
Cursor* GetCursor()
: Set/get the cursor that will be displayed when the pointing device is inside the interactor’s canvas. If the interactor does not explicitly set its cursor, it will use its parent’s cursor. GetCursor returns nil in this case.

INTERACTOR HIERARCHY

Scene* Parent()
: Return the interactor’s parent or *nil* if the interactor has not been inserted into a scene.

Scene* Root()
: Return the interactor’s root ancestor or *nil* if the interactor has no parent. Root effectively returns the world if an interactor is mapped but it can also return an unmapped scene if the root ancestor itself has not been inserted into the world.

World* GetWorld()
: Return a pointer to the world the interactor has been inserted into or *nil* if the interactor’s root ancestor is not mapped.

void GetRelative(Coord& x, Coord& y, Interactor* = nil)
: Map coordinates that are relative to this interactor’s canvas to be relative to another interactor’s canvas. If the other interactor is *nil*, then the coordinates are made relative to the world.

virtual void GetComponents(Interactor, int, Interactor**&, int&)**
: Construct an array of pointers to the interactors contained within this interactor. The first and second parameters specify an array of interactors that is already allocated. This array is used if it

is large enough, otherwise a new array is allocated from free store. The third and fourth parameters return the which array was used and the actual number of components. This operation is only defined by scenes; the default operation sets the number of elements to zero.

OUTPUT

Canvas∗ GetCanvas()
Return the interactor's canvas, which may be *nil* if the interactor is not mapped to a display.

virtual void Draw()
Draw is used to display the contents of an interactor, including the contents of any interior interactors. The default Draw operation calls Redraw(0, 0, *xmax*, *ymax*). Interactors usually don't need to redefine Draw unless they contain interior interactors (i.e., scene subclasses); most simple interactors redefine only Redraw.

virtual void Highlight(boolean)
Turn highlighting on or off, depending on whether the parameter is true or false. The default operation is a nop.

void Sync()
void Flush()
Sync waits until any pending operations have completed. Flush makes sure the local buffer of pending operations (if any) is sent to the display. An input operation will do a Sync automatically if it would block; thus, applications generally need not call Sync or Flush explicitly.

INPUT

void Listen(Sensor∗)
When an interactor is mapped onto a display, its input interest is determined by its *input* sensor. A different sensor can be specified with the Listen operation. To switch back to *input*, call Listen(*input*).

void Read(Event&)
Each application has a single input queue of events. Any interactor can use Read to take the next event from the queue. Redraw and Resize operations may be called as a side effect of a Read (or any input operation). The *target* field of the event specifies the interactor for which the event is intended, which is not necessarily the same as the interactor that performed the Read. The target is normally the interactor whose canvas is under the pointing device.

void UnRead(Event&)
UnRead puts an event back on the input queue as if it had never been read.

virtual void Handle(Event&)
When an interactor wishes to pass an event to another interactor, it calls the other interactor's Handle operation. Thus, input flow control can be either procedural with Read or event-driven with Handle.

void Run()
Run implements a simple event dispatching loop. It calls Read to get the next event and passes the event to the target interactor via Handle. The loop terminates if the Handle operation sets the event's target to nil.

void QuitRunning(Event&)
QuitRunning sets the event's target to nil. A Handle operation can call it to make Run exit its event dispatching loop.

boolean Check()
Check determines whether an event of interest has occurred.

void Poll(Event&)
Poll sets an event to reflect the current input state. Input polling can be wasteful of cycles and should be avoided if possible.

int CheckQueue()

CheckQueue returns the number of input packets that have been queued within the application. The event queue manager always reads as much information as possible from input; thus, a single Read might store many events in a local buffer. Subsequent reads can simply access the buffer. This buffer can include out-of-band packets, such as those requiring a Redraw. The number returned by CheckQueue does not correspond, therefore, to the actual number of input events.

VIEWS

virtual void Adjust(Perspective&)

Adjust suggests to an interactor that its perspective should change to the given perspective; the interactor may choose to accept any part of the new perspective and must ensure that the parameter matches its (new) perspective before returning. Adjust can be used by another interactor to scroll, pan, or zoom an interactor.

Perspective* GetPerspective()

GetPerspective returns the perspective associated with an interactor or nil if the interactor has not assigned one.

virtual void Update()

Change the display to reflect some change in state that the interactor depends on. This operation is used in a number of contexts. One example is in managing perspectives. If an interactor changes its perspective (e.g., the total of size of what it is displaying changes), it must notify its perspective, which in turn calls Update on the interactors that access the perspective (such as a scroller).

TOP-LEVEL INTERACTORS

void SetName(const char*)
const char* GetName()

Set/get the name associated with the top-level interactor. Window managers will display the name in title bars. If no name is set explicitly, then the name of the application will be assigned to top-level interactors.

void SetGeometry(const char*)
const char* GetGeometry()

Set/get the geometry specification associated with the top-level interactor. This operation must be performed before an interactor is mapped. See *World(3I)* for more information about geometry specifications. The InterViews library automatically stores a geometry with a top-level interactor if a resource or command-line option sets a ‘‘.geometry: value’’ property associated with the interactor and automatically fetches it again to parse it before creating the interactor’s canvas.

void SetCanvasType(CanvasType)
CanvasType GetCanvasType()

Set/get the type of canvas desired for an interactor. This operation must be performed before an interactor is mapped. The possible canvas types are CanvasShapeOnly, meaning the interactor performs no input or output (e.g., glue), CanvasInputOnly, meaning the interactor performs no output, CanvasInputOutput, which is the default, CanvasSaveUnder, which suggests that the interactor will be mapped for a short time (e.g., a popup menu) and that the information under the canvas should be saved, CanvasSaveContents, which suggests that Redraw calls are expensive and should be avoided by caching the display, and CanvasSaveBoth, which requests both CanvasSaveUnder and CanvasSaveContents.

void SetInteractorType(InteractorType)
void GetInteractorType(InteractorType)

Set/get the type of interactor seen by window managers. This operation must be performed before an interactor is mapped. The possible interactor types are InteriorInteractor, which is the default, PopupInteractor, which gets inserted into the world without any interaction with window managers, TransientInteractor, which suggests to window managers that they not ask the user to place this interactor nor should they decorate it, ToplevelInteractor, which suggests to window managers that they treat this interactor as part of a group of top-level interactors owned by the same application, ApplicationInteractor, which tells window managers this interactor is the application's "main" interactor, and IconInteractor, which tells the InterViews library to only create a canvas for this interactor, not map it, when Insert is called. Most applications call one of the World methods InsertPopup, InsertTransient, InsertToplevel, InsertApplication, or InsertIcon rather than call SetInteractorType directly.

void SetGroupLeader(Interactor*)
Interactor* GetGroupLeader()
void SetTransientFor(Interactor*)
Interactor* GetTransientFor()

Set/get additional information that ToplevelInteractor and TransientInteractor interactors need to provide window managers with. ToplevelInteractor interactors need to tell window managers which top-level interactor is the leader of the interactor group they belong to. TransientInteractor interactors need to tell window managers which interactor they are transient on behalf of, such as a popup dialog box for an application. Most applications provide the additional information when they call one of the World methods InsertToplevel or InsertTransient rather than call SetGroupLeader or SetTransientFor directly.

void SetIconBitmap(Bitmap*)
Bitmap* GetIconBitmap()

Set/get the bitmap that a top-level interactor wants window managers to display when they iconify this interactor. Most window managers will display the specified bitmap although some may clip or tile the bitmap if it's not the size they expect.

void SetIconMask(Bitmap*)
Bitmap* GetIconMask()

Set/get the bitmap that a top-level interactor wants window managers to mask the bitmap set by SetIconBitmap with. Some window managers will use the mask to create the appearance of irregularly shaped icons.

void SetIconInteractor(Interactor*)
Interactor* GetIconInteractor()

Set/get the icon interactor that a top-level interactor wants window managers to map when they iconify this interactor. The InterViews library automatically creates the icon interactor's canvas if necessary. Most window managers will map the icon interactor when iconifying this interactor in preference to showing a bitmap even if a bitmap was set by SetIconBitmap.

void SetIconGeometry(const char*)
const char* GetIconGeometry()

Set/get the geometry that a top-level interactor wants to give its icon interactor or the position that a top-level interactor wants window managers to give its icon bitmap. The InterViews library automatically stores an icon geometry with a top-level interactor if a resource or command-line option sets an ‘‘.iconGeometry: value’’ property associated with the interactor and automatically fetches it again to parse it before creating the icon interactor's canvas or setting a hint to tell window managers where to position the bitmap.

void SetStartIconic(boolean)
boolean GetStartIconic()

Set/get a hint to window managers that they should iconify this top-level interactor immediately when it starts up. This operation must be performed before an interactor is mapped. The InterViews library automatically sets or clears this hint on a top-level interactor before mapping it if a resource or command-line option sets an ‘‘.iconic: on/off’’ property associated with the interactor.

void Iconify()
void DeIconify()

Iconify (DeIconify) asks window managers to unmap the toplevel interactor (icon) and map the icon (interactor).

INTERNAL OPERATIONS

virtual void Redraw(Coord l, Coord b, Coord r, Coord t)

The Redraw operation is called when some portion of the Interactor needs to be redrawn, presumably because it was previously obscured. The Redraw operation should NOT redraw interior interactors; the Interviews library or the Draw operation will call their Redraw operations automatically. The default Redraw operation does nothing.

virtual void RedrawList(int n, Coord l[], Coord b[], Coord r[], Coord t[])

RedrawList notifies an interactor that several areas of its canvas need to be redrawn, presumably because it was raised to the top of other canvases. The default RedrawList operation redraws each area separately with Redraw.

virtual void Resize()

Resize notifies an interactor that its canvas has been created or modified. Only scenes are typically concerned with Resize, as they must place their component interactors within the new or resized canvas. The default Resize operation does nothing.

virtual void Activate()
virtual void Deactivate()

Activate and Deactivate notify an interactor that its canvas has been mapped (unmapped), which occur when window managers deiconify (iconify) the interactor. Interactors that perform a lot of computation or display a lot of information may wish to save cpu cycles when "deactivated." The default Activate and Deactivate operations do nothing.

SEE ALSO

Bitmap(3I), Canvas(3I), Painter(3I), Perspective(3I), Scene(3I), Sensor(3I), Shape(3I), World(3I)

NAME

Menu, MenuItem, MenuBar, PopupMenu, PulldownMenu, PullrightMenu − menus

SYNOPSIS

#include <InterViews/menu.h>

DESCRIPTION

N.B.: This Menu class is completely different from the Menu class in version 2.5. No attempt has been made to be compatible, as this class and its subclasses are more powerful and easier to use than the original class.

Menu is a subclass of Control that contains a scene of other controls, called its body. When a menu is opened, it inserts the body into the world with a drop shadow. The appearance of the menu itself is defined by its interactor component. For example, a menu in a menu bar might appear as ‘‘File’’ and insert a pulldown menu with commands such as ‘‘Save’’ when opened.

MenuItem is a subclass of Control for defining the leaves of a menu hierarchy. Normally, an application will derive a command class from MenuItem and implement the virtual function Do as appropriate. One technique is to pass the Command an application object and pointer to member function; the Do function calls the member function on the application object.

PopupMenu, PulldownMenu, and PullrightMenu are subclasses of Menu that implement a common styles of menus. MenuBar is a subclass of HBox that manages the control state associated with a set of menus. PopupMenu has no appearance; it is opened explicitly in response to an input event. PulldownMenu and PullrightMenu open the menu body below and to the right of the menu, respectively. These classes make it possible to use menus in an application without creating or passing control states explicitly.

PUBLIC OPERATIONS

Menu::Menu(Interactor*)

Construct a new menu.

Menu::~Menu()

The destructor deletes the menu body in addition to its appearance component.

virtual void Menu::Include(Control*)

Add an item to the menu. If no scene is specified, Menu will create a vbox and insert items into it.

virtual void Menu::Popup(Event&)

Insert the body into the world centered around the coordinates associated with the event and activate the controls in the body. The menu is removed from the world when the controls are deactivated (normally when a button is released).

void Menu::SetBody(Interactor*)
Interactor* Menu::GetBody()
void Menu::SetAlign(Alignment)
Alignment Menu::GetAlign()
void Menu::SetDepth(int)
int GetDepth()
void Menu::SetBodyState(ControlState*)
ControlState* Menu::GetBodyState()
void Menu::SetScene(Scene*)
Scene* Menu::GetScene()

Set or get attributes of the menu. The depth of a menu is the number of pixels separating the body and the drop shadow.

Coord Menu::InsertX()
Coord Menu::InsertY()

Return the coordinates where the menu's body was last inserted into the world.

MenuItem::MenuItem(Interactor*)
MenuItem::MenuItem(const char* str, Alignment = Left)

Construct a new menu item. The second constructor defines the appearance of the item to be a message containing the text in *str*.

PulldownMenu::PulldownMenu(Interactor*)
PulldownMenu::PulldownMenu(const char* str)
PullrightMenu::PullrightMenu(Interactor*)
PullrightMenu::PullrightMenu(const char* str)

Construct a new pulldown or pullright menu. The second constructor defines the appearance of the menu to be a message containing the text in *str*.

PopupMenu::PopupMenu()

Construct a new popup menu.

MenuBar::MenuBar()

Construct a new menu bar.

virtual void MenuBar::Include(Control*)

Add a control to a menu bar. In addition to inserting the control into the bar's hbox, this operation attaches the control to the bar's control state.

SEE ALSO

Control(3I), Event(3I)

NAME

Message − interactor containing text

SYNOPSIS

#include <InterViews/message.h>

DESCRIPTION

A message is an interactor that contains a line of text. The alignment of the text with respect to the interactor's canvas can be specified in the constructor.

PUBLIC OPERATIONS

Message(const char*, Alignment = Center, int pad = 0, int hstretch = 0, int vstretch = 0)

Construct a new message interactor with the given text string and alignment. The text will be surrounded on all sides by whitespace of width *pad*, and the horizontal and vertical stretchability of the message will be set to the indicated values.

SEE ALSO

Interactor(3I)

NAME

ObjectSpace − distributed objects

SYNOPSIS

#include <InterViews/space.h>

DESCRIPTION

ObjectSpace is a class that represents a tightly-coupled cluster of objects. It is analogous to an address space, with objects instead of bytes or words. The ObjectSpace class allows remote access to objects through Unix IPC.

An object space is a kind of server process. It runs a top-level loop that receives messages from other object spaces (processes) and delivers them to the appropriate object.

Clients connect to object spaces through the name server process *spaceman*(1). Object names have two parts: a string that identifies the object space and is known to spaceman, and a string that is resolved by the object space itself.

PUBLIC OPERATIONS

ObjectSpace(const char*)

Create a space and register it under the given name with the space manager.

void StartServer(Connection* local, Connection* remote)

Start listening to the given connections for object messages. This routine is not necessary in general because as the constructor will connect to the space manager and establish connections on which to listen. However, some applications, such as the space manager itself, must specify the connections explicitly.

void UsePath(const char* path)

Tell the space manager to prepend the given path when searching for other known spaces.

Connection* Find(const char* name, boolean wait = false)

Ask the space manager to look for an object space with the given name. If *wait* is true, then wait until a space is created with the given name if none currently exists. Find returns a connection to the named object space or *nil* is none is defined an *wait* was false.

void Dispatch()

Wait for a single message from a client and deliver it to the target object.

void Deliver(Connection*, ObjectTag, int op, void*, int len)

Send an untyped message of a given length to a target object by mapping the tag and calls Message on the target object stub.

virtual void Message(Connection*, ObjectTag, int op, void*, int len)

Handle a message to the object space itself. Messages with a zero tag in the target field are handled by the object space. These are used to find an object by name with a space, create a new instance of an existing object, or destroy an object. If an object is found or created, the tag specified by the sender should be associated with the object for use in subsequent messages.

ObjectStub* Map(Connection*, ObjectTag)

Translate a client's tag to a stub. The client is identified by the connection.

void AddChannel(int, ObjectStub*)
void RemoveChannel(int)

Add or remove an additional channel on which to receive data. This channel is treated as a stream instead of interpreting packets with tags identifying target objects. When data is pending on the channel, the object space calls ChannelReady on the stub.

void Attach(int)
void Detach(int)

Start or stop listening to a channel added by an earlier call to AddChannel.

SEE ALSO

spaceman(1), Connection(3I), ObjectStub(3I)

Linton, M.A., ‘‘Distributed Management of a Software Database’’, *IEEE Software*, Vol. 4, No. 6, November 1987, pp. 70-76.

NAME

ObjectStub − unpack remote messages

SYNOPSIS

#include <InterViews/stub.h>

DESCRIPTION

An object stub interprets byte stream messages by unpacking parameters and calling the appropriate operation on another object. Because object stubs are accessed by remote object spaces, reference counts are used to determine when they can be deallocated.

PUBLIC OPERATIONS

ObjectStub()
˜ObjectStub()

When an object stub is created, its reference count is set to one. When the stub is deleted, its reference count is decremented and it is only destroyed if the count reaches zero.

void Reference()

Increment the reference count.

boolean LastRef()

Return true if the reference count is currently one.

virtual void Message(Connection*, ObjectTag, int op, void*, int len)

Handle a message for the object that the stub represents. The message contains a connection that specifies the sender, a tag that is usually the stub itself, an opcode that is the method selector, and a varying-length array of bytes. The default Message operation does nothing.

virtual void ChannelReady(int channel)

Handle data that is pending on the given channel. The channel is uninterpreted by the object space, so the stub must read and interpret the data.

virtual ObjectStub* Clone()

Create a copy of the object represented by the stub. Remote clients create a new instance by first finding a prototype by name and then copying the object using Clone.

SEE ALSO

ObjectSpace(3I)

NAME

Painter − graphics output

SYNOPSIS

#include <InterViews/painter.h>

DESCRIPTION

Painter is a class that provides ‘‘immediate-mode’’ graphics operations for drawing on a canvas. The state of a painter defines the graphics context for the drawing operations and includes a brush, foreground and background colors, a fill pattern and mode, a text font, a text style, an output origin and current position, and a transformation matrix.

STATE OPERATIONS

Painter(Painter* = stdpaint)

Create a new painter and copy its state from the given painter.

void SetBrush(Brush*)
Brush* GetBrush()

Set or return the painter's brush. Default is the predefined brush ‘‘single’’.

void SetColors(Color* fg, Color* bg)
Color* GetFgColor()
Color* GetBgColor()

Set or return the painter's colors. If either argument to SetColors is nil, then the corresponding color is not changed. Defaults are ‘‘black’’ for foreground and ‘‘white’’ for background.

void SetFont(Font*)
Font* GetFont()

Set or return the painter's text font. Default is the predefined font ‘‘stdfont’’.

void SetStyle(int style)
int GetStyle()

Set or get the painter's text style. A text style is a bit vector that can be assembled from the predefined constants *Plain*, *Boldface*, *Underlined*, and *Reversed*. Default is Plain.

void SetPattern(Pattern*)
Pattern* GetPattern()
void FillBg(boolean mode)
boolean BgFilled()

Set or return the painter's fill pattern and mode. If the mode is *true*, fill operations will set pixels corresponding to ones in the current fill pattern to the foreground color and pixels corresponding to zeros to the background color. If *false*, then only foreground pixels will be set. Default pattern is ‘‘solid’’; default mode is *true*.

void SetOrigin(int x0, int y0)
void GetOrigin(int& x0, int& y0)

Set or return the origin by which all coordinates are offset. Default is (0, 0).

void Translate(float dx, float dy)
void Rotate(float angle)
void Scale(float x, float y)
void SetTransformer(Transformer*)
Transformer* GetTransformer()

Coordinates passed to drawing operations are transformed according to the current origin, translation (cumulative), rotation, and scale factor. Internally, a transformation matrix is stored that can be directly set and accessed using SetTransformer and GetTransformer. The default transformer is nil, meaning no transformations are performed.

void SetPlaneMask(int mask)

Set which bit planes are affected by drawing operations. If the Kth bit of *mask* is set, then display operations will draw on plane K.

void SetOverwrite(boolean)

Set whether a painter is allowed to write in subcanvases. If true, drawing operations will be able to write over the canvases of component interactors. If false, drawing operations will be clipped by any subcanvases. The default is false.

void Clip(Canvas*, Coord x1, Coord y1, Coord x2, Coord y2)
void NoClip()

Clip restricts output operations to the specified region of the canvas. NoClip removes the restriction so that operations affect the entire canvas. Only one clipping region may be in effect at a time.

void MoveTo(Coord x, Coord y)

Set the current output position. The output position is used and updated by Text and CurveTo.

DRAWING OPERATIONS

void Curve(Canvas*, Coord x0, y0, x1, y1, x2, y2, x3, y3)
void CurveTo(Canvas*, Coord x1, y1, x2, y2, x3, y3)

Paint a Bezier curve on the canvas from the first point to the last point (but not going through the intermediate control points). The curve will lie within the polygon formed by the four points. CurveTo uses the current position for the first point.

void BSpline(Canvas*, Coord x[], y[], int n)
void ClosedBSpline(Canvas*, Coord x[], y[], int n)
void FillBSpline(Canvas*, Coord x[], y[], int n)

Draw the B-spline defined by the *n* control vertices. If closed or filled, the last point is connected to the first point.

void Circle(Canvas*, Coord x, y, int r)
void FillCircle(Canvas*, Coord x, y, int radius)

Draw a circle with center (*x*, *y*) and radius *r*.

void Ellipse(Canvas*, Coord x, y, int xr, int yr)
void FillEllipse(Canvas*, Coord x, y, int xr, int yr)

Draw an ellipse with center (*x*, *y*), horizontal radius *xr*, and vertical radius *yr*.

void Line(Canvas*, Coord x1, y1, x2, y2)
void MultiLine(Canvas*, Coord x[], y[], int n)
void Polygon(Canvas*, Coord x[], y[], int n)
void FillPolygon(Canvas*, Coord x[], y[], int n)

Draw a path using the current brush and colors. The Line operation draws a vector between two points (inclusive); MultiLine draws a number of connected vectors; Polygon draws a closed set of vectors; FillPolygon fills the area inside a polygon using the current fill pattern and colors.

void Point(Canvas*, Coord x, y)
void MultiPoint(Canvas*, Coord x[], y[], int n)

Set a point or set of points to the current foreground color.

void Rect(Canvas*, Coord x1, y1, x2, y2)
void FillRect(Canvas*, Coord x1, y1, x2, y2)
void ClearRect(Canvas*, Coord x1, y1, x2, y2)

Draw a rectangle with opposite corners specfied by (*x1*, *y1*) and (*x2*, *y2*). FillRect fills the rectangle using the current pattern and colors; ClearRect fills the rectangle with the background color.

void Text(Canvas*, const char* str, Coord x, Coord y)
void Text(Canvas*, const char* str, int n, Coord x, Coord y)
void Text(Canvas*, const char* str)
void Text(Canvas*, const char* str, int n)

Draw a string or substring of text using the current Font and text style. The (*x*, *y*) coordinates specify the lower-left corner of the bounding box of the text. The width of the bounding box is the width of the string as reported by the Font::Width operation, and the height of the bounding box is the Font height. Most fonts will result in output which only affects pixels within the bounding box. The current transformation matrix is applied to both the positions and the shapes of characters drawn. If the matrix specifies a transformation involving rotation or scaling, the resulting operation may proceed much more slowly than normal. If background fill mode is on, then the characters are drawn in the foreground color, and other pixels within the bounding box are set to the background color. If background fill mode is off, only the foreground pixels are set. If no coordinates are specified, then the current position (defined by MoveTo) is used and updated to reflect the lower-right corner of the bounding box.

void Stencil(Canvas*, Coord x, Coord y, Bitmap* image, Bitmap* mask = nil)

Paint foreground and background colors through a stencil formed by positioning the image and mask Bitmaps with their origins at the point (*x*, *y*). Foreground color is painted where the *image* Bitmap has a true value and background color where image is false. However, only pixels corresponding to a true value in the mask Bitmap are affected. A nil *mask* is equivalent to a mask of the same size and shape as image and containing all true values. The current transformation matrix is applied to both the image and mask Bitmaps. If the matrix specifies a transformation involving rotation or scaling, the resulting operation may proceed much more slowly than normal.

void RasterRect(Canvas*, Coord x, Coord y, Raster*)

Render the Raster with its lower-left corner at the position (*x*, *y*). The current transformation matrix is applied to the Raster. If the matrix specifies a transformation involving rotation or scaling, the resulting operation may proceed much more slowly than normal.

void Read(Canvas*, void*, Coord x1, y1, x2, y2)
void Write(Canvas*, const void*, Coord x1, y1, x2, y2)
void Copy(Canvas* src, Coord x1, y1, x2, y2, Canvas* dst, Coord x0, y0)

Read copies a region of a canvas into memory. Write copies data from memory to a region of a canvas. Copy reads a region of one canvas and writes the data into a region of another canvas (or within a canvas if *src* and *dst* are the same). The point (*x0*, *y0*) is the lower-left corner of the destination region. Note that *Read* and *Write* are superceded by operations that use Rasters.

SEE ALSO

Bitmap(3I), Brush(3I), Canvas(3I), Color(3I), Font(3I), Pattern(3I), Raster(3I), Transformer(3I)

NAME

Panner − two-dimensional scrolling and zooming

SYNOPSIS

#include <InterViews/panner.h>

DESCRIPTION

A panner is an interactor that manipulates the perspective of another interactor to provide two-dimensional scrolling and zooming. A panner is composed of a *slider* and several adjusters, including two zoomers and four movers. The slider's position and size reflect the current origin, size, and aspect ratio in the interactor's perspective. Dragging the slider around with the mouse changes the perspective's *curx* and *cury* members.

PUBLIC OPERATIONS

Panner(Interactor∗, int size = 0)

Create a panner for the given interactor, optionally with the given width. If the size is 0, a reasonable default size is used. The panner's aspect ratio reflects that of the perspective.

Slider(Interactor∗)
Slider::Handle(Event&)
Slider::Reshape(Shape&)
Slider::Update()

A slider reflects and lets the user specify the visible portion of the view. A rectangle represents the visible area, while the area surrounding it represents the total area. Clicking and holding down any mouse button on the rectangle allows you to drag it to a different position within the total area. The perspective's current origin is shifted to reflect the new position when the button is released. Holding down the shift key while dragging the rectangle constrains its motion horizontally or vertically. Clicking outside the rectangle moves it in the corresponding one of eight directions; for example, clicking to the right of the rectangle moves it to the right. If the left mouse button is used, the movement increment equals the perspective's *sx* or *sy* member variables. The middle mouse button increments by *lx* or *ly*, and the right mouse button centers the rectangle around the cursor. The Reshape call sets the width of the slider to that of the passed Shape and calls Change on the parent interactor. Update redisplays the slider to reflect a change in the interactor's perspective.

X DEFAULTS

The ‘‘syncScroll’’ resource specifies whether the slider should work in real-time or not. When synchronized scrolling is employed the interactor scrolls synchronously as the slider is dragged; otherwise the slider's outline follows the mouse during dragging, and the interactor is scrolled only after the user stops dragging the slider. Sliders will scroll synchronously if their syncScroll resource is set to ‘‘on’’ or ‘‘true.’’

SEE ALSO

Adjuster(3I), Interactor(3I), Perspective(3I)

NAME

Pattern − fill patterns for a painter

SYNOPSIS

#include <InterViews/pattern.h>

DESCRIPTION

A pattern defines how to fill areas during graphics operations. The global variables ‘‘solid’’ and ‘‘clear’’ point to patterns for filling all or none of the pixels in a particular area; ‘‘lightgray’’, ‘‘gray’’, and ‘‘darkgray’’ point to patterns for filling one-eighth, one-half, or seven-eighths of the pixels in an area respectively.

PUBLIC OPERATIONS

Pattern(int b[patternHeight])

Construct a pattern from the array *b*. For each element of *b*, the least significant *patternWidth* bits are used. If a bit is one, then the foreground color will be used when filling. Otherwise, the background color will be used. The first pattern element defines the topmost row of pixels, and the least significant bit of each element defines the rightmost column.

Pattern(int i)

Construct a pattern by interpreting the least signficant 16 bits of *i* as a 4x4 bitmap and replicating it to make a full *patternWidth* by *patternHeight* array.

Pattern(Bitmap*)

Construct a pattern from the bitmap. If necessary, the bitmap will be tiled or truncated to satisfy window system requirements for fill pattern sizes.

SEE ALSO

Painter(3I)

NAME

Perspective − visible portion of an interactor

SYNOPSIS

#include <InterViews/perspective.h>

DESCRIPTION

A perspective object describes which part of an interactor's total display area is visible in the interactor's canvas. The main use of perspectives is for scrolling, panning, and zooming interactors. The member variables (*x0*, *y0*) define the origin of the interactor's display area; *width* and *height* define the total height of the area. The member variables (*curx*, *cury*) define the lower-left corner of the interactor's canvas relative to the total area; *curwidth* and *curheight* define how much of the total area is displayed in the canvas.

The member variables *sx*, *sy*, *lx*, and *ly* specify small and large increments in each dimension. These increments are used by a scroller to implement line and page scrolling.

PUBLIC OPERATIONS

Perspective()

Define a new perspective with all member variables initialized to zero.

void Attach(Interactor*)

Add to the list of interactors that are interested in updates to the perspective. These interactors reflect the perspective visually; examples include scrollers and sliders.

void Detach(Interactor*)

Remove an interactor from the list of those interested in updates. This operation should be called from the interactor's destructor.

void Init(Coord x0, Coord y0, Coord width, Coord height)

Initialize the perspective display area.

void Update()

Notify all the interactors interested in the perspective that it has changed. The perspective's interactor should call Update after modifying its perspective.

SEE ALSO

Adjuster(3I), Interactor(3I), Panner(3I), Scroller(3I)

NAME

PropertySheet − user-customizable attributes

SYNOPSIS

#include <InterViews/propsheet.h>

DESCRIPTION

A property sheet is a tree of directories and attributes. The syntax and semantics of property sheets is the same as the X Toolkit defines for resources, though the terminology here is slightly different.

An attribute is defined by a string name, string value, and optional string type descriptor. A property directory contains a list of attributes, a list of subdirectories, a list of ''default'' attributes, and a list of ''default'' subdirectories. Any of the lists can be empty. InterViews uses a global property sheet called ''properties'' to customize the appearance and behavior of interactive objects.

PUBLIC OPERATIONS

PropertySheet()

Construct a property sheet. The initial sheet contains a single root directory with no attributes.

˜PropertySheet()

Free all storage associated with the property sheet.

boolean Load(const char∗ filename)

Read a set of attributes from the given file. Each attribute is defined on a separate line, with its path from the root fully-qualified. A ''.'' separates directories, a ''∗'' separates a default, and a '':'' separate the path from the value. Thus, ''A.B∗C.D:this is a test'' defines an attribute D with value ''this is a test''. The path to D starts at the property sheet root, followed by a subdirectory named A, a subdirectory named B, and a default subdirectory named C.

If any of the directories are not present, Load will create them. If an attribute is already defined, Load will overwrite its value with the new value.

void Put(const char∗ path, const char∗ value, const char∗ type = nil)

Add an attribute to the property sheet explicitly.

boolean Get(PropertyDef&)

Find an attribute. PropertyDef is a structure containing a name, type descriptor, and value. To make lookup fast in the cases where the name can be processed ahead of time, the name is an index into a StringTable(3I). The search looks in each directory on the property directory stack in order. Get searches the top directory for an attribute and all other directories for a default attribute.

boolean GetLocal(PropDir∗, PropertyDef&)

Find an attribute that is local to a particular property directory.

PropDir∗ Find(PropertyName)

Find the property directory that matches the given directory name. The search looks in each directory on the property directory stack in order. If no directory can be found with the given name, Find returns *nil*.

void Push(PropDir∗, boolean sibling)

Add a property directory to the stack. If sibling is true, then Get will search both the top and next to top stack elements for an attribute.

void Pop()

Remove the top of the property directory stack.

PropDir∗ Root()

Return the root directory of the property sheet.

SEE ALSO

Interactor(3I), StringTable(3I)

NAME

Raster − simple rasterized image data

SYNOPSIS

#include <InterViews/raster.h>

DESCRIPTION

A *Raster* is a two−dimensional array of Colors that can be rendered by a Painter. Rasters can be constructed from data supplied by the application program or from a region of a Canvas. The only operations currently defined on rasters are for accessing individual raster cells.

PUBLIC OPERATIONS

Raster(Color∗∗ data, int width, int height)

Create a raster from the data specified. The data is formatted as a two−dimensional matrix of pointers to Colors, organized in row−major order. The data is transferred into an internal format, so subsequent changes in either the passed data or the raster will not affect the other. Passing nil to *data* is equivalent to passing an array with all cells containing a pointer to the predefined Color *white*.

Raster(Canvas∗, Coord x0, Coord y0, int width, int height)

Create a raster from the current contents of the Canvas. The coordinates *x0* and *y0* specify the lower left corner of the region of the Canvas to use.

Raster(Raster∗)

Create a copy of the given raster.

int Width()
int Height()

These operations return the size of the raster.

Color∗ Peek(int x, int y)
void Poke(Color∗, int x, int y)
boolean Contains(int x, int y)

Operations for reading and writing individual cells in the raster. If the specified coordinates are not within the raster bounds, *Contains* returns false, *Peek* returns nil, and *Poke* has no effect.

SEE ALSO

Painter(3I), Transformer(3I)

NAME

Regexp – regular expression searching

SYNOPSIS

#include <InterViews/regexp.h>

DESCRIPTION

A Regexp encapsulates a regular expression pattern and defines operations for searching and matching the pattern against a string. The syntax of the regular expression pattern is the same as that for ed(1). Information can be obtained about the most recent match of the regular expression (and its sub-expressions).

PUBLIC OPERATIONS

Regexp(const char* pattern)
Regexp(const char* pattern, int length)

Construct a new Regexp for *pattern*.

int Match(const char* text, int length, int index)

Attempt a match against *text* (of length *length*) at position *index*. The return value is the length of the matching string, or a negative number if the match failed.

int Search(const char* text, int length, int index, int range)

Search for a match in the string *text* (of length *length*). Matches are attempted starting at positions between *index* and *index* plus *range*. If *range* is positive the first match after *index* is reported. If *range* is negative the first match before *index* is reported. The return value is the index of the starting position of the match, or a negative number if there is no match in the specified range.

int BeginningOfMatch(int subexp)
int EndOfMatch(int subexp)

Return information about the most recent match. If *subexp* is zero (the default), information is reported for the complete regular expression. Other values of *subexp* refer to sub-expressions in the pattern. For example, if *subexp* is 2, information is returned for the sub-expression specified by the second pair of \(and \) delimiters in the pattern.

SEE ALSO

ed(1)

NAME

ReqErr − handle request errors

SYNOPSIS

#include <InterViews/reqerr.h>

DESCRIPTION

Some requests to the window server may generate an error. For example, if a window is destroyed while it is being repositioned interactively, the request to place the window in the new position will fail. Normally, errors cause a process to terminate. The ReqErr class is used to handle errors within an application. The base class defines the attributes associated with each error, including a *message* string describing the error, an integer *request* code that specifies the kind of call that generated the error, and a *id* object that was the target of the request. When an error occurs, the attributes of the current request handler are assigned and the handler's Error operation is called.

PUBLIC OPERATIONS

ReqErr* Install()

Use this error handler for subsequent errors and return the previous error handler.

virtual void Error()

Handle a detected error. The default handler is a nop.

SEE ALSO

Interactor(3I)

NAME

Resource − InterViews shared object

SYNOPSIS

#include <InterViews/resource.h>

DESCRIPTION

Resource is a class that provides a simple form of reference counting for shared objects. Every resource has a reference count that is initially set to zero and must be explicitly incremented through a call to Reference. A resource should not be deleted like other objects; instead, the global function Unref should be used to unreference the object. When a resource is unreferenced, the reference count is decremented and if it becomes zero the object is destroyed.

PUBLIC OPERATIONS

Resource()

Create a new resource object with its reference count set to zero.

void Reference()

Increment the resource's reference count.

FRIEND OPERATIONS

void Unref(Resource*)

If the resource pointer is non-nil, decrement its reference count and destroy the object if the reference count is now zero. Unref is a global function, not a Resource member function, because it must check if the target pointer is nil.

NAME

Rubband − rubberbanding graphical objects

SYNOPSIS

#include <InterViews/rubband.h>
#include <InterViews/rubcurve.h>
#include <InterViews/rubgroup.h>
#include <InterViews/rubline.h>
#include <InterViews/rubrect.h>
#include <InterViews/rubverts.h>

DESCRIPTION

Rubband is the base class for a set of classes that provide graphical feedback to a user during an interactive operation such as sizing a rectangle. A rubberband varies in appearance as its *tracking point* changes. A rubberband is *not* an interactor itself; interactors use rubberbands to provide animated effects. Because rubberbands are not interactors, rubberband constructors must be passed the painter and canvas to use for drawing.

PUBLIC OPERATIONS

RubberLine(Painter*, Canvas*, Coord fx, fy, mx, my, offx = 0, offy = 0)
A rubber line stretches to follow the tracking point while its other endpoint stays tied down.

RubberAxis(Painter*, Canvas*, Coord fx, fy, mx, my, offx = 0, offy = 0)
A rubber axis stretches either horizontally or vertically to follow the tracking point while its other endpoint stays tied down. One endpoint stays fixed while the other moves along either the horizontal or vertical axis, whichever is closer to the tracking point.

SlidingLine(Painter*, Canvas*, Coord x0, y0, x1, y1, rfx, rfy, offx = 0, offy = 0)
A sliding line slides around without changing its length or angle as the tracking point moves around a reference point.

RotatingLine(Painter*, Canvas*, Coord x0, y0, x1, y1, rfx, rfy, offx = 0, offy = 0)
A rotating line rotates around (*cx*, *cy*) based on the angle between the radius from (*cx*, *cy*) to (*rfx*, *rfy*) and the radius from (*cx*, *cy*) to the tracking point.

RubberRect(Painter*, Canvas*, Coord fx, fy, mx, my, offx = 0, offy = 0)
A rubber rectangle stretches to follow the tacking point while its opposite corner stays tied down.

SlidingRect(Painter*, Canvas*, Coord x1, y1, x2, y2, rfx, rfy, offx = 0, offy = 0)
A sliding rectangle slides around the screen without changing its side or orientation as the tracking point moves around a reference point.

ScalingRect(Painter*, Canvas*, Coord x1, y1, x2, y2, cx, cy, offx = 0, offy = 0)
A scaling rectangle scales equally in both dimensions about (*cx*, *cy*) to keep one edge intersecting the tracking point.

RotatingRect(Painter*, Canvas*, Coord x1, y1, x2, y2, cx, cy, rfx, rfy, offx = 0, offy = 0)
A rotating rectangle rotates about (*cx*, *cy*) based on the radius from (*cx*, *cy*) to (*rfx*, *rfy*) and the radius from (*cx*, *cy*) to the tracking point.

StretchingRect(Painter*, Canvas*, Coord x1, y1, x2, y2, Side ts, Coord offx = 0, offy = 0)
A stretching rectangle's tracking side follows the tracking point while the opposite side stays fixed. The *side* can be one of *LeftSide*, *RightSide*, *BottomSide*, or *TopSide*.

RubberCircle(Painter*, Canvas*, Coord cx, cy, rx, ry, offx = 0, offy = 0)
A rubber circle's radius varies with the magnitude of the tracking point's distance from (*cx*, *cy*).

RubberEllipse(Painter*, Canvas*, Coord cx, cy, rx, ry, offx = 0, offy = 0)
A rubber ellipse's horizontal and vertical radii vary with the magnitude of the tracking point's distance from (*cx*, *cy*).

RubberSpline(Painter*, Canvas*, Coord x[], y[], int n, pt, Coord offx = 0, offy = 0)

RubberClosedSpline(Painter∗, Canvas∗, Coord x[], y[], int n, pt, Coord offx = 0, offy = 0)
A rubber spline is the section of a B-spline defined by *n* control points that changes shape as the control point indexed by *pt* is set to the tracking point.

RubberHandles(Painter∗, Canvas∗, Coord x[], y[], int n, pt, size, Coord offx = 0, offy = 0)
A rubber handles object is a set of filled squares centered about each point in the list. The point index by *pt* follows the tracking point.

SlidingPointList(Painter∗, Canvas∗, Coord x[], y[], int n, pt, Coord rfx, rfy, offx = 0, offy = 0)
A sliding point list slides around the screen without changing its appearance as the tracking point moves around a reference point.

SlidingLineList(Painter∗, Canvas∗, Coord x[], y[], int n, pt, Coord rfx, rfy, offx = 0, offy = 0)
A sliding line list slides around the screen without changing the appearance of its connected lines as the tracking point moves around (*rfx*, *rfy*).

ScalingLineList(Painer∗, Canvas∗, Coord x[], Coord y[], int n, Coord cx, cy, rfx, rfy, offx = 0, offy = 0)
A scaling line list scales a list of lines about (*cx, cy*) by an amount based on the tracking point's distance from (*rfx, rfy*).

RotatingLineList(Painer∗, Canvas∗, Coord x[], Coord y[], int n, Coord cx, cy, rfx, rfy, offx = 0, offy = 0)
A rotating line list rotates a list of lines about (*cx, cy*) based on the angle between the radius from (*cx*, *cy*) to (*rfx*, *rfy*) and the radius from (*cx*, *cy*) to the tracking point.

GrowingVertices(Painter∗, Canvas∗, Coord x[], y[], int n, Coord offx = 0, offy = 0)
GrowingMultiLine(Painter∗, Canvas∗, Coord x[], y[], int n, Coord offx = 0, offy = 0)
GrowingPolygon(Painter∗, Canvas∗, Coord x[], y[], int n, Coord offx = 0, offy = 0)
GrowingBSpline(Painter∗, Canvas∗, Coord x[], y[], int n, Coord offx = 0, offy = 0)
GrowingClosedBSpline(Painter∗, Canvas∗, Coord x[], y[], int n, Coord offx = 0, offy = 0)
virtual void AppendVertex(Coord, Coord)
GrowingVertices is an abstract rubberband that maintains a list of vertices and allows the list to grow dynamically. The tracking point corresponds to the last vertex in the list. The AppendVertex operation adds a new vertex to the list, making it the new tracking point. GrowingMultiLine, GrowingPolygon, GrowingBSpline, and GrowingClosedBSpline are GrowingVertices subclasses that draw a multiline, polygon, B-spline, and closed B-spline, respectively.

RubberGroup(Painter∗, Canvas∗)
void Append(Rubberband∗, Rubberband∗ = nil, Rubberband∗ = nil, Rubberband∗ = nil)
void Remove(Rubberband∗)
void RemoveCur()
void SetCurrent(Rubberband∗)
Rubberband∗ GetCurrent()
Rubberband∗ First()
Rubberband∗ Last()
Rubberband∗ Next()
Rubberband∗ Prev()
boolean IsEmpty()
boolean AtEnd()
A RubberGroup composes other rubberbands, providing operations for including, excluding, and traversing its children. RubberGroups are useful for combining the behavior of several rubberbands into one, eliminating the need to call Track on multiple rubberbands. The RubberGroup's SetCanvas and SetPainter operations set the children's canvas and painter as well as the RubberGroup's, but the children do not otherwise inherit the group's canvas and painter. The Track operation calls Track on each child with the same parameters.

virtual void Draw()

void Redraw()
virtual void Erase()

Explicitly draw, redraw, and erase the rubberband. Multiple calls to Draw without an intervening Erase will only draw the rubberband once. Redraw forces the rubberband to be drawn. Since rubberbands are drawn in XOR mode, Redraw will erase the rubberband every other time it is called. These functions are used only on occasion, for example, when part of the screen is erased while a rubberband is visible. The Track function is normally used to draw the rubberband in response to mouse input.

virtual void Track(Coord x, Coord y)

Erase the rubberband and redraw it using the new tracking point.

void GetOriginal(Coord& x1, Coord& y2, Coord& x2, Coord& y2)
void GetCurrent(Coord& x1, Coord& y2, Coord& x2, Coord& y2)

GetOriginal obtains the original points specified for the rubberband. GetCurrent obtains the current points based on the most recent call to Track. The meaning of the points depends on the kind of rubberband. For lines, the points are the endpoints; for rectangles, the points are opposite corners; for curves, (*x1*, *y1*) is the center and *x2* and *y2* are the radii.

virtual void SetPainter(Painter*)
Painter* GetPainter()

Explicitly set and get the painter used by the rubberband. By default, rubberbands use a copy of the painter passed to their constructor. The user can specify a particular painter to use by passing a nil painter to the constructor and later calling SetPainter. *N.B.:* the rubberband may alter the painter specified via SetPainter.

virtual void SetCanvas(Canvas*)
Canvas* GetCanvas()

Explicitly set and get the canvas used by the rubberband.

SEE ALSO

Canvas(3I), Painter(3I)

NAME

Scene, MonoScene − composite interactors

SYNOPSIS

#include <InterViews/scene.h>

DESCRIPTION

Scene is the base class for all interactors that are defined in terms of one or more instances of other interactors. **MonoScene** is a subclass for scenes that contain a single instance. A monoscene's shape is copied from its component and the component's canvas is given the same size and location as the monoscene's canvas.

When a scene is deleted, it should delete all its component interactors. A monoscene deletes its existing component if a new component is inserted.

PUBLIC OPERATIONS

void Insert(Interactor*)
void Insert(Interactor*, Coord x, Coord y, Alignment = BottomLeft)
virtual Interactor* Wrap(Interactor*)
virtual void DoInsert(Interactor*, boolean, Coord& x, Coord& y, Shape*)

Add an interactor to the scene, optionally aligned with respect to a specific position relative to the lower-left corner of the scene. Valid alignment types are TopLeft, TopCenter, TopRight, CenterLeft, Center, CenterRight, BottomLeft, BottomCenter, BottomRight, Left, Right, Top, Bottom, HorizCenter, and VertCenter. The scene uses the interactor's desired shape, if it has non-zero width, to size the interactor. The base scene class always calls Wrap and DoInsert from Insert; thus subclasses only implement DoInsert (and Wrap if they want to wrap additional interactors around the inserted interactor) and users only call Insert. In the case of a monoscene, the current interior interactor, if not nil, is deleted and replaced by the given interactor. If an interactor is inserted into a scene that is already mapped, the interactor and its descendants will be configured before DoInsert is called.

void Propagate(boolean)
void Change(Interactor*)
virtual void DoChange(Interactor*, Shape*)

Change notifies a scene that a given interactor has changed its desired shape. If the propagate flag associated with the scene is true, Change will call DoChange and then Change(this) on the scene's parent. If the propagate flag is false, Resize will be called instead. The propagate flag is true by default and can be set with the Propagate operation.

void Move(Interactor*, Coord x, Coord y, Alignment = BottomLeft)
virtual void DoMove(Interactor*, Coord& x, Coord& y)

Request that an interactor be moved to the given coordinates using the given alignment. As with Insert and Change, users call Move and subclasses implement DoMove.

void Remove(Interactor*)
virtual void DoRemove(Interactor*)

Remove an interactor from a scene. Users call Remove and subclasses implement DoRemove.

void Lower(Interactor*)
virtual void DoLower(Interactor*)

Put the interactor's canvas below all others in the scene, potentially obscuring it. Users call Lower and subclasses implement DoLower.

void Raise(Interactor*)
virtual void DoRaise(Interactor*)

Put the interactor's canvas above all others in the scene, making it fully visible. Users call Raise and subclasses implement DoRaise.

PROTECTED OPERATIONS

Scene()

MonoScene()

Construct a new scene or monoscene. Scenes and monoscenes are abstract classes; thus, the constructors should only be called implicitly by subclass constructors.

void Place(Interactor*, Coord x1, Coord y1, Coord x2, Coord y2, boolean = true)

Assign an interactor's canvas to be a given region of the scene's canvas. Unless the optional sixth parameter is false, the interactor will be mapped. If the interactor is already assigned the given position and size, then this operation will have no effect. This optimization means that a scene can place all of its elements in response to a Change and be assured that only those elements that have really changed will be redrawn.

void Map(Interactor*, boolean = true)

Make the interactor visible on the display if and when the scene is visible. Unless the optional second parameter is false, the interactor will be raised to the top of the scene.

void Unmap(Interactor*)

Make the interactor not be visible on the display if and when the scene is visible.

void UserPlace(Interactor*, int w, int h)

Let the user interact with the window manager to specify the size and location of the interactor's canvas, whose default width and height must also be supplied by the program.

SEE ALSO

Canvas(3I), Interactor(3I), Shape(3I)

NAME

Scroller, HScroller, VScroller − scrolling an interactor

SYNOPSIS

#include <InterViews/scroller.h>

DESCRIPTION

A scroller is an interactor that manipulates the perspective of another interactor. It contains a bar whose position and size reflect the current origin and size in the interactor's perspective. Clicking and dragging in the scroller modifies the attached interactor's perspective. **VScroller** is a class for defining vertical scrollers, **HScroller** for horizontal scrollers.

PUBLIC OPERATIONS

HScroller(Interactor*, int size = 0)
VScroller(Interactor*, int size = 0)

Create a scroller of a given size (height for a horizontal scroller, width for a vertical scroller). A zero size means to use the default size.

virtual void Update()

Redisplay the scroller to reflect a change in the interactor's perspective.

X DEFAULTS

The ‘‘syncScroll’’ resource specifies whether the scroller should work in real-time or not. When synchronized scrolling is employed the interactor scrolls synchronously as the scroller's bar is dragged; otherwise the bar's outline follows the mouse during dragging, and the interactor is scrolled only after the user stops dragging the bar. Scrollers will scroll synchronously if their syncScroll resource is set to ‘‘on’’ or ‘‘true.’’ The state of the syncScroll attribute can be toggled temporarily by holding down the ‘‘control’’ key while the scroller is manipulated.

SEE ALSO

Interactor(3I), Perspective(3I)

NAME

Sensor − input interest for interactors

SYNOPSIS

#include <InterViews/sensor.h>

DESCRIPTION

A sensor specifies a set of interesting input events. Several sensors are predefined as global variables. The sensor ‘‘allEvents’’ is interested in all user input events (excludes ChannelEvent and TimerEvent); ‘‘noEvents’’ is not interested in any events; ‘‘updownEvents’’ is interested in UpEvents and DownEvents; and ‘‘onoffEvents’’ is interested in EnterEvents and LeaveEvents.

PUBLIC OPERATIONS

Sensor()

Create a new sensor that is not interested in any events.

Sensor(Sensor*)

Create a sensor whose interests are the same as the given sensor.

void Catch(EventType)
void CatchButton(EventType, int)
void CatchChannel(int)
void CatchTimer(int sec, int usec)

Express interest in an event. CatchButton can be used for keys as well as buttons, though it is necessary to know the key code. Not all types of events make sense for all the operations; for example, Catch(ChannelEvent) is not meaningful. Such requests are silently ignored.

void Ignore(EventType)
void IgnoreButton(EventType, int)
void IgnoreChannel(int)

Remove interest in an event. Ignore means no interest in any events of a particular kind; thus Ignore(DownEvent) means pressing any button will not generate an event.

void CatchRemote()
void IgnoreRemote()

Express or remove interest in access to the structure of an interactor's canvas. Only a world view should use these operations, as such access is reported using the WorldView::InsertRemote and WorldView::ChangeRemote operations.

SEE ALSO

Interactor(3I), World(3I), WorldView(3I)

NAME

Shape − what shape a canvas should have

SYNOPSIS

#include <InterViews/shape.h>

DESCRIPTION

A shape specifies the desired characteristics of a canvas. An interactor should set the fields of its shape when it is reconfigured and should call Change on its parent if any of these fields change.

The dimensions of a shape are defined by a ‘‘natural’’ size, a stretch amount, and a shrink amount. The *width* and *height* fields indicate the desired sizes. The *hstretch*, *vstretch*, *hshrink*, *vshrink* fields define how flexible these desired sizes are. For example, an interactor may have a natural size of 100, but can adequately handle any size between 50 and 200. The stretchability for this case would be 100 and the shrinkability 50.

The constants *hfil* and *vfil* are provided to indicate ‘‘infinite’’ stretching or shrinking. They are represented as very large numbers and manipulated exactly the same as other stretch and shrink parameters.

The *aspect* field specifies the desired aspect ratio. A value of zero means any aspect is acceptable.

The *hunits* and *vunits* fields indicate that the canvas dimensions should be multiples of some values.

PUBLIC OPERATIONS

Shape()

Construct a new shape with undefined natural size and infinite stretchability and shrinkability in both directions.

boolean Defined()
boolean Undefined()

Test if the shape's natural size is defined or not.

void Rect(int width, int height)
void Square(int side)

Short-hand for setting the dimensions of the shape. Square sets both dimensions to *side* and sets the aspect ratio to one.

void Rigid(int hshrink, int hstretch, int vshrink, int vstretch)
void Rigid()

Set the stretchability and shrinkability fields of the shape. If no arguments are specified, the fields are set to zero.

void SetUndefined()

Reset the shape's natural size to be undefined.

SEE ALSO

Interactor(3I), Scene(3I)

NAME

StringBrowser − browse and/or select items in a list

SYNOPSIS

#include <InterViews/strbrowser.h>

DESCRIPTION

StringBrowser is an interactor that provides a browsing and selection interface to a list of strings. StringBrowsers are often used in dialog boxes to let the user browse and select from an unbounded set of textual entries. A StringBrowser displays the list of strings and provides several ways to scroll the list. The user selects strings with either the mouse or the keyboard; the StringBrowser can restrict selection to a single string or allow multiple selections. The application queries the StringBrowser for the selected string(s).

A StringBrowser contains no entries initially; strings are appended consecutively or in random order, and they can be removed in any order. The strings are accessed with a **string index** from 0 to *n*-1, where *n* is the total number of strings. The StringBrowser maintains a separate list of strings that have been selected; these strings are accessed with a **selection index** from 0 to *m*-1, where m is the total number of selected strings.

Left-clicking inside the StringBrowser (or calling the Browse function) makes it interpret subsequent keyboard events as scrolling or selection operations on the entries. StringBrowser relinquishes control when it detects a click outside its canvas or when it receives a predetermined escape character.

Middle-clicking inside the StringBrowser lets the user ‘‘grab-scroll’’ the entries. During grab-scrolling, the StringBrowser scrolls the entries to follow the mouse position, making it appear as though the user is dragging the entries themselves. Right-clicking engages ‘‘rate-scrolling,’’ a joy-stick-like scrolling interface in which the scrolling rate increases as the user drags the mouse away from the initial click point. For example, dragging the mouse downwards after the initial click scrolls the browser downwards at an increasing rate; dragging upwards thereafter reduces the rate until scrolling stops entirely at the initial click point. Dragging up beyond this point makes the browser scroll in the reverse direction.

KEY BINDINGS

Below are the key bindings for browsing and selection commands:

g Go to the first string.

G Go to the last string.

a Select all.

DEL or BS
Unselect all.

p Select previous string.

n Select next string.

< Select topmost-visible string.

> Select bottommost-visible string.

j Scroll down one string.

k Scroll up one string.

SPACE Scroll down one screenful.

b Scroll up one screenful.

d Scroll down one-half screenful.

u Scroll up one-half screenful.

PUBLIC OPERATIONS

StringBrowser(ButtonState*, int rows, int cols, boolean uniqueSel, int highlight, const char* done)
StringBrowser(const char* name, ButtonState*, int, int, boolean, int, const char*)

Create a new StringBrowser object. The StringBrowser will use the ButtonState to communicate the result of browsing operations. The StringBrowser bases its shape on the *rows* and *cols* parameters: the height equals *rows* multiplied by the font height, and the width equals *cols* multiplied by the width of an average character. The *uniqueSel* parameter specifies whether or not selection is limited to one string (the default), and the *highlight* parameter specifies the appearance of selected strings (Reversed by default). Typing any character in the string *done* will make the StringBrowser stop interpreting keyboard events; the ButtonState is set to this terminating character as a side-effect. A double left-click inside the StringBrowser is equivalent to a single left-click selection followed by a key press that generates the first character in *done*. To be interpreted as a double-click, two clicks must occur not more than *clickDelay* milliseconds apart. The default click delay is 250 ms; a different value can be specified via the "clickDelay" user-preference attribute.

void Browse()

Initiate browsing. Subsequent keypresses will be interpreted as browsing and selection commands according to the key bindings listed above.

void Insert(const char* string, int index)

Insert *string* at position *index*. StringBrowser makes a copy of the string for internal use.

void Append(const char* string)

Insert a copy of *string* after the last string.

void Remove(int index)

Remove the string at position *index*, moving subsequent strings up to fill the gap.

int Index(const char* string)

Return the index of the first string that matches *string*.

char* String(int index)

Return the string at the given index.

int Count()

Return the total number of strings in the StringBrowser.

void Clear()

Remove all strings from the StringBrowser.

void Select(int index)
void Unselect(int index)

Select (unselect) the string at the given index. The string will be redrawn in the appropriate style. If multiple selections are not allowed and an another string has been selected already, then selecting a new string will unselect the original string.

void SelectAll()
void UnselectAll()

Select (unselect) all strings. SelectAll does nothing if multiple selections have been disallowed.

int Selection(int n)

Return the string index of the *n*th selected string.

int SelectionIndex(int n)

Return the selection index of the *n*th string.

int Selections()

Return the total number of selected strings.

boolean Select(int n)

Return whether the *n*th string has been selected.

SEE ALSO

Interactor(2I), Button(3I)

NAME

StringChooser − dialog box for choosing items in a list

SYNOPSIS

#include <InterViews/strchooser.h>

DESCRIPTION

StringChooser is a dialog that manages keyboard focus between a StringBrowser and a StringEditor to provide a convenient interface to selecting strings. The user can select strings either by selecting them in the StringBrowser or by typing them in the StringEditor. Like Dialog, StringChooser is an abstract class; programmers must derive from StringChooser to add push buttons, etc. and to compose them into custom arrangements.

The user can focus keyboard input towards either the StringEditor or the StringBrowser. To set focus to the StringEditor, for example, the user left-clicks in its canvas to make it interpret subsequent keyboard events. When either the StringEditor or the StringBrowser relinquishes keyboard focus, the other acquires it: for example, the user can toggle focus between the two by pressing the tab key repeatedly. If an entry in the StringBrowser is selected when it loses focus, the (first) selected string will be transferred into the StringEditor.

PUBLIC OPERATIONS

StringChooser(ButtonState*, int rows, int cols, const char* sample, Alignment)
StringChooser(const char* name, ButtonState*, int rows, int cols, const char* sample, Alignment)

Construct a new StringChooser with the given button state and pop-up alignment. These constructors create a StringBrowser of the specified size and a StringEditor with the given sample string. The StringChooser, StringBrowser, and StringEditor all share the same button state.

void Select(int point)
void Select(int left, int right)
void SelectMessage()

Select an insertion point, a subrange, or the entire edit buffer in the StringEditor.

void Message(const char* text)

Set the contents of the StringEditor's edit buffer to *text*.

virtual const char* Choice()

Return the chosen string. The string is returned in a static buffer and should be copied before use.

PROTECTED OPERATIONS

StringChooser(ButtonState*, Alignment = Center)

Construct a new StringChooser with the given button state and pop-up alignment *without* creating StringBrowser and StringEditor instances. This constructor should be called in subclasses that require a specialized StringEditor or StringBrowser or both.

void Init(StringEditor*, StringBrowser*)

Initialize the StringEditor and StringBrowser that StringChooser will use. This function is useful in the constructors of StringChooser subclasses that require a specialized StringEditor or StringBrowser or both.

virtual void SwitchFocus()

Prepare to shift keyboard focus to the StringBrowser if the StringEditor is focused currently, and vice versa. Focus actually changes when HandleFocus is called (see below).

virtual boolean CanFocus(Interactor*)

Return whether or not the given interactor can be focused. This function is used to determine whether the StringChooser should switch focus in response to a downclick.

virtual void HandleFocus()

Transfer event reading to the StringBrowser or StringEditor, whichever is the current focus.

virtual void UpdateEditor()

virtual void UpdateBrowser()

Update StringEditor and StringBrowser state when the dialog is accepted or in response to a change in focus. This function guarantees that the Choice operation will return the last choice. By default, UpdateEditor changes the StringEditor's contents to match the StringBrowser selection (if any), and UpdateBrowser clears out any selected strings in the StringBrowser.

SEE ALSO

Button(3I), Dialog(3I), StringBrowser(3I), StringEditor(3I)

NAME

StringEditor − single line interactive string editor

SYNOPSIS

#include <InterViews/stringeditor.h>

DESCRIPTION

StringEditor is an interactor that provides a convenient mouse-based interactive editor for text strings. It is suitable for incorporation into other components such as dialog boxes. Clicking inside the StringEditor (or calling the Edit function) initiates an edit. Subsequent keyboard events, which need not be inside the StringEditor, are interpreted as editing operations on the text. Clicking outside the StringEdit terminates the edit. StringEditor works with either fixed width or proportionally spaced fonts.

EDITING MODEL

Text is selected with the mouse or with the keyboard. A single click of the left mouse button selects a new insertion point between characters. Dragging across the text selects a range of characters. A set of control characters is mapped into common editing operations. A character not specifically associated with commands is inserted in place of the the current selection, the replaced text is lost, and the selection becomes an insertion point following the inserted character. Commands currently defined include the following.

CharacterLeft (^B)
CharacterRight (^F)
BeginningOfText (^A)
EndOfText (^E)

Move the selection one character position to the left or right, or to the beginning or end of the text.

Erase (^H, DEL)
Delete (^D)

Delete the text of the current selection. If the selection is an insertion point, delete the previous character (Erase) or the next character (Delete) instead.

SelectAll (^U)
SelectWord (^W)

Select the entire text, or extend the selection to the left by one whole word. These commands enable common editing operations to be performed without using the mouse. For example, to replace the previous word in the text, do a SelectWord and type the new text.

SCROLLING

Strings that are too long to fit into the StringEditor can be scrolled horizontally. Middle-clicking inside the StringBrowser initiates ‘‘grab-scrolling’’. While the button is held down, the StringEditor scrolls the text to follow the mouse position, making it appear as though the user is dragging the test. Right-clicking engages ‘‘rate-scrolling,’’ a joy-stick-like scrolling interface in which the scrolling rate increases as the user drags the mouse away from the initial click point. For example, dragging the mouse rightwards after the initial click scrolls the browser rightwards at an increasing rate; dragging leftwards thereafter reduces the rate until scrolling stops entirely at the initial click point. Dragging left beyond this point makes the browser scroll in the reverse direction.

PUBLIC OPERATIONS

StringEditor(ButtonState*, const char* sample, const char* done)

Create a new StringEditor object. The ButtonState will be used to communicate the result of editing operations. An edit of the string will be terminated if any character in the string *done* is typed, and the ButtonState will be set to the terminating character. The shape of the new object is calculated from the length of the sample string.

void Message(const char* text)

Set the contents of the edit buffer to *text*.

void Select(int point)

void Select(int left, int right)
Select an insertion point or a subrange of the edit buffer.

void Edit()
void Edit(const char∗ text, int left, int right)
Initiate an edit. Specifying a string and selection range is short hand for first calling Message and Select with the corresponding parameters.

const char∗ Text()
Return the current value of the edit buffer. Note that this buffer is owned by the StringEditor, and that its contents are subject to change. It is the caller's responsibility to copy the string if the value will be needed in the long term.

Handle(Event&)
Handle the event, and read and process subsequent events until an Accept or Cancel command is executed or a down click occurs outside the StringEditor's bounds.

PROTECTED OPERATIONS

virtual boolean HandleChar(char)
void InsertText(const char∗ text, int length)
Subclasses of StringEditor can perform additional processing on the edit buffer. For instance, an editor for file names might do file name completion, or an editor for numeric input might check the validity of the string as it is entered. Derived classes should redefine the virtual function HandleChar as required. HandleChar should return true to indicate that the edit is completed, or false otherwise. InsertText can be used to insert text into the edit buffer, replacing any currently selected text.

SEE ALSO

Interactor(3I), Button(3I)

NAME

StringPool − store strings in large chunks

SYNOPSIS

#include <InterViews/strpool.h>

DESCRIPTION

A string pool is an area for character storage. Because strings in a pool cannot be independently deallocated, it is generally faster to allocate them than to use *new* in C++ or *malloc* in C.

PUBLIC OPERATIONS

StringPool(int size = 800)

Construct a string pool. The size is used to allocate contiguous blocks of storage. When one block is consumed, another block is allocated from the operating system. The default block size will fit in a small page (1024-byte).

~StringPool()

Free the storage associated with the string pool.

char*∗* Append(const char*∗*, int)

Add a string of a given length to a pool. Append does not add a null to terminate strings in the pool; if desired, null-termination must be performed explicitly by the caller.

SEE ALSO

malloc(3)

NAME

StringTable − map strings to unique pointers

SYNOPSIS

#include <InterViews/strtable.h>

DESCRIPTION

A string table is an index for character strings. The table maps strings or substrings to a unique pointer to type **StringId**. The table guarantees that the same string will always be mapped to the same pointer; thus, one can quickly compare if two strings are equal by comparing their precomputed pointers. String tables use a static StringPool(3I) to manage the storage of the characters.

PUBLIC OPERATIONS

StringTable(int size)

Construct a string table of the given size. The size will be rounded up to the next power of two so that indexing the table does not require division (just take the bitwise-and of the string's hash value and (size-1)).

˜StringTable()

Free storage associated with the string table.

StringId* Id(const char*)
StringId* Id(const char*, int)

Convert a string (null-terminated) or substring (explicit length) to a StringId. If the string cannot be found, it will be entered into the table. In any event, the pointer to the string table entry for the string is returned.

void Remove(const char*)
void Remove(const char*, int)

Delete a string from the table. This operation will not release any character storage, only the storage associated with table entry.

const char* StringId::Str()

Return the string associated with a StringId.

SEE ALSO

StringPool(3I)

NAME

Subject − object with interactive views

SYNOPSIS

#include <InterViews/subject.h>

DESCRIPTION

A subject is an object that has one or more views that it wishes to notify when it changes.

PUBLIC OPERATIONS

virtual void Attach(Interactor*)
virtual void Detach(Interactor*)

Add or remove an interactor from the subject's list of views.

virtual void Notify()

Call Update on all the views attached to the subject. Update is a virtual function defined on interactors.

void DeleteViews()

Delete all the views attached to a subject.

SEE ALSO

Interactor(3I)

NAME

Table − associative store for object pointers

SYNOPSIS

#include <InterViews/table.h>

DESCRIPTION

A table is an index where the key and data are pointers of unspecified type, declared as are **void**∗.

PUBLIC OPERATIONS

Table(int size)

Construct a table of the given size. The size will be rounded up to the next power of two so that indexing the table does not require division (just take the bitwise-and of the key and (size-1)).

˜Table()

Free storage associated with the table.

void Insert(void∗ key, void∗ value)

Add (*key*, *value*) pair to the table. Insert does not prevent a key from being entered in the table more than once.

boolean Find(void∗& value, void∗ key)

If *key* has been inserted into the table, set *value* to the associated value and return true. Otherwise, do not modify *value* and return false. In the presence of duplicate keys, Find gets the most recently inserted value.

void Remove(void∗ key)

Delete *key* from the table. If *key* appears in the table more than once, the most recently inserted entry is removed.

NAME

TextBuffer − operations on unstructured text

SYNOPSIS

#include <InterViews/textbuffer.h>

DESCRIPTION

TextBuffer defines common editing, searching, and text movement operations on a buffer of unstructured text. Text positions are specified by an index into the buffer and logically refer to positions between characters. For example, the position referred to by the index *0* is before the first character in the text. Indices can be compared for equality or ordering, but they should not be used to directly access the buffer because TextBuffer might rearrange the text to improve the efficiency of some operations.

PUBLIC OPERATIONS

TextBuffer(char∗ buffer, int length, int size)
˜TextBuffer()

Create or destroy an instance of TextBuffer. All operations on the text contained in *buffer* should be performed through TextBuffer functions. The text is assumed to be of length *length*, and the total available buffer size is *size*.

int Search(Regexp∗ regexp, int index, int range, int stop)
int ForwardSearch(Regexp∗ regexp, int index)
int BackwardSearch(Regexp∗ regexp, int index)

Search for a match with the regular expression *regexp*, beginning at position *index*. Search searches the part of the buffer specified by *range* and *stop* and returns the index of the beginning of the matched text. Positive values of *range* specify forward searches, and negative values specify backward searches. In either case, the matched text will not extend beyond the position given by *stop*. ForwardSearch searches for matches from *index* to the end of the text and returns the index of the end of the match. BackwardSearch searches from *index* to the start of the text and returns the index of the beginning of the match. All three functions return a negative number if there was no match.

int Match(Regexp∗ regexp, int index, int stop)
boolean ForwardMatch(Regexp∗ regexp, int index)
boolean BackwardMatch(Regexp∗ regexp, int index)

Attempt to match the regular expression *regexp* at the position *index*. Match returns the length of the matching string, or a negative number if there was no match. Matching will not succeed beyond the position given by *stop*. ForwardMatch looks for a match that begins at *index*. BackwardMatch looks for a match that ends at *index*.

int Insert(int index, const char∗ string, int count)
int Delete(int index, int count)
int Copy(int index, char∗ buffer, int count)

Edit the text in the buffer. Insert inserts count characters from *string* at the position *index*. It returns the actual number of characters inserted, which might be less than count if there is insufficient space in the buffer. Delete deletes *count* characters from the buffer. A positive *count* deletes characters after *index*, and a negative value deletes character before *index*. Delete returns the actual number of characters deleted, which might be less than *count* if index is near the beginning or the end of the text. Copy copies *count* characters into *buffer*. A positive *count* copies characters after *index* and a negative *count* copies characters before *index*. Count returns the actual number of characters copied. It is the caller's responsibility to ensure that *buffer* contains sufficient space for the copied text.

int Height()
int Width()
int Length()

Return information about the text. Height returns the number of lines in the text, Width returns the number of characters in the longest line, and Length returns the total number of characters.

const char∗ Text()
const char∗ Text(int index)
const char∗ Text(int index1, int index2)
char Char (int index)

Access the contents of the text. Char returns the character immediately following *index*. The three Text calls return pointers to character strings representing the text. They make various guarantees about the format of the returned string. With no parameters, Text returns a pointer to a string that contains the entire text of the buffer. With a single parameter the string contains at least the text from *index* to the end of the line. With two parameters, the returned string contains at least the text between *index1* and *index2*. In any case, the returned string should be considered temporary and its contents subject to change. To maximize efficiency, you should prefer the more restricted forms of Text.

int LineIndex(int line)
int LinesBetween(int index1, int index2)
int LineNumber(int index)
int LineOffset (int index)

Map between text indices and line and offset positions. LineIndex returns the index of the beginning of line *line*. LineNumber returns the number of the line that contains *index*. LineOffset returns the offset of *index* from the beginning of its containing line. LinesBetween returns the difference between the numbers of the lines containings *index1* and *index2*; a return value of zero indicates that *index1* and *index2* are on the same line, and a positive value indicates that the line containing *index2* is after the line containing *index1*. Lines are numbered starting from zero.

int PreviousCharacter(int index)
int NextCharacter(int index)

Return the index immediately following or preceding *index*. The returned value is never before the beginning or after the end of the text.

boolean IsBeginningOfText(int index)
int BeginningOfText()
boolean IsEndOfText(int index)
int EndOfText()

Return the index of the beginning or end of the text, or query whether *index* is at the beginning or end of the text.

boolean IsBeginningOfLine(int index)
int BeginningOfLine(int index)
int BeginningOfNextLine(int index)
boolean IsEndOfLine(int index)
int EndOfLine(int index)
int EndOfPreviousLine(int index)

Return information about the line structure of the text around *index*. BeginningOfLine returns the index of the beginning of the line containing *index*. BeginningOfNextLine returns the index of the beginning of the next line that begins after *index*. EndOfLine returns the index of the end of the line containing *index*. EndOfPreviousLine returns the index of the end of the last line that ends before *index*. The beginning of a line is logically immediately after a newline character, and the end of a line is logically immediately before a newline character. The beginning and end of the text are considered to be the beginning and end of the first and last lines, respectively.

boolean IsBeginningOfWord(int index)
int BeginningOfWord(int index)
int BeginningOfNextWord(int index)
boolean IsEndOfWord(int index)
int EndOfWord(int index)
int EndOfPreviousWord(int index)

Return information about the word structure of the text around *index*. BeginningOfWord returns the index of the beginning of the word containing *index*. BeginningOfNextWord return the index of the beginning of the nest word that begins after *index*. EndOfWord returns the index of the end of the word that contains *index*. EndOfPreviousWord returns the index of the end of the last word that ends before *index*. A word is defined as a sequence of alpha-numeric characters. The beginning and end of the text are considered to be the beginning and end of the first and last words, respectively.

SEE ALSO

Regexp(3I)

NAME

TextDisplay − unstructured text display

SYNOPSIS

#include <InterViews/textdisplay.h>

DESCRIPTION

A TextDisplay manages and displays an array of lines of text. Operations are provided to insert and delete lines, to insert and delete text within a line, and to scroll the display. TextDisplays are typically used to implement interactors that require non-trivial display of unstructured text.

The array is addressed by a line number and an index into the line. Text can be inserted or deleted at arbitrary positions. By default, the first character in line number zero is positioned with its top left corner at the top left corner of the TextDisplay. The display is automatically updated following modifications to the text or when scrolling. Each character in the array has an associated text style. Operations are provided to apply, add, or remove font styles to a range of the text. TextDisplay can be used with both constant-width and proportionally-spaced fonts and with arbitrary geometric transformations.

PUBLIC OPERATIONS

TextDisplay()
˜TextDisplay(boolean autosized = false)

Create or destroy a TextDisplay. If *autosized* is *true*, the TextDisplay will automatically grow its size as text is added. The default is to keep the size constant.

void LineHeight(Coord lineheight)
void TabWidth(Coord tabwidth)

Lines of text will be positioned with baselines separated by *lineheight*. Tab characters in the text will cause the following character to be positioned an integral multiple of *tabwidth* from the beginning of the line.

void Resize(Coord xmin, Coord ymin, Coord xmax, Coord ymax)
void Bounds(Coord& xmin, Coord& ymin, Coord& xmax, Coord& ymax)

Specify or query the size of the display. Only lines of text that fall completely inside the specified region will be displayed. TextDisplay will not draw on any part of the canvas outside the specified bounds. Conversely, TextDisplay is free to draw on any part of the canvas within the specified bounds.

void Draw(Painter∗, Canvas∗)

Specify the painter and canvas to use for drawing operations; the painter specifies the font, colors, and geometric transformation. Draw should be called before performing any operation on the TextDisplay which produces output or requires graphical information, or when there is a possibility that the painter or canvas has changed since the function was last called.

void Redraw(Coord left, Coord bottom, Coord right, Coord top)

Redraw a specified region of the display.

void Scroll(int line, Coord x, Coord y)

Scroll the display so that line *line* is positioned with its upper-left corner at (x, y). There are no restrictions on the arguments: it is possible to scroll the display so that no lines are visible.

void InsertLinesAfter(int line, int count)
void InsertLinesBefore(int line, int count)
void DeleteLinesAfter(int line, int count)
void DeleteLinesBefore(int line, int count)

Insert or delete whole lines of text. Line *line* is not affected by the operation. Other lines will move up or down to accommodate the changes. Newly inserted lines are blank. The specified line need not refer to an existing line.

void InsertText(int line, int index, const char∗, int count)
void DeleteText(int line, int index, int count)

void ReplaceText(int line, const char*, int count)

Modify the text within line *line*. InsertText and DeleteText will cause the following characters on the line to move to accommodate the changes. ReplaceText replaces the entire text of the line. If the specified line is non-existent, a new line will be created.

void Style(int line1, int index1, int line2, int index2, int style)
void AddStyle(int line1, int index1, int line2, int index2, int style)
void RemoveStyle(int line1, int index1, int line2, int index2, int style)

Modify the styling of a range of text. Style replaces any existing style; AddStyle adds style *style* in addition to any existing styles; RemoveStyle removes style *style* without affecting other existing styles. Styles are specified as any combination of the constants Plain, Boldface, Underlined, and Reversed.

void CaretStyle(int style)
void Caret(int line, int index)

Control the shape and position of a caret. Valid caret styles are currently NoCaret, BarCaret, UnderscoreCaret and OutlineCaret. TextDisplay does not automatically adjust the position of the caret following insertions and deletions.

int LineNumber(Coord y)
int LineIndex(int line, Coord x)

Map x and y display coordinates into line and index text coordinates. LineNumber returns the number of the line that contains the specified vertical position *y*. LineIndex returns the index into line *line* that best corresponds to the specified horizontal position *x*.

Coord Width()
Coord Height()

Return the width or the height of the text currently displayed by the TextDisplay. Width returns the width of the longest line in the display. Height returns the distance from the top of the topmost line to the bottom of the bottommost line. Note that these dimensions do not necessarily correspond to the TextDisplay's bounds as returned by Bounds.

Coord Base(int line)
Coord Top(int line)
Coord Left(int line, int index)
Coord Right(int line, int index)

Map line and index text coordinates into x and y display coordinates. The return values define a bounding box for the character specified by *line* and *index*.

SEE ALSO

Painter(2I), Canvas(2I)

NAME

TextEditor − basic unstructured text editing

SYNOPSIS

#include <InterViews/texteditor.h>

DESCRIPTION

A TextEditor is an interactor that provides an interactive interface for simple text editing of a TextBuffer. TextEditor uses an editing model based on a single current selection. Editing operations operate on the text in the selection, alter the position or size of the selection, or scroll the display to view other parts of the text. TextEditor interprets a perspective for interactive scrolling using a scroller.

PUBLIC OPERATIONS

TextEditor(int rows, int cols, int tabsize, int highlight)
~TextEditor()

Create or destroy an instance of TextEditor. The natural size of a TextEditor is specified by *rows* and *columns*. The TextEditor will be tall enough to display *rows* lines of text in the current font, and it will be wide enough to display *columns* characters. For proportionally-spaced fonts, the width of the character 'n' is taken to be representative. Tab characters in the text are expanded to multiples of *tabsize* character widths. The current selection highlighted with the text style *highlight*.

void Edit(TextBuffer*, int index = 0)

Specify the text buffer to edit. A text buffer must be specified before the TextEditor is displayed. The optional *index* parameter specifies the initial selection point. If necessary, the TextEditor is scrolled so that the selection point is visible.

int Dot()
int Mark()

The current selection is bounded by two indices into the text buffer. By convention these indices are called dot and mark. Dot is the position at which text will be inserted and deleted. Mark locates the end of the selection that moves as the selection is modified. When the selection is empty (an insertion point), dot and mark are equal.

void InsertText(const char* string, int count)
void DeleteText(int count)
void DeleteSelection()

Edit the text buffer at the current selection. InsertText inserts count characters from string after dot. The selection becomes an insertion point following the newly inserted text. DeleteText deletes count characters at dot. If count is positive, the characters following dot are deleted; if count is negative the characters before dot are deleted. The selection becomes an insertion point in the place of the deleted text. DeleteSelection deletes the text between dot and mark, changing the selection to an insertion point.

void BackwardCharacter(int count)
void ForwardCharacter(int count)
void BackwardLine(int count)
void ForwardLine(int count)
void BackwardWord(int count)
void ForwardWord(int count)
void BackwardPage(int count)
void ForwardPage(int count)

Move the current selection forward or backward by the specified number of the specified units. The default movement is one unit. The selection is not moved before the beginning of after the end of the text.

void BeginningOfLine()
void EndOfLine()

void BeginningOfWord()
void EndOfWord()
void BeginningOfSelection()
void EndOfSelection()
void BeginningOfText()
void EndOfText()

Move the current selection forwards or backwards to the specified feature of the text.

void ScrollToSelection(boolean always = false)
void SetScrollAlignment(Alignment)
Alignment GetScrollAlignment()

ScrollToSelction scrolls the display so that dot will be visible. If dot is currently visible and *always* is false, no scrolling takes place; if dot is not visible or *always* is true, the display is scrolled so that dot will be positioned as close as possible to the position specified by SetScrollAlignment. GetScrollAlignment returns the current scroll alignment. The default alignment is Center.

void ScrollToView(Coord x, Coord y)
void ScrollBy(Coord dx, Coord dy)

ScrollToView scrolls the display so that the text currently at the position (*x*, *y*) will be visible. If the specified point is currently visible, no scrolling takes place. ScrollBy specifies an amount by which to scroll the display. Positive values scroll the display upwards and to the right. In each case, the final position of the display is limited so that some text will be visible.

void GrabScroll(Event&)
void RateScroll(Event&)

Initiate grab scrolling or rate scrolling. Once called, GrabScroll (RateScroll) polls the current mouse position as long as the middle (right) button is pressed. As GrabScroll polls, it scrolls the text to follow the mouse position, making it appear as though the user is dragging the text itself. Each time RateScroll polls the mouse, it scrolls the text by an amount corresponding to the differential between the current and initial mouse positions. This creates the illusion of a joystick interface: the further away the mouse is from its initial position, the greater the scrolling increment and the faster the text appears to scroll; scrolling slows down, stops, and reverses direction as the user drags the mouse back towards and past the initial position.

void Select(int dot)
void Select(int dot, int mark)
void SelectMore(int mark)
void SelectAll()

Modify the current selection in terms of text indices. With a single parameter, Select moves both dot and mark to the specified index. With two parameters, dot and mark can be controlled independently. SelectMore leaves dot unchanged and moves mark. SelectAll sets dot to the end of the text and mark to the beginning.

int Locate(Coord x, Coord y)

Return the text index most closely matching the point (*x*, *y*).

SEE ALSO

Interactor(2I), TextBuffer(3I)

NAME

Transformer − transformation matrix for Painter

SYNOPSIS

#include <InterViews/transformer.h>

DESCRIPTION

A transformer object is a transformation matrix for use in translating coordinates. Most users will not need to access a transformer directly; the *Painter*(3I) operations will be sufficient.

PUBLIC OPERATIONS

Transformer(Transformer* = nil)

Create a copy of the given transformer. The identity transformer is returned by default.

Transformer(float a00, float a01, float a10, float a11, float a20, float a21)

Create a matrix with the given values.

void GetEntries(float& a00, float& a01, float& a10, float& a11, float& a20, float& a21)

Get the transformer's matrix entries.

void Rotate(float angle)

Modify the matrix to rotate coordinates by *angle* degrees.

boolean Rotated(float tolerance = 1e-6)
boolean Rotated90(float tolerance = 1e-6)

Rotated returns *true* if coordinates would be rotated more than plus or minus arcsin(tolerance) degrees when transformed. Rotated90 returns *true* only if the angle of rotation is 90 plus or minus arcsine(tolerance) degrees.

void Scale(float sx, float sy)

Modify the matrix to multiply coordinates by *sx* and *sy*.

boolean Scaled(float tolerance = 1e-6)
boolean Stretched(float tolerance = 1e-6)

Scaled returns *true* if coordinates are scaled when transformed beyond 1 plus or minus the given tolerance. Stretched returns *true* if the scaling is not the same in both dimensions.

void Translate(float dx, float dy)

Add (*dx*, *dy*) to coordinates after multiplying by the matrix.

boolean Translated(float tolerance = 1e-6)

Return *true* if coordinates are translated when transformed and the absolute value of the translation in either dimension is greater than the tolerance.

void Transform(Coord& x, Coord& y)
void Transform(Coord x, Coord y, Coord& tx, Coord& ty)
void Transform(float x, float y, float& tx, float& ty)
void TransformList(Coord x[], Coord y[], int n)
void TransformList(Coord x[], Coord y[], int n, Coord tx[], Coord ty[])

Transform a point or set of points using the transformer's matrix.

void TransformRect(Coord& x0, Coord& y0, Coord& x1, Coord& y1)
void TransformRect(float& x0, float& y0, float& x1, float& y1)

Transform the rectangle defined by two points and return the bounding box of the result.

void InvTransform(Coord& tx, Coord& ty)
void InvTransform(Coord tx, Coord ty, Coord& x, Coord& y)
void InvTransform(float tx, float ty, float& x, float& y)
void InvTransformList(Coord tx[], Coord ty[], int n)
void InvTransformList(Coord tx[], Coord ty[], int n, Coord x[], Coord y[])

Perform a reverse transformation on a point or set of points.

void InvTransformRect(Coord& x0, Coord& y0, Coord& x1, Coord& y1)
void InvTransformRect(float& x0, float& y0, float& x1, float& y1)

Inverse transform the rectangle defined by two points and return the bounding box of the result.

void Premultiply(Transformer*)
void Postmultiply(Transformer*)

Multiply in place the transformer's matrix by the given transformer's matrix.

SEE ALSO

Painter(2I)

NAME

Tray − compose interactors into arbitrary or constrained layouts

SYNOPSIS

#include <InterViews/tray.h>

DESCRIPTION

A tray is a scene of interactors that overlap, tile, or have other constraints on their alignment relative to each other. One of the interactors in a tray can serve as a background for the other interactors.

Constraints on the layout of interactors are made by *aligning* interactors to each other. Two interactors are aligned by specifying which edges coincide. For example, the lower left corner of one interactor may be aligned to the upper right of another. **TGlue** objects can be used to introduce transparent space between aligned interactors. TGlue has a natural size, shrinkability, and stretchability (though TGlue objects are *not* interactors). Trays stretch or shrink the TGlue along with the aligned interactors to satisfy the alignment constraints.

The tray adopts the shape of the background interactor if there is one; otherwise, the tray's shape depends on the shapes of its components and potentially on their alignments. By default, a tray without a background takes on a width and height equal to the largest of the widths and heights of its components. However, alignments involving the tray can in effect override this default.

For example, if the left and right sides of a particular component are aligned to the tray's left and right sides, respectively, then the tray's sides are constrained to coincide with the component's sides. Thus the tray will adopt the width, horizontal shrinkability, and horizontal stretchability of that component. Another example: To ensure that a tray circumscribes a collection of (mutually aligned) components, align the outer edges of the components on the periphery of the collection to the outer edges of the tray, thereby constraining the tray to assume the shape of the collection.

TGLUE PUBLIC OPERATIONS

TGlue(int w = 0, int h = 0, int hstretch = hfil, int vstretch = vfil);

Define TGlue of a minimum size. The TGlue can stretch from the given size but cannot shrink.

TGlue(int, int, int hshrink, int hstretch, int vshrink, int vstretch);

Define general TGlue with a given natural size (width and height), shrinkability, and stretchability.

TRAY PUBLIC OPERATIONS

Tray(Interactor* background = nil)

Create a new tray, optionally having an interactor as a background.

void Align(Alignment, Interactor*, TGlue* = nil)
void Align(Alignment, Interactor*, Alignment, Interactor*, TGlue* = nil)

Align an interactor to another interactor (the tray itself by default), optionally with TGlue between them. Align operations insert interactors into the tray if they have not been inserted already.

void Align(Alignment, Interactor*, ..., Interactor* = nil)

Apply an alignment to a set of interactors. This operation is shorthand for aligning the interactors to each other explicitly. For example, Align(Left, i1, i2, i3, i4) aligns the left sides of interactors i1 through i4. Two to seven interactors can be aligned at once.

void HBox(Interactor*, ..., Interactor* = nil)
void VBox(Interactor*, ..., Interactor* = nil)

The HBox and VBox operations align the specified interactors such that they tile left-to-right and top-to-bottom, respectively. These operations align in one dimension only. Two to seven interactors can be aligned at once. If the first (last) component is the tray or the background interactor, then the leftmost (rightmost) component will be aligned with the left (right) side of the tray.

void Insert(Interactor*)

Insert an interactor into the tray without an alignment. The interactor will appear in the lower left corner of the tray.

void Change(Interactor*)

Notify the tray that the given interactor's shape has changed. The tray will recompute the layout of its component interactors to satisfy any alignments. If the tray does not contain a background, then a change in the shape of one of its components may in turn change the tray's shape. If the tray has a background, then the tray's shape will change only if the shape of the background changes.

void Remove(Interactor*)

Take an interactor out of a tray and eliminate any alignments that have been made to it.

SEE ALSO

Interactor(3I), Scene(3I), Shape(3I)

NAME

Viewport − scrollable view

SYNOPSIS

#include <InterViews/viewport.h>

DESCRIPTION

Viewport is a subclass of MonoScene that allocates its component exactly as much space as it wants (determined from its shape). This space will be clipped to the viewport's size and the viewport maintains a perspective to adjust the portion of the component that is actually displayed. If the component is smaller than the viewport's canvas, then the viewport background will be filled with a light gray pattern.

PUBLIC OPERATIONS

Viewport(Interactor* = nil, Alignment = Center)

Construct a new viewport. If the interactor is not nil, it is inserted as the viewport's component. The second parameter determines where the component is placed relative to the viewport.

Insert(Interactor*)

Set the component interactor.

void Change(Interactor*)

A viewport does not propagate changes. Thus, if the component's shape changes the viewport simply resizes the component according to the new shape.

void Remove(Interactor*)

Take out the component interactor.

virtual void Adjust(Perspective&)

Change the viewport's perspective to the given one. This operation will cause the desired area of the component to be drawn.

void AdjustTo(float px, float py, float zx, float zy)
void AdjustBy(float dpx, float dpy, float dzx, float dzy)
void ScrollTo(float px, float py)
void ScrollXTo(float px)
void ScrollYTo(float py)
void ScrollBy(float dpx, float dpy)
void ScrollXBy(float dpx)
void ScrollYBy(float dpy)
void ZoomTo(float zx, float zy)
void ZoomXTo(float zx)
void ZoomYTo(float zy)
void ZoomBy(float dzx, float dzy)
void ZoomXBy(float dzx)
void ZoomYBy(float dzy)

Short-hand operations for explicit manipulation of the viewport's perspective.

float XPos()
float YPos()
float XMag()
float YMag()

Short-hand for retrieving information about the current perspective.

SEE ALSO

Perspective(3I), Scene(3I), Shape(3I)

NAME

World − root scene for a display

SYNOPSIS

#include <InterViews/world.h>

DESCRIPTION

A world is an application's root scene for a particular display. The world must be created before any other InterViews objects are created. Scene operations such as Insert, Raise, and Lower can be used to affect the display. Thus, an interactor is placed as a top-level window on the screen by inserting it into the world.

PUBLIC OPERATIONS

World(const char∗ classname, int& argc, char∗ argv[])

Construct the world object for a display. Top-level interactors that have not called SetClassName will use the world's class name when looking up default attributes. Top-level interactors that have not called SetInstance will use the world's instance name when looking up attributes. The world's instance name comes from the -name option if it was given; otherwise the instance name comes from the environment variable RESOURCE_NAME if it is nonnil. If the name has still not been defined, then argv[0] with all leading directories stripped is used. The following arguments are interpreted automatically and are removed from *argv*:

−background	next argument sets the background color
−bg	same as −background
−display	next argument specifies the target workstation display
−foreground	next argument sets the foreground color
−fg	same as −foreground
−fn	same as −font
−font	next argument sets the text font
−geometry	next argument sets the first top-level interactor's position and size
−iconic	starts up the first top-level interactor in iconic form
−name	next argument sets the instance name of all top-level interactors that don't have their own instance names
−reverse	swaps default foreground and background colors
−rv	same as −reverse
−title	next argument sets the first top-level interactor's title bar name
−xrm	next argument sets an ''attribute: value'' property

The geometry specification has the form ''=WxH+XOFF+YOFF''. A negative XOFF (YOFF) specifies the offset of the interactor's right (bottom) edge from the right (bottom) side of the screen. The constructor sets *argc* to the number of uninterpreted arguments that remain.

World(const char∗, OptionDesc∗, int&, char∗[])

Construct a world as above, but using an array of option descriptors to parse application-specific attributes. OptionDesc contains four fields: a *name* used on the command line, a *path* specifying the attribute, a *style* specifying where the value is, and a default *value*. Valid styles are *OptionPropertyNext* (use next argument as an attribute:value pair, not just the value), *OptionValueNext* (use next argument as value), *OptionValueImplicit* (use default value), *OptionValueIsArg* (use argument as value), and *OptionValueAfter* (use remainder of argument as value).

World(const char∗, PropertyData∗, OptionDesc∗, int&, char∗[])

Construct a world as above and specify a set of initial attribute values. PropertyData contains three string fields: a *path* specifying the attribute, a *value* specifying the value, and a *type* specifying the type name. Attributes are entered in the following order: first any application defaults (specified by the PropertyData array), then user defaults (usually in $HOME/.Xdefaults), and then command-line arguments. Thus, command-line arguments override user defaults, and both override application defaults.

World(const char∗ instance = nil, const char∗ display = nil)

Construct the world object for a specific display. This constructor is for backward compatibility and should no longer be used. Since the application cannot get its instance name for looking up resources from the command line, the first parameter specifies the instance name to use rather than the class name. If *display* is omitted, then the DISPLAY environment variable is used to connect to the appropriate window server. The format of the string is ‘‘hostname:N.S’’ where N is the display number and S is the screen number (S is optional).

void InsertPopup(Interactor*)
void InsertPopup(Interactor*, Coord x, Coord y, Alignment = BottomLeft)

Insert a popup interactor into the world, optionally aligned with respect to a specific position relative to the lower-left corner of the screen, without any interaction whatsoever with window managers. A popup interactor inserted without a specific position will be centered on the screen, since the user cannot use a window manager's help to place the interactor.

void InsertTransient(Interactor*, Interactor*)
void InsertTransient(Interactor*, Interactor* owner, Coord x, Coord y, Alignment = BottomLeft)

Insert a transient interactor into the world on behalf of another interactor, optionally aligned with respect to a specific position relative to the lower-left corner of the screen. Most window managers will not decorate the transient interactor or ask the user to place it. Some window managers may also unmap the transient interactor when its *owner* interactor is iconized.

void InsertToplevel(Interactor*, Interactor*)
void InsertToplevel(Interactor*, Interactor* group, Coord x, Coord y, Alignment = BottomLeft)

Insert a toplevel interactor into the world, optionally aligned with respect to a specific position relative to the lower-left corner of the screen. A nonnil *group* informs window managers that this toplevel interactor is part of a group of toplevel interactors led by *group*. Some window managers may unmap the toplevel interactor when its ‘‘leader’’ interactor is iconized.

void InsertApplication(Interactor*)
void InsertApplication(Interactor*, Coord x, Coord y, Alignment = BottomLeft)

Insert an application interactor into the world, optionally aligned with respect to a specific position relative to the lower-left corner of the screen. Window and session managers will consider an application interactor to be the program's main top-level interactor.

void InsertIcon(Interactor*)
void InsertIcon(Interactor*, Coord x, Coord y, Alignment = BottomLeft)

Insert an icon interactor into the world, optionally aligned with respect to a specific position relative to the lower-left corner of the screen, but do not make it visible (‘‘map’’ it). Another top level interactor may call Interactor::SetIconInteractor to tell window managers to use this interactor for its icon. InsertIcon is called automatically when a top level interactor is inserted if the interactor has an icon and the icon has not yet been inserted into the world.

int Width()
int Height()

Return the width and height in pixels of the screen.

int NPlanes()

Return the number of user-accessible bit planes the workstation display has.

int NButtons()

Return the number of buttons on the input pointing device associated with the world.

int PixelSize()

Return the natural size in bits for storing a pixel.

unsigned ParseGeometry(const char* spec, Coord& x, Coord& y, int& w, int& h)

Interpret a geometry specification (e.g., =80x24+1+1), returning a bit mask that identifies which fields were specified.

void SetHint(const char*)

Provide window system specific information. See next section.

void RingBell(int vol)
Ring the workstation bell with a volume equal to a percentage of the possible range, with 0 the quietest and 100 the loudest.

void SetKeyClick(int vol)
Set the volume of the audible keyclick equal to a percentage of the possible range, with 0 the quietest and 100 the loudest.

void SetAutoRepeat(boolean on)
Turn on or off autorepeat for the keyboard.

void SetFeedback(int threshold, int scale)
Set the amount of magnification to give the user's movements with the pointing device. The *threshold* is the minimum distance after which to scale the movement and the *scale* is the factor by which to magnify the movement.

void SetCurrent()
Set the current world. Some routines use global state associated with a world object. Creating a world automatically sets the state to that of the new world.

int Fileno()
Return the file descriptor of the connection to the window server.

WINDOW SYSTEM HINTS

The performance of some InterViews library functions depends heavily on the characteristics of the underlying window system. Window system hints provide a way for the programmer or user to provide information to maximize performance. There is usually no need to provide hints: the default behavior is appropriate for most installations. By convention, a hint consists of a name and a value, separated by a colon. Meaningful names and values are implementation-specific. Non-meaningful hints are simply ignored. Currently defined hints include:

RubberbandPixel
The value is a hexadecimal number which is used to control the drawing of rubber bands. On many window systems, rubber bands are drawn in XOR mode so that they can easily be erased without damaging underlying screen contents. On a color workstation, the resulting color depends on the screen contents and the XOR mask pixel. The mask should be chosed to maximize the visibility of the rubber band, but the best value may depend on the workstation color rendering model and on how an application uses colors.

TransformFonts
The value is one of ‘‘off’’, ‘‘on’’, or ‘‘cache’’. On many window systems, drawing text which has been rotated or scaled is much slower than drawing untransformed text. This hint suggests a strategy for optimizing text transformations. ‘‘Off’’ suggests that fonts should not be transformed. This usually results in text drawn with the origins of the characters at the correct positions but with untransformed character glyphs. ‘‘On’’ suggests that the character glyphs should be transformed. ‘‘Cache’’ further suggests that transformed glyphs be cached to speed up future text operations that might require the same glyph. Cached glyphs consume both application and window system resources but are drawn much faster.

Dash The value is one of ‘‘none’’, ‘‘thin’’, or ‘‘all’’. Some window systems draw dashed lines slowly, particularly if the line is not of minimal thickness. InterViews can simulate dashed lines with stipple patterns, but the results may be less that satisfactory. This hint suggest a strategy for dashed lines. ‘‘None’’ suggests that all dashed lines should be simulated with stipple patterns; ‘‘thin’’ suggests that minimal-width lines should be dashed, but that thicker dashed lines should be simulated; ‘‘all’’ suggests that lines of all thicknesses should be dashed.

Window system hints can be specified through the function *World::SetHint* or as application attributes. *SetHint* allows an application to specify hints dynamically. For example, an application that uses transformed text might temporarily disable font transformations to speed up drawing of a rough draft and re-enable transformations to show the final version. Attributes allow users to control the default values of hints from their user preference files or from the application command line. Attribute names are derived from the corresponding hint names by appending the string ''Hint''. For example, the RubberbandPixel hint can be set through the ''RubberbandPixelHint'' attribute.

SEE ALSO

Interactor(3I), Scene(3I)

NAME

WorldView − manipulate root scene

SYNOPSIS

#include <InterViews/worldview.h>

DESCRIPTION

WorldView is a class for writing window managers that manipulate a root scene. The operations manipulate objects of type **RemoteInteractor**, which are interactors that exist in a different address space than the window manager. Typically, a subclass of **WorldView** is defined that handles input events according to a particular window manager style.

PUBLIC OPERATIONS

WorldView(World*, Sensor*, Painter*)

Define a worldview interactor associated with a given world, and pass the sensor and painter to the interactor constructor.

RemoteInteractor Find(Coord x, Coord y)

Return the interactor under the given (absolute) coordinates.

RemoteInteractor Choose(Cursor*, boolean waitforup = true)

Interactively choose an interactor by changing the cursor to the given one and waiting for a button to be pressed. If *waitforup* is true, then wait until the button is released before returning.

virtual void InsertRemote(RemoteInteractor)

Handle a request from an application (process) to insert an interactor into the root scene. The default operation is a nop.

virtual void ChangeRemote(RemoteInteractor, Coord left, Coord top, int w, int h

Handle a request from an application (process) to change an interactor's canvas to the specified position and size. The default operation is simply to perform a Change operation with the given parameters.

void Move(RemoteInteractor, Coord left, Coord top)
void Change(RemoteInteractor, Coord left, Coord top, int w, int h)

The Move operation sets the interactors position given the upper left corner. Change both moves and resizes the interactor.

void Raise(RemoteInteractor)

Raise the interactor's canvas to be above all others on the screen.

void Lower(RemoteInteractor)

Lower the interactor's canvas to be below all others on the screen.

void Focus(RemoteInteractor)

Set input focus to the interactor. If nil is passed, the input focus is based on the location of the pointing device.

RemoteInteractor GetFocus()

Return the current interactor that has input focus (or nil if none).

void GetList(RemoteInteractor*&, int&)

Return all the top-level interactors and a count of how many there are. The space for the list is automatically allocated.

void FreeList(RemoteInteractor*)

Free the space allocated by a GetList call.

void GetInfo(RemoteInteractor, Coord& x1, Coord& y1, Coord& x2, Coord& y2)

Get the current position of an interactor.

RemoteInteractor GetIcon(RemoteInteractor)

Gets the icon for the specified Interactor. If the argument Interactor is an icon, then GetIcon gets the original Interactor.

void AssignIcon(RemoteInteractor i, RemoteInteractor icon)
Associate *icon* as the icon interactor for *i*.

void UnassignIcon(RemoteInteractor)
Remove the icon association for the given interactor.

char* GetName(RemoteInteractor)
Returns the name of the Interactor (as set by Scene::SetIcon(const char*)).

RedrawAll()
Force all the interactors on the screen to redraw themselves.

PROTECTED OPERATIONS

void GrabMouse(Cursor*)
void UngrabMouse()
Get or release control of the pointing device. GrabMouse will wait until the device is available and then sets the cursor to the given cursor.

boolean GrabButton(unsigned mask, Cursor*)
void UngrabButton(unsigned mask)
Get or release control of a button or set of buttons. The cursor is used when the button is pressed. GrabButton returns *false* if the button is not available. The button mask is defined by the constants in ‘‘<InterViews/Xlib.h>’’.

void Lock()
void Unlock()
Get or release exclusive access to the server. While locked, the server will ignore other clients.

void ClearInput()
Discard any pending input events.

void MoveMouse(Coord x, Coord y)
Set the pointing device position to the given screen coordinates.

SEE ALSO

Interactor(3I), Scene(3I)