NAME

libgraphic − InterViews structured graphics library

SYNOPSIS

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

DESCRIPTION

The *libgraphic* library provides a set of classes for defining persistent structured graphics. **Graphic** is the base class for structured graphics objects; prederived classes classes include simple graphical objects (**Point, Ellipse, Spline, Label,** etc.), a composite object (**Picture**), and an object that is a variation on another object (**Instance**). Each graphic can be manipulated and transformed independently of others. Functions are provided for computing the graphic's bounding box and whether or not it intersects a given point or rectangle.

Graphics are persistent in that each defines a representation of itself that can be automatically written to and read from disk. The **Graphic** class is derived from **Persistent**, described in Persistent(3I), which provides this functionality in a general persistent object class. Graphics use persistent paint objects that are analogous to the paint objects used by painters. Persistent paint objects are defined in <InterViews/Graphic/ppaint.h>.

Graphic(3I) contains all you need to know to use **Graphic** and derived classes without the persistence feature. If persistent graphics are required, then you should read Persistent(3I).

The *libgraphic* provides two additional classes for use with structured graphics objects. **Damage** supports incremental redraw of graphical objects whose appearance has been modified. **GraphicBlock** is an interactor that displays a structured graphics object. A GraphicBlock interprets a perspective to allow scrolling and zooming of the graphic it contains.

SEE ALSO

libInterViews(3I)

NAME

Damage − manages and repairs damage to graphics

SYNOPSIS

#include <InterViews/Graphic/damage.h>

DESCRIPTION

Damage is a class that keeps track of and repairs portions of a graphic that have been *damaged*, that is, that need to be redrawn. A graphic is damaged when its (component's) appearance changes. Damage objects try to minimize the amount of redraw needed to repair a graphic. They are most useful when the graphic is complicated enough that it would be undesirable to redraw the entire canvas when the graphic is modified. Damage objects do *not* eliminate the need for an interactor's *Redraw* member function, though they could be used to implement it.

PUBLIC OPERATIONS

Damage(Canvas* = nil, Painter* = nil, Graphic* = nil)

Create a new Damage object for a graphic on the given canvas. The painter is used to erase damaged areas using its ClearRect function prior to redrawing them.

void Incur(Graphic*)
void Incur(BoxObj&)
void Incur(Coord left, Coord bottom, Coord right, Coord top)

Notify the damage object that a rectangular area has been damaged, either by passing the graphic that contributed the area or the area itself.

void Repair()

Repair all damage incurred since the last Repair (or since the damage object was created if this is the first Repair).

void Added(Graphic*)

Notify the damage object that the given graphic was appended (*not* inserted) to the graphic but never drawn. The damage object can thus be responsible for drawing newly added graphics as well.

void Reset()

Reset the damage object. Any damage incurred is lost.

boolean Incurred()

Returns true if any damage has been incurred.

void SetCanvas(Canvas*)
void SetPainter(Painter*)
void SetGraphic(Graphic*)
Canvas* GetCanvas()
Painter* GetPainter()
Graphic* GetGraphic()

Set and get attributes of the damage object.

SEE ALSO

libgraphic(3I)

NAME

Graphic − persistent structured graphics objects

SYNOPSIS

#include <InterViews/graphic.h>
#include <InterViews/Graphic/ellipses.h>
#include <InterViews/Graphic/instance.h>
#include <InterViews/Graphic/label.h>
#include <InterViews/Graphic/lines.h>
#include <InterViews/Graphic/picture.h>
#include <InterViews/Graphic/polygons.h>
#include <InterViews/Graphic/rasterrect.h>
#include <InterViews/Graphic/splines.h>
#include <InterViews/Graphic/stencil.h>

DESCRIPTION

This manual page describes the **Graphic** base class for structured graphics objects and the derived classes provided by the *libgraphic* library. See libgraphic(3I) for a general overview of the library.

STATE OPERATIONS

Graphic(Graphic∗ = nil)

Create a new graphic having a copy of the given graphic's state. All state values are set to nil (undefined) by default.

virtual Graphic∗ Copy()

Returns an exact duplicate of the graphic. This function should be redefined in user-derived graphic classes to guarantee these semantics.

void SetBrush(PBrush∗)
PBrush∗ GetBrush()

Set or return the graphic's brush.

void SetColors(PColor∗ fg, PColor∗ bg)
PColor∗ GetFgColor()
PColor∗ GetBgColor()

Set or return the graphic's colors.

void SetFont(PFont∗)
PFont∗ GetFont()

Set or return the graphic's label font.

void SetPattern(PPattern∗)
PPattern∗ GetPattern()
void FillBg(boolean mode)
boolean BgFilled()

Set or return the graphic's fill pattern and mode. If the mode is *true*, fill operations will set pixels corresponding to 1's in the current fill pattern to the foreground color and pixels corresponding to 0's to the background color. If *false*, then only foreground pixels will be set. The default mode is *true*.

Graphic∗ Parent()
virtual boolean HasChildren()

Parent returns the graphic's parent, if any. HasChildren returns *false* by default; it should be redefined if a subclass should provide this information.

void SetTag(Ref)
Ref GetTag()

Set or return a tag associated with the graphic.

virtual Graphic& operator = (Graphic&)

The assignment operator is overloaded to copy the passed graphic's graphic state into this object's.

void Translate(float dx, float dy)
void Rotate(float angle, float cx = 0.0, float cy = 0.0)
void Scale(float x, float y, float cx = 0.0, float cy = 0.0)
void SetTransformer(Transformer*)
Transformer* GetTransformer()
void TotalTransformation(Transformer&)

Coordinates passed to drawing operations are transformed according to the current translation (cumulative), rotation, and scale factor. Optionally, scaling and rotation can be performed relative to a point (*cx*, *cy*). 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. TotalTransformation postmultiplies the given transformer by the total transformation incurred by the graphic's parents, if any.

Unless otherwise noted, input and output coordinates are relative to the coordinate system defined by the graphic's transformer.

virtual void GetBounds(float& left, float& bottom, float& right, float& top)

Return the exact coordinates of the smallest box circumscribing the graphic. The return values are in absolute coordinates.

void GetBox(Coord& left, Coord& bottom, Coord& right, Coord& top)
void GetBox(BoxObj&)

Return the smallest box circumscribing the graphic, in Coords. The return values are in absolute coordinates. A **BoxObj** represents a rectangle defined by lower left and upper right coordinates. **BoxObj** is defined in <InterViews/Graphic/geomobjs.h>.

virtual void GetCenter(float& x, float& y)

Return the center point of the graphic in absolute coordinates.

virtual boolean Contains(PointObj&)
virtual boolean Intersects(BoxObj&)

Return whether or not the graphic contains the given point or intersects the given rectangular area specified in absolute coordinates. **PointObj**, defined in <InterViews/Graphic/geomobjs.h>, describes a point.

void Align(Alignment, Graphic*, Alignment)

Position the given graphic relative to *this*. The graphic that performs the operation stays fixed, while the graphic supplied as the argument is translated appropriately. The first Alignment parameter specifies the alignment with respect to *this*, while the second specifies the alignment with respect to the given graphic.

DRAWING OPERATIONS

virtual void Draw(Canvas*)
virtual void Erase(Canvas*)

Draw draws the graphic on the given canvas using its current state. Erase erases the graphic by temporarily setting its foreground color equal to its background color and then drawing it.

virtual void Draw(Canvas*, Coord left, Coord bottom, Coord right, Coord top)
virtual void Erase(Canvas*, Coord left, Coord bottom, Coord right, Coord top)

Draw or erase the graphic on the canvas within the given bounding box, specified in absolute coordinates, without using fine clipping. The bounding box should be used simply as a hint about the graphic's visibility.

virtual void DrawClipped(Canvas*, Coord left, Coord bottom, Coord right, Coord top)
virtual void EraseClipped(Canvas*, Coord left, Coord bottom, Coord right, Coord top)

Draw or erase the graphic on the canvas strictly within the given bounding box, specified in absolute coordinates.

DERIVED CLASSES AND OPERATIONS

The following derived class are predefined. In addition to the inherited operations, each derived class defines a **GetOriginal(...)** operation that returns the values supplied to the constructor when the object was created.

Point(Coord x, Coord y, Graphic* graphic = nil)
Line(Coord x0, Coord y0, Coord x1, Coord y1, Graphic* graphic = nil)
MultiLine(Coord* x, Coord* y, int n, Graphic* graphic = nil)

Define a point, line, or set of connected lines at the specified coordinates, optionally having the same state as the given graphic. MultiLine's constructor copies the arrays of Coords that define its vertices. MultiLine provides two GetOriginal operations: one returns copies of the stored arrays, while the other the returns constant pointers to the arrays. MultiLine also defines == and != operators that compare the vertices of one instance against another's; two MultiLine instances are considered equivalent if they share the same vertices.

Ellipse(Coord x0, Coord y0, int r1, int r2, Graphic* graphic = nil)
FillEllipse(Coord x0, Coord y0, int r1, int r2, Graphic* graphic = nil)

Define an ellipse or filled ellipse with center at (*x0*,*y0*) and radii *r1* and *r2*.

Circle(Coord x0, Coord y0, int radius, Graphic* graphic = nil)
FillCircle(Coord x0, Coord y0, int radius, Graphic* graphic = nil)

Define a circle or filled circle with center at (*x0*,*y0*) and radius *radius*.

Rect(Coord x0, Coord y0, Coord x1, Coord y1, Graphic* graphic = nil)
FillRect(Coord x0, Coord y0, Coord x1, Coord y1, Graphic* graphic = nil)

Define a rectangle or filled rectangle at the specified coordinates.

Polygon(Coord* x, Coord* y, int n, Graphic* graphic = nil)
FillPolygon(Coord* x, Coord* y, int n, Graphic* graphic = nil)

Define a polygon or filled polygon specified by the *n* vertices. The vertex semantics are the same as those for MultiLine.

BSpline(Coord* x, Coord* y, int n, Graphic* graphic = nil)
ClosedBSpline(Coord* x, Coord* y, int n, Graphic* graphic = nil)
FillBSpline(Coord* x, Coord* y, int n, Graphic* graphic = nil)

Define a B-spline specified by the *n* control vertices. If closed or filled, the last point is connected to the first point. The vertex semantics are the same as those for MultiLine.

Label(const char* string, Graphic* graphic = nil)
Label(const char* string, int n, Graphic* graphic = nil)

Define a string or substring of text. If background fill mode is on, then the characters are drawn in the foreground color and other pixels within character boundaries are set to the background color. If background fill mode is off, only the foreground pixels are set.

Stencil(Bitmap* image, Bitmap* mask = nil, Graphic* graphic = nil)

A stencil graphic is defined by an image Bitmap and an optional mask Bitmap. The image defines the bits that will be drawn in the current foreground color (set bits) and background color (clear bits), and the mask defines which parts of the image will be drawn. The stencil graphic adopts the width and height of the mask if there is one; otherwise, it adopts the width and height of the image.

RasterRect(Raster*, Graphic* graphic = nil)

A raster rect is a graphic that displays a color raster image defined by a Raster object. The raster rect adopts the width and height of the Raster.

Picture(Graphic* graphic = nil)
void Picture::Append(Graphic*)
void Picture::Prepend(Graphic*)

void Picture::InsertAfterCur(Graphic*)
void Picture::InsertBeforeCur(Graphic*)
void Picture::Remove(Graphic*)
void Picture::RemoveCur()
void Picture::SetCurrent(Graphic*)
Graphic* Picture::GetCurrent()
Graphic* Picture::First()
Graphic* Picture::Last()
Graphic* Picture::Next()
Graphic* Picture::Prev()
boolean Picture::IsEmpty()
boolean Picture::AtEnd()
Graphic* Picture::FirstGraphicContaining(PointObj&)
Graphic* Picture::LastGraphicContaining(PointObj&)
int Picture::GraphicsContaining(PointObj&, Graphic&)**
Graphic* Picture::FirstGraphicIntersecting(BoxObj&)
Graphic* Picture::LastGraphicIntersecting(BoxObj&)
int Picture::GraphicsIntersecting(BoxObj&, Graphic&)**
Graphic* Picture::FirstGraphicWithin(BoxObj&)
Graphic* Picture::LastGraphicWithin(BoxObj&)
int Picture::GraphicsWithin(BoxObj&, Graphic&)**
void Picture::Propagate()

A **Picture** is a graphic that contains other graphics (subgraphics). The subgraphics are deallocated when the picture is destroyed. A picture has its own state which is ''concatenated'' with the state of its subgraphics when the picture is drawn or erased. The semantics of concatentation are that subgraphics are drawn with the picture's state information substituted for their own, except for attributes that are undefined (nil) in the picture; the subgraphics' individual attributes are used for these values. Furthermore, each subgraphic's transformer is postmultiplied by the picture's transformer when the picture is drawn, so that subgraphics are drawn relative to the picture's coordinate system.

Several operations are defined on Picture to allow subgraphics to be added, removed, and manipulated. Append and Prepend allow subgraphics to be added to the beginning and end of the picture, respectively. Subgraphics are drawn starting from the beginning of the picture, so that the most recently appended graphic is drawn last. Each picture maintains a pointer to its ''current'' subgraphic which can be set explicitly using SetCurrent and retrieved with GetCurrent. InsertAfterCur and InsertBeforeCur insert a graphic after or before the current subgraphic, respectively. Remove and RemoveCur remove a subgraphic from the picture. First and Last return the first and last graphics in the picture. Next and Prev return the subgraphic following and preceding the current one, respectively, and set the current subgraphic to the value returned. IsEmpty returns whether or not the picture has any subgraphics, and AtEnd returns *true* if the current subgraphic pointer has been advanced past the last subgraphic.

FirstGraphicContaining, LastGraphicContaining, FirstGraphicIntersecting, LastGraphicIntersecting, FirstGraphicWithin, and LastGraphicWithin return the first or last graphic in the picture that contains a point, intersects a box, and falls completely within a box, respectively. GraphicsContaining, GraphicsIntersecting, and GraphicsWithin return as a side-effect an array containing those subgraphics that contain a point, intersect a box, and fall completely within a box, respectively. These functions return the size of the array.

Finally, Propagate modifies the graphics state of the picture's subgraphics by concatenating in place each subgraphic's state with the picture's own. This is useful when the subgraphics should retain their appearance after removal from the picture. The picture's transformer is set to nil following the propagation to avoid concatenating its coordinate transformation twice.

Instance(Graphic* obj = nil, Graphic* graphic = nil)

An **Instance** of a graphic consists of all the usual state plus a reference to another graphic specified by *obj*. When drawn or erased, the instance concatenates the state of the graphic it references with its own.

FILES

<InterViews/Graphic/ppaint.h>	persistent paint objects
<InterViews/Graphic/geomobjs.h>	persistent geometry objects

SEE ALSO

libInterViews(3I), libgraphic(3I), Bitmap(3I), Canvas(3I), Transformer(3I), Painter(3I), Persistent(3I), Raster(3I)

NAME

GraphicBlock − interactor containing a Graphic

SYNOPSIS

#include <InterViews/Graphic/grblock.h>

DESCRIPTION

A GraphicBlock is an interactor that contains a graphic. GraphicBlocks are useful for interactors whose appearance is most conveniently defined using structured graphics. The graphic will be drawn whenever the interactor's Draw member function is called.

The graphic can be aligned with respect to the GraphicBlock's canvas. When the canvas is first allocated, the graphic is positioned so that its bounding box is aligned with the canvas. On subsequent resizes, the portion of the graphic within the canvas is aligned. With the default alignment, for example, a graphic will be centered in the canvas initially. If the canvas is resized, then the part of the graphic in the center of the canvas prior to resizing will appear in the center of the canvas after resizing.

The graphic can be scrolled and zoomed as well using a perspective. Zooming is performed by changing either the curwidth/curheight or the width/height members of the GraphicBlock's perspective. For example, to enlarge the view of the graphic by a factor of two, call the Adjust member function with a perspective whose curwidth and curheight are half that of the GraphicBlock's or whose width and height are double that of the GraphicBlock's. Zooming can be limited to magnifications that are powers of two by specifying Binary for the Zooming parameter.

The natural size of a GraphicBlock depends on the size of the graphic it contains and on any padding (white space surrounding the graphic) specified. GraphicBlocks are infinitely stretchable horizontally and vertically by default.

PUBLIC OPERATIONS

GraphicBlock(Sensor*, Graphic*, Coord pad = 0, Alignment = Center, Zooming = Continuous)

Create a new GraphicBlock with the specified sensor, graphic, padding, alignment, and zooming.

virtual void Invert()

Invert displays the GraphicBlock with the graphic's foreground and background colors exchanged.

void Update()

The Update operation should be called after the graphic is changed.

Graphic* GetGraphic()

Get the graphic in the GraphicBlock.

float GetMagnification()
void SetMagnification(float)

Get and set the magnification applied to the graphic, relative to its magnification when the GraphicBlock was created.

PROTECTED OPERATIONS

virtual void Zoom(Perspective&)
virtual void Scroll(Perspective&)

These functions define how zooming and scrolling are inferred from the perspective that is supplied to the Adjust function. The default behaviors implement standard scroller-style semantics and zooming as described above.

virtual float LimitMagnification(float)

Limit the amount of magnification that can be applied to the graphic. This function is used as a filter to convert desired magnifications into legal ones. By default, the argument is returned; no limits are imposed.

void UpdatePerspective()

Changes the GraphicBlock's perspective to reflect the bounding box of the graphic and calls Update on the perspective. UpdatePerspective encapsulates this functionality to facilitate redefinition of the Update function in derived classes.

SEE ALSO

Interactor(2I), libInterViews(3I), libgraphic(3I), Scroller(3I)

NAME

Persistent − persistent object package for Graphic

SYNOPSIS

#include <InterViews/Graphic/persistent.h>

DESCRIPTION

Persistent is a class that implements a general persistent object. Persistent objects can exist in memory, on disk, or both; the implementation supports a transparent interface to objects regardless of their location. Thus persistent objects can exist beyond the lifetime of the programs that create and use them. Though Persistent was designed to support the persistent structured graphics provided by Graphic(3I), programmers can build their own persistent classes by deriving from Persistent. The file <InterViews/Graphic/persistent.h> need not be included if Graphic header files are already included; it should be included only when deriving new Persistent objects.

The following operations are defined on **Persistent** objects:

virtual ˜Persistent()
: Deleting a persistent object will flush it from the object manager cache.

virtual ClassId GetClassId()
: Every class derived from Persistent must have a globally unique identifier associated with it. This member function must be redefined in derived classes to return the identifier reserved for that class.

virtual boolean IsA(ClassId)
: Returns *true* if the persistent object is an instance of the class (or subclass of the class) specified by the given ClassId. This member function must be redefined in derived classes to return the appropriate value.

virtual Persistent* GetCluster()
: If a persistent object (the *parent* object) contains other persistent objects as part of its internal state, then the parent and the objects it contains can be *clustered* on disk, or written and read consecutively. This can improve performance when the parent object is read since the objects within it are not scattered throughout the disk. GetCluster is called internally to determine whether the object is clustered within a parent or not. GetCluster returns *this* by default, meaning that the object is not to be clustered within another. If GetCluster is redefined to return nil, then the object is assumed to be clustered within a parent and will be written and read automatically whenever the parent is written or read.

virtual boolean write(File* f)
virtual boolean read(File* f)
: These functions define what information is written and read when the persistent object is transferred to and from disk. The argument is a class (defined in <InterViews/Graphic/file.h>) that encapsulates stdio file operations. *ParentClass*::read(f) must be called in the read function before anything else is read. Similarly, *ParentClass*::write(f) must be called in the write function before anything else is written. Write and read operations typically write and read ASCII and/or binary data and/or references to other persistent objects (provided by the **Ref** class, described below). Write and read return *false* if the operation was unsuccessful for any reason; otherwise they return *true*. These functions should not perform any operation (such as File::SeekTo and the like, or dereferencing a **Ref**) that changes the position of the next input or output operation on the File.

virtual boolean writeObjects(File* f)
virtual boolean readObjects(File* f)
: These functions define what clustered persistent objects should be written and read along with instances of the class. *ParentClass*::readObjects(f) *must* be called in the readObjects function before anything else is read. Similarly, *ParentClass*::writeObjects(f) must be called in the writeObjects function before anything else is written. WriteObjects and readObjects operations

typically perform WriteObjects and ReadObjects operations on references to other persistent objects that were written/read in the write and read functions. WriteObjects and readObjects return *false* if the operation was unsuccessful for any reason, otherwise they return *true*. These functions should not perform any operation that changes the position of the next input or output operation on the File.

virtual boolean initialize()

Initialize can be used to initialize an object's internal state with pointers to its clustered objects. Dereferencing of **Ref**s to clustered objects should only take place in this function, *not* in read or readObjects. *ParentClass*::initialize() should be called in this function before anything else is done.

virtual boolean Save()

Explicitly saves the object. Persistent objects that are part of a cluster should not be saved explicitly since they are automatically written and read with the parent object.

virtual boolean IsDirty()
virtual void Touch()
virtual void Clean()

A dirty bit is maintained for each persistent object. IsDirty returns the status of the dirty bit, Touch sets the dirty bit, and Clean resets it. Persistent objects are automatically touched when they are created.

REFERENCES TO PERSISTENT OBJECTS

The **Ref** class provides a mechanism for referring to a persistent object in a manner analogous to pointer references to conventional objects. Refs are dereferenced to yield the object to which they refer. They can be compared with each other, tested for validity, written to and read from disk. Each Ref has a unique universal identifier (UID) that serves as the object's ''address''. **Ref** is defined in <InterViews/Graphic/ref.h>.

Ref()

Ref(UID)
Ref(Persistent*)

A parameterless constructor creates a nil (undefined) reference. The constructor with a UID argument creates a reference with a particular UID; this constructor should not be used unless one knows what he or she is doing. The third constructor is the one most commonly used. It creates a reference to the given object.

Persistent* operator()()

Parenthesis are overloaded to perform the dereferencing operation on a Ref. The return value will most likely have to be cast into a pointer to the proper derived persistent class.

boolean operator==(Ref r)
boolean operator!=(Ref r)

Refs can be compared for equality and inequality using these overloaded operators.

boolean Write(File*)
boolean Read(File*)

Write or read a reference to an object. Theses functions return *false* if the operation is unsuccessful for any reason; otherwise *true* is returned.

boolean WriteObjects(File*)
boolean ReadObjects(File*)

Write or read consecutively the object(s) referenced by the Ref. If the referenced object is the head of a cluster, then these operations are nop's. These functions return *false* if the operation is unsuccessful for any reason; otherwise *true* is returned.

A **RefList** is class derived from **Ref** used to form circular doubly-linked lists of references to persistent objects. Each RefList object references a persistent object and contains pointers to the RefLists preceding and following it in the list. The **RefList** class is defined in <InterViews/Graphic/reflist.h>.

RefList()
RefList(UID u)
RefList(Persistent*)

RefList constructors perform the same functions as their **Ref** counterparts.

void SetRef(Ref r)

Explicitly set the reference to the given one.

boolean IsEmpty()

Returns *true* if the RefList has no neighbors, *false* otherwise.

void Append(RefList*)
void Prepend(RefList*)

Append inserts the given RefList after *this*; prepend inserts it before *this*.

void Remove(RefList*)
void Delete(Ref)
RefList* Find(Ref)

Remove finds the given RefList and removes it from the list. Delete finds the given Ref and deletes the RefList containing it. Find returns the RefList containing the given Ref.

RefList* First()
RefList* Last()
RefList* End()
RefList* Next()
RefList* Prev()

These functions return a particular element of the list relative to the position of this RefList in the list. First returns a pointer to the RefList following *this*, Last returns a pointer to the last RefList starting from *this*, and End returns a pointer to the RefList marking the end of the list. Next and Prev return pointers to the RefLists following and preceding *this* in the list.

boolean Write(File*)
boolean Read(File*)
boolean WriteObjects(File*)
boolean ReadObjects(File*)

These functions behave just as their **Ref** counterparts do, except they perform their operations on all elements in the RefList.

RefList* operator[](int n)

Returns a pointer to the *n*th element in the RefList, returning *this* if *n* = 0, *this*->Next() for *n* = 1, etc.

OBJECT MANAGER CLASS AND OPERATIONS

The **ObjectMan** class implements the *object manager*, a one-of-a-kind global object responsible for reading and writing persistent objects to and from disk. The object manager also serves as a well-known entity from which programs can retrieve the persistent objects they have created. **ObjectMan** is derived from **Persistent** and so is itself persistent. There is exactly one object manager per group of related persistent objects; no references can exist between persistent objects created by different object managers (different in the sense that they use different files for storage). Most of the member functions defined in **ObjectMan** need never be used by programs unless they derive new persistent classes.

ObjectMan(char* filename, void (*userInit)(RefList*) = nil, Persistent* (*userCreator)(ClassId) = nil)

The filename parameter requires a null-terminated string which will be used to prefix the names of two files that will contain all persistent object information: *filename*.map and *filename*.sto. If these files do not already exist when the constructor is called, then they will be created and the function pointed to by *userInit* will be called. *userInit* must point to a function that creates primordial root objects needed by your application and installs them in the passed **RefList**. If you want to use the default persistent paint objects in Graphic, then this function should call InitPPaint, defined in

<InterViews/Graphic/ppaint.h>. *userCreator* must point to a function that can create an instance of any persist object used in the application, given a ClassId. *userCreator* should call GraphicConstruct, defined in <InterViews/Graphic/grconstruct.h>, if it receives an unrecognized ClassId. If your application does not derive any new persistent objects, *userCreator* can simply point to GraphicConstruct. A newly allocated object manager should be assigned to the global variable *TheManager*. Further references to the object manager should be made via this global.

RefList* GetRoot()

Returns a pointer to the RefList containing well-known objects. Persist objects that are to be accessed directly should have references to them appended to the root RefList in a well-known order for later retrieval. For example, the default persistent paint objects defined in InitPPaint should be appended to the root RefList so that they can be accessed when the program is restarted.

Persistent* Create(ClassId)

Creates a persistent object of class specified by *ClassId*. Create calls the function pointed to by userCreator.

boolean Invalidate(Persistent*)

Used to permanently deallocate a persistent object. Returns *false* if the operation was unsuccessful, otherwise returns *true*.

boolean IsCached(Ref*)

Returns *true* if the referenced object has been read from disk into memory, *false* if not. IsCached updates the reference to reflect the address of the object if it is in memory.

boolean Find(Ref*)

Seeks for and reads the referenced object from disk if it is not in memory and updates the reference to reflect the memory address of the object. Find returns *true* if the operation was successful.

boolean Retrieve(Ref*)

Like Find except does not perform a seek; the object is read blindly from disk. Retrieve is used to read clustered objects.

boolean Store(Persistent*)
boolean Update(Persistent*)

Store allocates new disk space for an object and writes it to disk. Update rewrites an existing object and updates the map containing its disk offset. Both return *true* if the operation was successful.

boolean ObjectIsDirty(Persistent*)
void ObjectTouch(Persistent*)
void ObjectClean(Persistent*)

The object manager maintains a dirty bit for each persistent object in its object cache. These functions allow the dirty bit for an object to be tested, set and reset. The dirty bit operations defined in **Persistent** actually call these functions through the global variable TheManager. Dirty objects are automatically written to disk when TheManager is deleted.

SEE ALSO

libgraphic(3I)