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 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

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 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 , Canvas , Painter , Perspective ,

Scene , Sensor , Shape , World