# InterViews Reference Manual
# Version 3.1-Beta

*June 26, 1992*

Mark A. Linton
Paul R. Calder
John A. Interrante
Steven Tang
John M. Vlissides

# Release Notes

The InterViews 3.1 distribution contains a README file and a single subdirectory, "iv", that contains the source and documentation. InterViews can be compiled with any C++ compiler that accepts the 2.0, 2.1, or 3.0 revisions of the language, and can run on X11R4 or X11R5. You can specify your compiler and other site definitions in the file "iv/src/config/InterViews/local.def".

The README file describes how to build InterViews. Under "iv/src", the directory "include" contains include directories, "bin" contains applications (each in its own subdirectory), "lib" contains libraries (each in its own subdirectory), "config" contains configuration-specific files, "man" contains PostScript or troff for the manual, and "papers" contains PostScript for user tutorials.

The work at Stanford has been supported by Fujitsu America, Digital Equipment Corporation, and NASA CASIS project under Contract NAGW 419, and a grant from the Charles Lee Powell Foundation. Special thanks to Ira Machefsky of Digital and Charles Brauer of Fujitsu America for their assistance. We are also grateful to the ever-growing InterViews user community for its encouragement and support.

Please address questions or comments about InterViews to

Mark Linton
linton@sgi.com

# Differences between 3.0 and 3.1

InterViews 3.1 contains several improvements over 3.0.1. WidgetKit is an improved implementation of the Kit class that was in 3.0.1. DialogKit is a class that provides a file chooser dialog using WidgetKit compoments. LayoutKit is a class that provides convenient operations for creating layout objects such as boxes and glue, replacing the many independent classes such as LRBox and VGlue in 3.0.1. Chapter 9 of the reference manual describes WidgetKit, Chapter 10 describes DialogKit, and Chapter 11 describes LayoutKit.

Glyph and other subclasses of Resource are no longer derived as virtual base classes. This change was made primarily for efficiency and convenience because many C++ compilers do not generate particularly efficient code for virtual base classes (especially in space), and some compilers have bugs in this area.

The Glyph protocol has been extended with an *undraw* operation that notifies a glyph that it no longer has an allocation on the canvas. This operation is primarily for objects that cache information or perform computation when they are visible.

A new monoglyph subclass, called InputHandler, replaces the old Listener class. InputHandler is much simpler to use than the previous combination of Listener and PointerHandler, as well as providing limited support for focus management.

The Window class now has a style object as one of its attributes. The style can be set to defined attributes such as name and geometry, as well as a “visual” type for the window. On systems that support overlay planes, the attribute “overlay” can be used to request the window be allocated in the window planes.

InterViews 3.1 also includes a new faster implementation of the Style class and a copy of the latest version of Sam Leffler’s TIFF library (v3.0) for reading images. The old 2.6 structured graphics library is no longer included, though it probably would still work with this distribution.

The documentation has been reorganized to reflect some of the 3.1 changes, as well as provide a structure for future releases. In addition to the reference manual, a new collection of example programs is provided under iv/src/examples.

# Chapter 1

# Introduction

InterViews is a software system for window-based applications. Like most user environments, InterViews is *object-oriented* in that components such as windows, buttons, menus, and documents are active elements with inherited behavior. The name “InterViews” comes from the idea of a user interface object presenting an *inter*active *view* of some data. For example, a text editor implements an interactive view of the contents of a text file.

InterViews provides a set of classes that define the behavior of user interface objects. We distinguish InterViews classes into two groups: *protocols* and *kits*. A protocol defines the set of operations that an object can perform, such as drawing or handling input. A kit defines a set of operations for creating other objects. The idea of a kit is also sometimes referred to as an “object factory”. Using kits hides the details of object construction and subclassing-instancing tradeoffs made by the implementation, as well as providing a higher-level organizational structure to the system.

## 1.1 Organization

In this first chapter, we define the basic notation and classes, as well as give an overview of the system by way of several example programs. Chapter 2 defines the base protocol for user interface objects, called *Glyph*, which supports geometry management, rendering, picking, and structuring multiple glyphs into an aggregate. Chapter 3 describes the input event processing model and the *InputHandler* protocol. InputHandler is a descendant of Glyph that receives input events. Chapter 4 presents the *View* protocol, which is derived from InputHandler and adds additional operations for creating and updating multiple views of a shared data object.

Chapter 5 defines the *Window* protocol for associating glyphs with a window on the screen and communicating with a window manager. Chapter 6 presents the basic protocols for rendering to the screen or a printer.

Chapter 7 is reserved for the future to describe the *FigureKit* class that will create common 2D graphic objects, such as rectangles, circles, and polygons. In traditional graphics terminology, FigureKit will support “structured” graphics, while the basic rendering protocols described in Chapter 6 support “immediate-mode” graphics.

Chapter 8 defines the *Style* protocol, which manages a collection of named attributes with string values. Chapter 9 presents the *WidgetKit* class for creating common user interface components such as buttons, menus, and scrollbars. The precise appearance and input behavior of a widget can be adjusted by the attributes in a style. Chapter 10 presents the *DialogKit* class for building common dialogs.

Chapter 11 defines the *LayoutKit* class for creating glyphs that control formatting. These objects are based on the TeX document preparation system. Chapter 12 describes the *DocumentKit* class for creating and editing documents.

Appendix A defines classes for interfacing to the underlying operating system. These classes are not intended to provide a complete or standard interface to the operating system, but merely a more convenient and portable set of protocols.

# 1.2 Notation

*N.B.: The notation in this manual is somewhere between C++ and the OMG Interface Definition Language (IDL). We expect to use IDL in the future.*

We use a syntax similar to C++ to specify the InterViews classes and operations. However, the specification of a class here is *not* identical to its C++ declaration in a header file. To make the distinction clear between our specification and C++, we use the keyword “interface’ instead of “class”. We assume inheritance is always “public” in C++ terminology.

All operations are assumed public; we do not list the protected or private members. We also do not list members that are implicitly part of the implementation. For example, C++ destructors are normally public but often simply free storage allocated by the object. Thus, there is no need to document destructors as part of a class interface.

Unless explicitly specified as “static”, all operations are virtual functions in C++. In the case of an inherited operation, the choice of whether or not to provide the operation may depend on the implementation. We therefore do not list inherited operations unless the subclass extends the semantics in some way.

## 1.2.1 Names

We use identifiers that begin with an upper case letter for types; we use lower case for operations and parameters. In type names consisting of multiple words, we capitalize the beginning of each word, as in FirstSecondThird. For operations or parameters we use underscores to separate words, as in first_second_third. An operation *f* for a class *C* is denoted by *C*::*f*.

We assume that some mechanism allows us to use whatever names we wish for global symbols; that is, there is no need for a special prefix on all class and type names. Ideally, this capability would be provided by the implementation language, but this is not yet the case for C++. Currently, the implementation uses #define macros to prepend the prefix ‘‘iv’’ to all class names and global type names. This redefinition is hidden from the programmer, except inasmuch as it shows up during debugging. To undefine these macros, a source file must include “<InterViews/leave-scope.h>”. After leaving the InterViews scope, InterViews names are specified by “_lib_iv(*name*)”, where *name* is the class or type name defined in this manual.

### 1.2.2 Use of const

C++ allows the type modifier "const" to be used in a variety of different ways. For concrete objects (int, float, char*), it refers to read-only storage. For abstract objects, however, storage access should not be visible in an interface. For example, an operation on a transformation matrix could be defined that returns whether the matrix is the identity or not. A simple implementation could compute the identity test and could be defined as const in the storage sense. However, another implementation might cache the result of the test to avoid the overhead of the test when the matrix is not changing. This second implementation is not const in the storage sense because it modifies the transformation matrix object.

We use const for operations that do not change the *behavior* of an object. That is, an operation is const if a call to it could be omitted without changing the effect of subsequent operations performed on the object. This definition is consistent with the notion that a compiler could eliminate redundant calls to the same const function. The one important counter-example is reference counting, where incrementing and decrementing the reference count of a shared object changes its lifetime (an operation that could not be eliminated by the compiler) but does not change its behavior (the operation is considered const).

Using our semantics of const, the transformation matrix identity function mentioned above should be defined as const. This approach implies that an implementation may be forced to cast the *this* pointer from const to non-const to avoid a compiler error message because C++ compilers normally assume that a function should not be const if the function does modifies storage.

It is also possible (indeed, likely) that a function does not follow our semantics of const even though it does not modify the object's storage. If an object contains a pointer to another object and an operation performs a non-const operation on that object, then the first operation may also need to be defined as non-const.

### 1.2.3 Common Symbols

Several definitions are pervasive throughout the system. These definitions are automatically defined as a side effect of using any other InterViews classes. The type *boolean* is defined in the conventional sense of a language such as Pascal, as are the constants *true* and *false*. The constant *nil* is presumed to be type-equivalent to any pointer type and represents an invalid pointer value. In C++, we #define nil to zero.

The type *String* is used as a parameter to a number of operations, though no specific interface or implementation is presumed. Whereever a string parameter appears, one can assume that a C++ "const char*" type can also be passed.

### 1.2.4 Coordinates

Many objects and operations manipulate coordinates. We define the type *Coord* to represent a position or a distance. The default units for a coordinate are "printers points”, or 1/72 of an inch. The scaling factor can be customized on a per-screen basis.

Ideally, Coord would be an abstract type with set of operations and conversions to concrete types. However, current C++ compilers do not make it practical to define a Coord class that is represented as a single word. The current implementation therefore defines Coord as the C++ type “float’.

For applications with simple graphics needs, the use of floating point coordinates is typically not a problem. Applications with more sophisticated text or graphics usually need to manipulate floating point coordinates anyway.

The use of non-pixel units allows objects to be resolution-independent, but also means that one cannot rely on precise output at low resolutions. Coordinates are rounded-off to guarantee that objects that abut in coordinates will abut on the screen. This choice means that a one point line might generate one or two pixels, depending on where it is on the screen. Objects that need to generate consistent pixel sizes can explicitly round to whole-pixel coordinate values using Canvas to_pixels_coord.

### 1.2.5 Dimensions

Many composition strategies manipulate coordinates in one dimension. To use a single strategy in any dimension, we define the type *DimensionName* and values *Dimension_X*, *Dimension_Y*, or *Dimension_Z*. It is possible that other dimensions may be defined in the future. The constant *Dimension_Undefined* is defined after any other dimensions.

### 1.2.6 Parameter conventions

In specifying an operation, we list only the parameter type unless there are several parameters to the operation with the same type. In this case, we list both the type and a name for the parameter that clarifies what it is.

For parameters that are objects, we use a pointer type if the operation may store a pointer to the object with a lifetime beyond the operation’s activation. If the operation will not store a pointer, then we pass a reference type. Therefore, one should never pass the expression “&x” for a local or parameter object “x”.

### 1.2.7 Storage management

Because C++ does not provide garbage collection, it is necessary to manage shared objects within a program. We make objects easier to share by deriving from class Resource, which manages a reference count. If we had garbage collection, we would not need the resource class. Therefore, we do not consider resource really part of the InterViews programming interface and it does not appear in any function type signatures. However, it is necessary to understand which objects can be shared and as a practical matter the implementation must manage references correctly.

Figure 1.1 shows the Resource class interface. Every resource has a reference

```
interface Resource {
   static void ref(const Resource*);
   static void unref(const Resource*);
   static void unref_deferred(const Resource*);
   static void flush();
   void cleanup();
};
```

Figure 1.1: Resource class interface

count that is initially zero and must be explicitly incremented by calling Resource::ref. The reason the count is initially zero is that an object is often created and immediately passed as a parameter to another object that stores the reference. Since the receiver must reference the parameter anyway, it is confusing to force the creator to unreference the resource after passing it.

The C++ delete operator should not be used on resources directly; instead, Resource::unref or Resource::unref_deferred should be called to decrement the reference count of an object. If the count is no longer positive, then the resource's cleanup operation will be called. In the case of Resource::unref, the object is immediately destroyed. In the case of Resource::unref_deferred, the object will be put on a queue of objects to be destroyed in order the next time Resource::flush is called. Resource deferral is useful when an object initiates a delete on one of its ancestors or some other object with an active member function.

### 1.2.8 Callbacks

C++ provides a pointer-to-function type, but often one wishes to encapsulate an object and a pointer to a member function to call on the object. Our approach is to define a base class containing the callback signature and a parameterized subclass for a callback to an object of a specific type. For example, the *action* class defines a single operation, Action::execute, with no parameters and no return value. A *macro* is an action that contains a list of actions, each of which is executed in order. An *action callback* is the subclass that is expanded for each destination type. The current implementation uses preprocessor macros, but will use templates in the future. Figure 1.2 shows the action, macro, and action-callback class interfaces.

## 1.3 Basic concepts

The goal of InterViews is to make it easy to compose user interfaces from reusable components. The central class for physical composition is Glyph because it defines the geometry of a user interface object. The central class for logical composition is InputHandler because it defines the input handling policy and update management. InputHandler is a subclass of glyph, normally delegating its

```
typedef long MacroIndex;

interface Action : Resource {
  void execute() = 0;
};

interface Macro : Action {
  Macro(Action* = nil, Action* = nil, Action* = nil, Action* = nil);
  void prepend(Action*);
  void append(Action*);
  void insert(MacroIndex, Action*);
  void remove(MacroIndex);
  MacroIndex count() const;
  Action* action(MacroIndex) const;
};

interface ActionCallback(T) : Action {
  ActionCallback(T)(T*, void (T::*)());
};
```

Figure 1.2: Action and related class interfaces.

geometry and appearance to another glyph.

The Canvas class defines a 2-dimensional surface upon to which a group of glyphs are attached. The glyphs negotiate for space on the canvas, draw on the canvas to refresh their appearance, and damage the canvas to cause an update traversal.

The Window class creates a canvas for a top-level input handler and allows the user to control the canvas on a screen through a window manager. The window class does not define a window management policy, only the mechanism for communicating appropriate information between a window manager and the glyphs.

The InterViews input model is ‘‘top-down’’. That is, an input event is first received by the appropriate window as determined by the window manager. The receiving window passes the event to the root input handler, which may pass it down to a nested input handler.

### 1.3.1 Main event loop

Every application creates a Session object to run an event dispatching loop. Sessions initially attach to the user’s display and can attach to additional displays. A session defines a root Style object derived from user customization files and command-line arguments. Styles define named attributes with string values, wildcarding for matching classes of objects, and quick access to common attributes such as fonts and colors. Figure 1.3 shows the session protocol.

The session is given an alias (class name) for the root style, an array of command-line arguments, an optional argument description, and optional initial style settings. The root style’s name comes from the “-name” command-line option (if given), or the environment variable RESOURCE_NAME (if defined), or the value of argv[0] with leading path entries stripped.

```
struct PropertyData {
  const char* path;
  const char* value;
};

struct OptionDesc {
  const char* name;
  const char* path;
  OptionStyle format;
  const char* value;
};

interface Session {
  Session(
    const char* name, int& argc, char** argv
    const OptionDesc* = nil, const PropertyData* = nil
  );
  static Session* instance();
  int argc() const;
  char** argv() const;
  Style* style() const;
  void default_display(Display*);
  Display* default_display() const;
  Display* connect(const String&);
  Display* connect(const char*);
  void disconnect(Display*);
  int run();
  int run_window(Window*);
  void quit();
  boolean done();
};
```

Figure 1.3: Session protocol

Figure 1.4:

The option description is an array of structures that describe command-line arguments. The structure contains four fields: a *name* that is the command-line string, a *path* specifying a style attribute, a *format* specifying where the associated value is, and a default *value*. Valid formats are OptionPropertyNext (use the next argument as an attribute-value pair), OptionValueNext (use the next argument as the value), OptionValueImplicit (use the default value), OptionValueIsArg (use the agument as the value), and OptionValueAfter (use the remainder of the argument as the value). If a command-line argument is found that matches an option description, it will be interpreted and removed from argv. The agument count (argc) will be set to the number of uninterpreted command-line arguments.

In addition to the program-specified options, the Session constructor automatically matches the options listed in Table 1.1. The optional initial style settings are specified by a nil-terminated array of pairs <*s1*,*s2*>, where *s1* is an attribute name and *s2* is the value. User defaults will override any initial settings,

| | |
|---|---|
| -background | next argument sets the background color |
| -bg | same as -background |
| -dbuf | double-buffer windows by default |
| -display | next argument specifies the target workstation display |
| -dpi | next argument is coordinates-to-pixels ratio |
| -fg | same as -foreground |
| -flat | next argument sets the base color for bevels |
| -fn | same as -font |
| -font | next argument sets the default text font |
| -foreground | next argument sets the foreground color |
| -geometry | next argument sets the first window's position and size |
| -iconic | starts up first window in iconic form |
| -malloc | run with memory-management debugging on |
| -monochrome | use monochrome style |
| -motif | use Motif style |
| -name | next argument sets the instance name of the session |
| -nodbuf | do not double-buffer windows |
| -openlook | use OpenLook style |
| -synchronous | force synchronous operation with the window system |
| -title | next argument sets the session's default title bar name |
| -visual | next argument is visual class to use for windows |
| -visual_id | next argument is visual id number to use for windows |
| -xrm | next argument is "name:value"; sets named attribute |

Table 1.1: Predefined command-line options

and command-line arguments will override user defaults.

### 1.3.2 Common widgets

InterViews provides common behavior objects, such as buttons and menus, which are built using glyphs and input handlers. Unlike many other toolkits, InterViews objects are cheap enough that behavior objects are separate from appearance objects. For example, the button class is given a separate glyph that denotes its appearance; the button only implements input behavior.

This approach of separating input and output objects makes the toolkit more flexible and the individual objects simpler, but it can make the task of constructing higher-level user interface objects more confusing. In particular, it becomes less obvious whether it is appropriate to subclass or instance to create a particular component.

InterViews provides a widget kit object that encapsulates these decisions in a single programming interface while allowing various appearances and behaviors. A widget kit is an object that creates common user interface objects such as buttons, scrollbars, and menus. For example, a push button has button behavior, a beveled or highlighted appearance when pressed, and a style for customization. The WidgetKit class provides a function to create a push button object; the implementation is free to compose what objects are necessary to provide the appropriate functionality. Each application normally creates a single WidgetKit object, which is typically a subclass defined for a particular look-and-feel such as OpenLook or Motif. The application can call the static function WidgetKit:::instance to return the object and therefore be unaware which specific look-and-feel is being used.

## 1.4 A simple example

Figure 1.5 shows a simple InterViews application that display the text "hi mom!" in a window. The application creates a session and runs it starting with a window containing the text over a background. The window is an "application window", meaning it is recognized by other desktop services (window manager, session manager) as the main window for the application.

The window contains a background that contains a label. A background glyph paints a given color behind its contents. The label's font and color are obtained from the default style for the session.

This application does not handle user input. It must therefore be terminated externally, either from a window manager menu or through the system.

## 1.5 Geometry management

See iv/src/examples/box[12].

```
#include <IV-look/kit.h>
#include <InterViews/background.h>
#include <InterViews/session.h>
#include <InterViews/window.h>

int main(int argc, char** argv) {
   Session* session = new Session("Himom", argc, argv);
   WidgetKit& kit = *WidgetKit::instance();
   return session->run_window(
      new ApplicationWindow(
         new Background(
            kit.label("hi mom!"), kit.background()
         )
      )
   );
}
```

Figure 1.5: InterViews "hi mom!" program.

## 1.6 Creating a push button

See iv/src/examples/button[123].

## 1.7 Summary

This chapter has presented a broad overview of the InterViews architecture, introducing the basic concepts and giving some simple examples. In the next chapter, we focus on the semantics of glyphs as the smallest unit of user interface construction.

# Chapter 2
# Glyphs

Glyphs are the basic building blocks for the presentation side of a user interface, providing a unified substrate for interactive objects, structured graphics, and formatted text. The glyph protocol supports geometry, drawing, picking, composition, and structure. Glyph subclasses provide higher-level operations, such as input handling and update management.

The base class defines no storage and operations are passed contextual information during rendering. Thus, glyphs may be shared and a glyph structure need not be strictly hierarchical; it may be a directed acyclic graph. Figure 2.1 shows the glyph protocol.

## 2.1 Geometry management

Glyph::request asks a glyph to specify its desired geometry. This operation should be fast to allow a composite glyph to compute rather than store the desired geometry of its components. Glyph::allocate notifies a glyph that a portion of the canvas has been allocated for it and returns an area that represents a conservative estimate of the actual area where the glyph will draw.

## 2.2 Drawing

Glyph::draw displays the glyph on the canvas at the position and size specified by the allocation. No clipping or translation is implicit in this operation–the canvas is the entire drawing area and the allocation is in canvas coordinates. However, a composite glyph may set clipping on the canvas before drawing its children.

There is no guarantee that a call to draw will affect the screen immediately because of the possibility (indeed, likelihood) that the canvas is double-buffered. Furthermore, drawing order is important for glyphs when they are partially transparent and are drawn in the same plane. A glyph's draw operation therefore should not be called directly, but rather is implicitly called during screen update.

Glyph::undraw notifies a glyph that its allocation is no longer valid. This operation is generally a performance hint to free cached information associated with drawing or inhibit some operation that is unnecessary while the glyph is not visible, such as cursor blinking. Glyph::undraw is *not* to be used when a glyph's allocation changes; in that case, allocate and draw can simply be called with different parameters.

Glyph::print generates a representation of the glyph graph suitable for printing. The canvas and printer rendering interfaces are identical, and the default

```
typedef long GlyphIndex;
typedef unsigned int GlyphBreakType;

interface Glyph : Resource {
  void request(Requisition&) const;
  void allocate(Canvas*, const Allocation&, Extension&);
  void draw(Canvas*, const Allocation&) const;
  void undraw();
  void print(Printer*, const Allocation&) const;
  void pick(Canvas*, const Allocation&, int depth, Hit&);

  enum { no_break, pre_break, in_break, post_break };
  Glyph* compose(GlyphBreakType) const;

  void append(Glyph*);
  void prepend(Glyph*);
  void insert(GlyphIndex, Glyph*);
  void remove(GlyphIndex);
  void replace(GlyphIndex, Glyph*);
  void change(GlyphIndext);
  GlyphIndex count() const;
  Glyph* component(GlyphIndex) const;
  void allotment(GlyphIndext, DimensionName, Allotment&) const;
};
```

Figure 2.1: Glyph protocol

implementation of print is simply to call draw. Most glyphs therefore need not define a print operation. The reason for a distinct print operation is to allow a glyph to use different rendering requests for the screen and a printer. For example, a glyph might use 3D requests to the screen, or might compute more precise output for printing.

Glyph::pick finds the glyphs that intersect a point or rectangle specified in canvas-relative coordinates. Conceptually, picking is like drawing and determining what glyphs intersect the specified point or rectangle. The coordinates are contained in the hit parameter. The depth parameter specifies which level in the Hit object to store the intersecting glyphs. When pick returns, the Hit parameter contains the glyphs that were hit.

Figure 2.2 shows the hit protocol. A hit object may be constructed with a point, a rectangle, or an event. In the case of the event, the event pointer coordinates are used to detect intersection and glyphs can associate a handler with the pick result.

Hit::event, Hit::left, Hit::bottom, Hit::right, Hit::top return information about the specified hit area. Hit::event returns nil if the point or rectangle constructors were used.

Hit::push_transform, Hit::transform, and Hit::pop_transform modify the current intersection area for picking. These operations are just like the canvas operations with the same names except they apply to the hit information instead of a canvas.

Glyphs record information in a hit object with Hit::begin, Hit::target and

```
interface Hit {
  Hit(const Event*);
  Hit(Coord x, Coord y);
  Hit(Coord left, Coord bottom, Coord right, Coord top);

  const Event* event() const;
  Coord left() const, bottom() const, right() const, top() const;

  void push_transform();
  void transform(const Transformer&);
  void pop_transform();

  void begin(int depth, Glyph*, GlyphIndex, Handler* = nil);
  void target(int depth, Glyph*, GlyphIndex, Handler* = nil);
  void end();
  void remove(int depth, GlyphIndex target = 0);
  void retarget(
    int depth, Glyph*, GlyphIndex, Handler* = nil, GlyphIndex = 0
  );

  boolean any() const;
  int count() const;
  int depth(GlyphIndex target = 0) const;
  Glyph* target(int depth, GlyphIndex target = 0) const;
  GlyphIndex index(int depth, GlyphIndex = 0) const;
  Handler* handler() const;
};
```

Figure 2.2: Hit class interface.

Hit::end. Hit::target indicates that a glyph's output intersects the hit region. Hit::begin and Hit::end are used by composite glyphs that should be on the hit list if and only if one of their components calls Hit::target. The parameters to Hit::begin and Hit::target have the same meaning. The depth is the level in the hit list where the information should be stored. The glyph is the hit glyph. The index is additional information about the hit. For a composite glyph, this is typically the index of the hit child. Hit::remove and Hit::retarget modify the current hit information.

The remaining hit operations return information about a pick. The result is a list of paths, each of which contains a list of glyphs terminating with the glyphs that would draw through the pick region. Hit::count returns the number of paths. Hit::depth returns the index of the last glyph in a specified path. The path corresponding to the top-most glyph is in position zero. Hit::target and Hit::index return the information for a given path and depth.

If a pick is done on a hit object constructed from an event, and one or more glyphs find the event of interest, they will associate a handler with the hit object. Hit::handler returns the top-most, deepest handler, or nil if there is none.

## 2.3 Composition

BreakType defines the choices for how a composite glyph might break a group of glyphs in a layout. The break may occur before a glyph (pre_break), in the glyph (in_break), or after the glyph (post_break).

Glyph::compose returns a new glyph that should be used to replace the target glyph when the break occurs. For example, discretionary white space in a document will turn into zero-size glue if a line-break or page-break occurs on it.

## 2.4 Structure

Several operations are provided to manipulate the list of components that make up a composite glyph. Primitive glyphs ignore these operations, while monoglyphs pass the operation through to their body. A monoglyph is therefore ‘‘transparent’’ with respect to structure, allowing one to put a monoglyph around a composite and pass the result to another object that manipulates the components of the composite.

Glyph::append (prepend) adds a given glyph to the end (beginning) of the component list. Glyph::insert adds a given glyph at a specified index in the list. Glyph::remove removes the glyph at the specified index. Glyph::replace replaces the glyph at the specified index with the given glyph. Glyph::change notifies a glyph that its component at the given index has changed, and therefore it might be necessary to reallocate it.

Glyph::count returns the number of glyphs in its component list. Glyph::component returns the glyph in its component list at the given index. Glyph::allotment returns the allocation information in the given dimension for the glyph at the given index in its component list.

## 2.5 Requisitions

Glyphs define their geometric needs with a *requisition*, which specifies a *requirement* in each dimension. Figure 2.3 shows the requisition and requirement class interfaces.

Requisition::penalty is overloaded to set or get the penalty associated with choosing a break at the position defined by the requisition. Requisition::require sets a requirement in a given dimension. Requisition::requirement returns the requirement for a given dimension.

A requirement consists of a natural size, stretchability, shrinkability, and alignment. The maximum size is the natural size plus the stretchability; the minimum is the natural minus the shrinkability. It is possible to have negative minimum sizes. The alignment is a fraction that indicates the origin of the area. An alignment of zero means the origin is at the lower coordinate value; an

```
interface Requisition {
   void penalty(int);
   int penalty() const;
   boolean equals(const Requisition&, float epsilon) const;
   void require(DimensionName, const Requirement&);
   const Requirement& requirement(DimensionName) const;
   Requirement& requirement(DimensionName);
};

interface Requirement {
   Requirement(Coord natural);
   Requirement(Coord natural, Coord stretch, Coord shrink, float);
   Requirement(
      Coord natural_lead, Coord max_lead, Coord min_lead,
      Coord natural_trail, Coord max_trail, Coord min_trail
   );

   boolean equals(const Requirement&, float epsilon) const;
   boolean defined() const;
   void natural(Coord), stretch(Coord), shrink(Coord);
   Coord natural() const, stretch() const, shrink() const;
   void alignment(float);
   float alignment() const;
};
```

Figure 2.3: Requisition and Requirement classes.

alignment of one means it is at the upper coordinate value.

There are four constructors for requirement: with no parameters, which makes the natural size undefined and the other fields zero; with a single coordinate that defines the natural size and makes the other fields zero; with explicit coordinates for the natural size, stretchability, and shrinkability, and a float value for the alignment; and with explicit coordinates for specifying the distances on each side of the origin.

Requirement::equals compares two requirements (the target object and the argument) using a given tolerance for the individual coordinate comparisons. Requirement::defined returns whether the natural size has been defined. The overloaded functions natural, stretch, shrink, and alignment set and get the respective values.

## 2.6 Allocations

The actual size a glyph is given is defined by an *allocation*, which specifies an *allotment* in each dimension. Figure 2.4 shows the allocation and allotment class interfaces. An allotment specifies one dimension of an allocation with three values: an origin, a span, and an alignment. The origin is a position within the allotment and the span is the size of the allotment. The alignment is a fraction specifying the position of the origin. For example, if the origin is 1, span is 10,

```
interface Allocation {
   boolean equals(const Allocation&, float epsilon) const;
   void allot(DimensionName, const Allotment&);
   Allotment& allotment(DimensionName);
   const Allotment& allotment(DimensionName) const;
   Coord x() const, y() const;
   Coord left() const, right() const, bottom() const, top() const;
};

interface Allotment {
   Allotment(Coord origin, Coord span, float alignment);
   boolean equals(const Allotment&, float epsilon) const;
   void origin(Coord), offset(Coord), span(Coord);
   Coord origin() const, span() const;
   void alignment(float);
   float alignment() const;
   Coord begin() const;
   Coord end() const;
};
```

Figure 2.4: Allocation and allotment class interfaces

and alignment is 0.5, then the allotment begins at -4 and ends at 6.

The allocation constructor initializes each allotment to have zero values. Allocation::equals compares two allocations using a tolerance for individual coordinate comparisions. Allocation::allot sets the allotment for a given dimension. Allocation::allotment returns the allotment for a specified dimension.

Several operations are provided for convenience when accessing allotments in the X and Y dimensions. Allocation::x and Allocation::y return the X and Y origins. Allocation::left and Allocation::right return the X dimension end-points, Allocation::bottom and Allocation::top return the Y dimension end-points.

The Allotment class also provides an equals operation for comparing two allotments with a given tolerance. Allotment::origin, Allotment::span, and Allotment::alignment are overloaded names for setting and getting the allotment values. Allotment::offset adds to the current origin.

Allotment::begin and Allotment::end return the end-points of an allotment. Allotment::begin is equivalent to the origin minus the alignment times the span. Allotment::end is equivalent to Allotment::begin plus the span.

## 2.7 Extensions

The area that defines where a glyph actually draws is called an *extension*. This area is typically used for update. If a glyph needs to be completely redrawn, the glyph's extension must be damaged.

Extensions are represented in device-independent units, but must be rounded

```
interface Extension {
  void set(Canvas*, const Allocation&);
  void set_xy(Canvas*, Coord left, Coord bottom, Coord right, Coord top);
  void clear();
  void merge(const Extension&);
  void merge(Canvas*, const Allocation&);
  void merge_xy(
    Canvas*, Coord left, Coord bottom, Coord right, Coord top
  );
  Coord left() const, bottom() const, right() const, top() const;
};
```

Figure 2.5: Extension protocol

out to the nearest device-dependent units. For example, if one of the bounds for one glyph's extension is 10.2 and another is 10.5, we would need to redraw both glyphs if the bounds translate to the same pixel coordinate. Because extensions are typically used to damage a canvas, the extension coordinates are canvas-relative.

A glyph computes its extension as part of the allocate operation. Typically, a glyph will simply return its allocation rounded according to the canvas.

Figure 2.5 shows the Extension protocol. Extension::set initializes an extension to the given allocation transformed and rounded for the given canvas. Extension::clear sets an extension to be an empty area. Extension::merge extends an extension to include a new area in addition to its current one.

Extension::set_xy and Extension::merge_xy are short-hand for adjusting an extension to include a given bounding box. Extension::left, bottom, right, and top return the bounding box.

## 2.8 Polyglyphs

A composite glyph contains one or more components. Any glyph subclass may contain subcomponents, but composite glyphs are typically derived from PolyGlyph to be able to store and retrieve their children.

The PolyGlyph protocol, shown in Figure 2.6 adds one operation beyond the base class, PolyGlyph::modified(GlyphIndex), which notifies the glyph that the given component has changed. This operation is called when a glyph is inserted or removed from the composite.

## 2.9 MonoGlyphs

MonoGlyph is an abstract class for glyphs that contain a single glyph, called its *body*. The default behavior of monoglyphs is to pass operations on to the body. For example, the implementation of MonoGlyph::draw simply calls draw on

```
interface PolyGlyph : Glyph {
  PolyGlyph(GlyphIndex initial_size = 10);
  void modified(GlyphIndex);
};
```

Figure 2.6: Polyglyph protocol

the body. This feature is very useful because it means that a monoglyph can affect one aspect of the body's behavior without changing other aspects. In particular, monoglyphs usually do not change structure management. Therefore, a monoglyph can be "wrapped" around a composite glyph and the result passed to an object that modifies the structure of the composite without knowing about the presence of the monoglyph. For example, suppose a prototype menu is built and returned to a function that appends the items. All the function need assume is that the prototype is some glyph; it can use Glyph::append to put in the items. The arrangement of the items will be defined by the aggregate and any additional decoration, such as a 3D borderframe or shadow, can be wrapped around the aggregate independently.

Figure 2.7 shows the MonoGlyph base class interface. A non-nil glyph specifies the initial body. MonoGlyph::body is overloaded to set and return the body.

### 2.9.1 Patches

A patch stores its canvas and allocation for subsequent use to update its body. A patch is useful for objects that need to be redrawn independently, such as those that depend on some external data. Typically, a patch appears in the middle of a glyph instance graph, where a change would be too expensive to redraw the entire graph.

Figure 2.8 shows the Patch class interface. Patch::canvas and Patch::allocation return the canvas and allocation most recently passed to Patch::draw. Patch::reallocate calls allocate on the body using the current allocation, usually because the body has changed in some way and will allocate its itself differently. Patch::redraw calls draw on the body using the current allocation. Patch::repick returns the result of calling pick on the body with the current canvas and allocation.

```
interface MonoGlyph : Glyph {
  void body(Glyph*);
  Glyph* body() const;
};
```

Figure 2.7: MonoGlyph protocol

```
interface Patch : MonoGlyph {
  Patch(Glyph*);
  Canvas* canvas() const;
  const Allocation& allocation() const;
  void reallocate();
  void redraw() const;
  boolean repick(int depth, Hit&);
};
```

Figure 2.8: Patch protocol

## 2.10 Example

See iv/src/examples/circle.

# Chapter 3

# Event processing

This chapter describes the InterViews mechanisms for processing user input. An *event* is an object that represents a user input action, such as pressing a mouse button or a keystroke. Events are low-level objects that application code should rarely need to access directly. A *handler* is an object that processes events. An *input handler* is a glyph that provides a handler that translates events to operations such as press, release, and keystroke.

## 3.1 Events

The event class interface is shown in Figure 3.1. Input events arrive in a single stream (per display) to an application. EventType is the kind of event (mouse motion, button down, etc.). An application receives all window system events, but Event::type is only defined for device input. Other events have the type Event::other_event. EventButton is the number of a button on a pointing device, where Event::any refers to any one of the buttons.

Event::window returns the window that received the event. Event::pending tests if more events are waiting to be read. Event::unread puts the event at the front of the input queue for the display.

```
typedef unsigned int EventType;
typedef unsigned long EventTime;
typedef unsigned int EventButton;
typedef unsigned int EventModifierKey;

interface Event {
   enum { undefined, motion, down, up, key, other_event };
   enum { none, any, left, middle, right, other_button };
   enum { control, shift, capslock, meta };

   Window* window() const;
   boolean pending();
   void unread();
   EventType type() const;
   EventTime time() const;
   Coord pointer_x() const, pointer_y() const;
   EventButton pointer_button() const;
   boolean button_is_down(EventButton) const;
   boolean modifier_is_down(EventModifierKey) const;
   unsigned char keycode() const;
   unsigned int mapkey(char*, unsigned int len) const;
};
```

Figure 3.1: Event protocol

Event::time returns a timestamp in milliseconds for the event. Event::pointer_x and Event::pointer_y return the pointer location for the event in coordinates relative to the lower-left corner of the window. Pointer locations are defined for motion, button, and keyboard events. Event::pointer_button returns the EventButton for a down or up event, Event::none for all other events. Event::button_is_down (Event::modifier_is_down) returns whether the given button (modifier key) was down *before* the event occurred.

Event::keycode returns the code associated with a keystroke. Key codes are potentially platform-specific and should be avoided if possible. Event::mapkey finds the string binding for a given key. Event::mapkey is passed a buffer for the result and the size of the buffer; it returns the number of characters in the translated string. If the event is not a key event, Event::mapkey returns -1.

## 3.2 Handlers

A handler is an object that is passed an event to process. The only operation is Handler::event, which translates a raw event into an abstract operation. InputHandler is a monoglygph subclass that uses a handler to translate events. ActiveHandler is a subclass of input handler that detects when the input pointer enters or leaves its body. Figure 3.2 shows the handler, input handler, and active handler protocols.

The InputHandler constructor is passed the glyph body (which can also be set with MonoGlyph::body) and a style. Though a glyph can in general have multiple parents, input handlers form a hierarchy for focus management. InputHandler::append_focusable adds a child, setting the child's parent to the target input handler. InputHandler::remove_focusable removes a child. InputHandler::focus sets the current focus for the input handler. The effect of setting focus is that key events (or in general any focusable event) will be processed by the focus input handler as opposed to an input handler under the input pointer coordinates.

InputHandler::next_focus and InputHandler::prev_focus move the focus forward and backward through the children of an input handler. InputHandler::focus_in moves the focus "down" the hierarchy and returns the handler that now has focus. InputHandler::focus_out notifies an input handler that it no longer will receive focusable events, in case it wishes to change its appearance.

InputHandler::move, press, drag, release, and keystroke are called by the handler in response to motion, button down, motion while a button is down, button up, and key events. InputHandler::double_click is called when two button down events occurs within a threshold time. The threshold is set by the "clickDelay" style attribute; the default is 250 milliseconds.

InputHandler::allocation_changed is a notification that the glyph's size or position has changed; subclasses can override this operation and therefore

```
interface Handler : Resource {
  boolean event(Event&) ;
};

interface InputHandler : MonoGlyph {
  InputHandler(Glyph*, Style*);

  Handler* handler() const;
  InputHandler* parent() const;
  Style* style() const;
  void append_input_handler(InputHandler*);
  void remove_input_handler(InputHandler*);
  void focus(InputHandler*);
  void next_focus();
  void prev_focus();
  InputHandler* focus_in();
  void focus_out();

  void move(const Event&);
  void press(const Event&);
  void drag(const Event&);
  void release(const Event&);
  void keystroke(const Event&);
  void double_click(const Event&);

  void allocation_changed(Canvas*, const Allocation&);
  void redraw() const;
};

interface ActiveHandler : InputHandler {
  ActiveHandler(Glyph*, Style*);
  void enter();
  void leave();
};
```

Figure 3.2: Handler and InputHandler protocols

typically do not need to define an allocate operation. InputHandler::redraw damages the canvas and area where the input handler was most recently drawn.

The implementation of ActiveHandler notices when motion events first intersect the glyph's body, in which case ActiveHandler::enter is called, and when the motion next leaves the body, in which case ActiveHandler::leave is called. The default enter and leave operations do nothing; subclasses should define these operations as desired.

## 3.3 Targets

A *target* controls the behavior of picking on its body. A target is useful for defining a pick area that is not identical to the underlying object. For example, if one wanted to pick in a line of characters then one might wish to interpret the pick based on the line height instead of the heights of the individual characters. So, a pick lower than the top of the ‘‘h’’ in ‘‘the’’ but above the ‘‘e’’ will not miss.

Figure 3.3 shows the target class interface. The target behavior is controlled by the sensitivity parameter to the constructor. TargetPrimitiveHit means the body is hit if the pick region intersects the target’s allocation. TargetCharacterHit means the body is hit if the area intersects the target’s allocation in the X dimension.

```
enum TargetSensitivity {
  TargetPrimitiveHit,
  TargetCharacterHit
};
interface Target : MonoGlyph {
  Target(Glyph* body, TargetSensitivity);
};
```

Figure 3.3: Target class interface

# Chapter 4

# Views

This chapter describes the mechanisms for supporting multiple views of data. Currently, this support consists of three protocols: *observable*, *observer*, and *adjustable*. An observable object is one that has one or more observers that it can notify (typically when it changes). An adjustable object uses an observable for each dimension to keep track of observers that can scroll or zoom the visible areas of the object.

## 4.1 Observable and Observer

Figure 4.1 shows the observable and observer protocols. An observer can be attached or detached to an observable. Observable::notify calls Observer::update on each of the attached observers. When an observable object is destroyed, it will call Observer::disconnect on any attached observers.

## 4.2 Adjustable

An adjustable is an object that handles requests to modify its viewing area. For example, a scrollable list is adjustable in that a scrollbar or other object can request that a particular subrange of items in the list be shown.

Figure 4.2 shows the adjustable protocol. Adjustable::attach and Adjustable::detach add an observer to the list of objects notified when a particular dimension changes. Adjustable::notify calls update on those observers attached to the given dimension. Adjustable::notify_all is equivalent to calling notify on every dimension.

```
interface Observable {
   void attach(Observer*);
   void detach(Observer*);
   void notify();
};

interface Observer {
   void update(Observable*);
   void disconnect(Observable*);
};
```

Figure 4.1: Observable and observer protocols

```
interface Adjustable {
   Observable* observable(DimensionName) const;
   void attach(DimensionName, Observer*);
   void detach(DimensionName, Observer*);
   void notify(DimensionName) const;
   void notify_all() const;

   Coord lower(DimensionName) const;
   Coord upper(DimensionName) const;
   Coord length(DimensionName) const;
   Coord cur_lower(DImensionName) const;
   Coord cur_upper(DimensionName) const;
   Coord cur_length(DimensionName) const;

   void scroll_forward(DimensionName);
   void scroll_backward(DimensionName);
   void page_forward(DimensionName);
   void page_backwards(DimensionName);

   void scroll_to(DImensionName, Coord lower);
   void scale_to(DimensionName, float fraction);
   void zoom_to(float magnification);

   void constrain(DimensionName, Coord&) const;
};
```

Figure 4.2: Adjustable protocol

Adjustable::lower and Adjustable::upper define a coordinate range for adjustment in a specific dimension. Adjustable::length is equivalent to Adjustable::upper minus Adjustable::lower. Adjustable::cur_lower and Adjustable::cur_upper define the region within a coordinate range that is currently in view. Adjustable::cur_length is equivalent to Adjustable::cur_upper minus Adjustable::cur_lower.

Adjustable::scroll_forward and Adjustable::scroll_backward add a small fixed amount to Adjustable::cur_lower (where the amount is determined by the adjustable object). Adjustable::page_forward and Adjustable::page_backward add a larger fixed amount. Adjustable::scroll_to sets the adjustable's current lower coordinate to a specific value. Adjustable::scale_to sets the current length to a given fraction of the total length. Adjustable::zoom_to sets the scale in all dimensions at once. Adjustable::constrain restricts a value to be within the adjustable's lower and upper bounds.

*Example: see iv/src/examples/bvalue/main.c.*

## 4.3 View

*This section will be completed in the future.*

## 4.4 Data

*This section will be completed in the future.*

## 4.5 Inset

*This section will be completed in the future.*

# Chapter 5

# Windows

The previous chapters covered the composition of physical (glyphs) and logical (views) objects. This chapter discusses the other objects necessary to manage where glyphs are drawn. A *window* is an object that can be mapped onto a *screen* and receive input. Associated with a window is the root of a directed acyclic graph of glyphs. The window creates a *canvas* that is bound to a portion of the screen when the window is mapped. The window calls Glyph::draw on the root glyph to refresh the canvas and Glyph::pick to determine what to do with input events

Figure 5.1 shows the Window base class interface. Window::style sets or gets the style associated with the window. Several style attributes control characteristics of a window: "double_buffered" controls whether the window is double-buffered by default, "visual" specifies the name of a visual to use for the window, "visual_id" specifies the id of the desired visual, and "overlay" specifies whether overlay planes should be used for the window.

```
interface Window {
   Window(Glyph* = nil);
   void style(Style*);
   Style* style() const;
   void display(Display*);
   Display* display() const;
   Canvas* canvas() const;
   void cursor(Cursor*);
   Cursor* cursor() const;
   void push_cursor();
   void pop_cursor();

   void place(Coord left, Coord bottom);
   void align(float x, float y);
   Coord left() const, bottom() const, width() const, height() const;

   void map(), unmap();
   boolean is_mapped() const;
   void raise(), lower();
   void move(Coord left, Coord bottom);
   void resize();

   void receive(const Event&);
   void grab_pointer(Cursor* = nil) const;
   void ungrab_pointer() const;
   void repair();
};
```

Figure 5.1: Window protocol

Window::display sets or gets the display that a window is mapped on (or nil if the window is not currently mapped). Window::canvas returns the canvas that the window passes to its glyph for drawing.

Window::cursor sets or gets the image that tracks a pointing device's position when the it is inside the window. Cursors are defined by two bitmaps and a ‘‘hot spot’’. The mask bitmap specifies which pixels are to be drawn, and the pattern bitmap specifies which pixels are in foreground color and which are in background color. The hot spot specifies the location of the pointing device relative to the cursor's lower-left corner. The foreground and background colors for a cursor are defined by the attributes “pointerColor” and “pointerColorBackground” in the window's style.

Figure 5.2 shows the cursor protocol. A cursor can be created from specific data, pattern and mask bitmaps, a character in a font, or an index into the standard cursor information for the target window system. If the cursor is specified with bitmaps, the hot spot is the origin of the pattern bitmap; if specified with a character, it is the origin of the character's bitmap in the font. An index implies both bitmaps as well as the hot spot. Specific values for the index are usually defined in an include file. For example, standard X11 cursors are defined in the file <X11/cursorfont.h>.

Window::place specifies the desired screen coordinates for a window. Window::align specifies a desired alignment. The alignment values are fractions that indicate where the window should appear relative to its coordinates as specified by a call to Window::place. For example, an alignment of 0.0,1.0 means the placement specifies the upper-left corner of the window. An alignment of 0.5,0.5 means the placement specifies the center of the window. Unless specified, the alignment will be 0.0,0.0, meaning the placement coordinates specify the lower-left corner of the window.

Window::map requests the window to be mapped onto the screen. If no display has been set, then the session's default display is used. Window::map may be asynchronous–one cannot assume the window is usable immediately after calling map. Window::unmap requests that the window be removed from the screen.

Window::raise and Window::lower control the stacking order of a window on the screen. Window::raise makes the window above all the other windows on the screen; Window::lower make the window below all the others. These operations usually are neither necessary nor appropriate, as stacking order should normally

```
interface Cursor {
   Cursor(short x, short y, const int* pattern, const int* mask);
   Cursor(const Bitmap* pat, const Bitmap* mask);
   Cursor(const Font*, int pattern, int mask);
   Cursor(int index);
};
```

Figure 5.2: Cursor class interface

be under control of the user through a window manager.

The window systems delivers input events to a particular window, which in turn tries to find a handler to process each event. Window::receive examines an event, handling window maintenance events internally. For example, X expose and configure events are handled directly by Window::receive.

Window::grab_pointer takes control of pointer input events for the display. Other applications will not receive pointer events until Window::ungrab_pointer is called to release control. If a cursor is passed to Window::grab_pointer, it will be used when the pointer is outside the window during the grab.

If any part of a window's canvas has been damaged, Window::repair will call draw on the root glyph and perform the necessary screen update when double-buffering. All windows on a display will be repaired automatically before blocking for input from the display, so applications usually need not call Window::repair directly.

## 5.1 ManagedWindow

A managed window specifies information for a window manager to use. ManagedWindow is an abstract base class with four predefined descendants: ApplicationWindow, TopLevelWindow, TransientWindow, and IconWindow. An application should create one application window, which makes information about command-line arguments available to a session manager, if present on the system. After the application window, normal windows are top-level. A top-level window can have a ‘‘group leader’’, which is typically the application window. A window manager may allow quick control of all windows with the same group leader, such as when iconifying or deiconifying.

A transient window is often treated specially by window managers. It may be decorated differently (or not at all), or automatically unmapped when the main window is iconified. Transient windows are used for temporary controls, such as dialog boxes, but not for unmanaged windows such as popups or pulldowns. A transient is usually associated with another managed window.

An icon window is a window that is mapped when its associated window is iconified and unmapped when its associated window is deiconified. Calling Window::map on an icon window will therefore bind it to the window system, but will not map it on the screen.

Figure 5.3 shows the ManagedWindow protocol. Most of the operations on a managed window set or return information associated with the window. In addition, ManagedWindow interprets additional attributes in its style inherited from the base class. The attribute “name” specifies a string name for the window manager to use, “iconName” specifies a string for the window's icon, “geometry” specifies the desired geometry, and “iconGeometry” specifies the desired geometry for the window's icon. Geometry specifications are strings of the form ‘‘WxH+X+Y’’ where W is the width, H the height, X the left corner, and

```
interface ManagedWindow : Window {
  void icon_bitmap(Bitmap*);
  Bitmap* icon_bitmap() const;
  void icon_mask(Bitmap*);
  Bitmap* icon_mask() const;
  void icon(ManagedWindow*);
  ManagedWindow* icon() const;

  void iconic(boolean);
  boolean iconic() const;
  void iconify();
  void deiconify();

  void focus_event(Handler* in, Handler* out);
  void wm_delete(Handler*);
};
```

Figure 5.3: ManagedWindow class interface

Y the top corner of the window. Either the position or the size may be omitted, and the position can use ‘‘–’’ instead of ‘‘+’’ to denote distance from the opposite of the display to the opposite side of the window. For example, a –X value specifies that the right side of the window should be a distance of X from the right edge of the screen.

ManagedWindow::icon_bitmap and ManagedWindow::icon_mask specify two bitmaps to use to draw an icon. The mask defines the area to be drawn and the bitmap defines the foreground and background areas. Pixels that correspond to a one in the bitmap and a one in the mask are drawn with the foreground color. Pixels that correspond to a zero in the bitmap and a one in the mask are draw with the background color. Pixels that correspond to a zero in the mask are not drawn.

ManagedWindow::icon specifies a second window to map when the first window is iconified. Using a window as an icon overrides the other icon information. Therefore, it does not make sense to use ManagedWindow::icon in conjunction with icon_bitmap, icon_mask, or the “iconName” and “iconGeometry” attributes.

ManagedWindow::iconic specifies the initial state of a window. If iconic is true, mapping a window will actually map its icon instead. ManagedWindow::iconic need not be called directly by an application; it is called automatically if specified by user customization information.

ManagedWindow::iconify requests the window be unmapped and the window’s icon be mapped to the screen. ManagedWindow::deiconify reverses the operation, unmapping the icon and mapping the original window. It does not make sense to iconify or deiconify an icon window.

ManagedWindow::focus_event specifies handlers for the window receiving and losing keyboard focus from the window manager. ManagedWindow::wm_delete specifies a handler for a request from the window manager to delete the window.

```
interface ApplicationWindow : ManagedWindow {
   ApplicationWindow(Glyph*);
};

interface TopLevelWindow : ManagedWindow {
   TopLevelWindow(Glyph*);
   void group_leader(ManagedWindow*);
   ManagedWindow* group_leader() const;
};

interface TransientWindow : TopLevelWindow {
   TransientWindow(Glyph*);
   void transient_for(ManagedWindow*);
   void ManagedWindow* transient_for() const;
};

interface IconWindow : ManagedWindow {
   IconWindow(Glyph*);
};
```

Figure 5.4: ManagedWindow subclasses

If the handler is nil (which is the initial value), then the response to this event will be to call Session::quit.

Figure 5.4 shows the operations on the ManagedWindow subclasses. ApplicationWindow and IconWindow provide no additional operations beyond a constructor. TopLevelWindow provides an operation to set or return its group leader. TransientWindow is a subclass to TopLevelWindow that can additionally be associated with a primary window with TransientWindow::transient_for.

## 5.2 PopupWindow

A popup window is mapped directly to a screen without window manager interaction (or knowledge). In the X Window System, a popup window will override the normal redirection of map requests to window managers. Popups on X also will request that the pixels under the popup be saved to avoid a subsequent exposure when the popup is unmapped.

Popups should only be used for temporary windows, such as popup or pulldown menus. Because they do not go through the window manager, popups should be placed explicitly. Here is an example of using a popup that appears below a menubar, aligning the top of the popup to the lower left corner of the menubar:

```
void pulldown(Window* menubar, Glyph* g) {
   PopupWindow* popup = new PopupWindow(g);
      popup->place(menubar->left(), menubar()->bottom());
      popup->align(0.0, 1.0);
      popup->map();
```

```
}
```

## 5.3 Display

A *display* is the unit of window system control; typically it consists of a single screen, keyboard, and a mouse or other pointing device. Application objects typically need not deal directly with a display; the functionality of the window class is normally sufficient.

Figure 5.5 shows the display class interface. Display::open is a static member function that opens a connection to the display with the given name. The interpretation of a display name is system-dependent. On X, the name is *host:number* where *host* is a machine's hostname and *number* is the index for the display connected to that host (typically 0). If successful, Display::open returns a pointer to a display object. If not successful, it returns nil. Display::close terminates the connection.

Display::width and Display::height return the dimensions in coordinates of the display's current screen. Display::a_width and Display::a_height return the dimensions in points (72 points = one inch).

---

```
interface Display {
  static Display* open(const String&);
  static Display* open();
  virtual void close();

  virtual Coord width() const;
  virtual Coord height() const;
  virtual Coord a_width() const;
  virtual Coord a_height() const;
  int to_pixels(Coord) const;
  Coord to_coord(int) const;

  virtual void set_screen(int);

  virtual void style(Style*);
  virtual Style* style() const;

  virtual void repair();
  virtual void flush();
  virtual void sync()

  virtual void ring_bell(int);
  virtual void set_key_click(int);
  virtual void set_auto_repeat(boolean);
  virtual void set_pointer_feedback(int thresh, int scale);
  virtual void move_pointer(Coord x, Coord y);
};
```

Figure 5.5: Display class interface.

---

Display::to_pixels and Display::to_coord convert between coordinates and pixels. The conversion is a function of the *dpi* attribute, which is 75 by default. One coordinate unit length is a printer's point, defined as 72/*dpi* pixels.

Display::set_screen sets the current screen to use for display operations. Initially, current screen is set to 0.

Display::repair calls Window::repair for each window on the display that has a canvas with damage. It is not necessary to call Display::repair directly, as windows will automatically be repaired before blocking for input events.

Display::flush and Display::sync are used to synchronize with the window system. Display::flush repairs all damaged windows on the display and ensures that any pending requests have been sent to the window system. Display::sync is the same as Display::flush, but additionally waits for an acknowledgement from the window system.

Display::ring_bell sounds the workstation's bell at a specified volume. The parameter should be between 0 and 100, where 0 is silent and 100 is the loudest possible bell.

The operations set_key_click, set_auto_repeat, and set_pointer_feedback modify the key click volume, the flag determining whether keys should repeat, and the pointer interpretation parameters, respectively. Display::move_pointer changes the position of the input pointer. This operation can have surprising effects to the user and should generally be avoided.

# Chapter 6

# Rendering

This chapter describes the InterViews classes for drawing on the screen and on a printer. The two primary classes are Canvas, which represents an area on the screen, and Printer, which sends output suitable for printing to an output stream. The drawing classes are intended to be simple and resolution-independent. The programming interface resembles the PostScript drawing operations.

Printer is a subclass of Canvas, and as such implements the same drawing operations as Canvas. However, it is possible that glyphs may use other rendering operations than those provided by Canvas, such as for 3D. In this case, glyphs should provide distinct draw and print operations. If a glyph does not need operations other than those provided by Canvas then the glyph can rely on default implementation of drawing on a printer, which just calls the canvas-directed draw with the printer as the target.

## 6.1 Graphics Attributes

InterViews provides classes that represent graphics attributes such as colors and fonts. The instances are all sharable, meaning the classes are derived from Resource. The objects are also display-independent, meaning they will correspond to several underlying objects in applications that run on multiple displays. For example, a single InterViews color object might have different pixel values on different displays.

### 6.1.1 Brush

A brush defines the line thickness and line style for drawing operations. 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 resulting display may vary slightly across devices. A solid brush style paints all pixels along the path with a single color. A dashed brush defines alternating foreground and background segments, measured along the length of the path. Foreground segments are painted, while background segments are not.

Figure 6.1 shows the Brush class interface. The first constructor creates a solid brush of the given width. The second constructor creates a brush with the given width and line style. The pattern is 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

```
interface Brush : Resource {
  Brush(Coord width);
  Brush(int* pattern, int count, Coord width);
  Brush(int pattern, Coord width);
};
```

Figure 6.1: Brush class interface

indices specify background segments. The count is the number of entries in the array. The count can be zero, which specifies a solid brush. The last constructor defines 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, with one bits specifying foreground segments and zero bits specifying background segments.

### 6.1.2 Color

A color object defines an output color, which is specified by a mix of RGB (red, green, and blue) intensities, and an alpha value for blending. Figure 6.2 shows the Color class interface. RGB and alpha values are represented as floating point numbers between 0 and 1, where 1 is full intensity (or visibility in the case of alpha). A color object is created with the RGB intensities, an alpha value (default is 1.0), and a drawing operation. A color drawing operation need be specified only on rare occasions. The default operation, Copy, blends the color in directly. The Xor operation uses a pixel value computed by taking the exclusive-or of the color and the existing pixel value. Xor is only useful on a monochrome system.

Color::lookup returns the color with the given name as defined on the given display or nil if the name is not defined. Color::distinguished determines if two colors are distinct on a particular display. A common use of Color::distinguished is to check if a highlighting color is distinct from foreground and background colors. Color::intensities returns the RGB values for a given color on a given display. Color::brightness creates a new color that is brighter or darker than the given color by a certain adjustment. If the adjust parameter is positive, it indicates the new intensity should be the given fraction of the distance between the current intensity and full intensity. If the parameter is negative, its absolute value specifies a distance to zero intensity.

InterViews automatically translates an RGB specification to the appropriate pixel value for a window. This approach hides the system-dependent details of color management from applications, making them more portable and giving greater flexibility to graphics system implementors. Under the X Window System, color-intensive applications might not find the default color implementation acceptable. To assist such applications, InterViews provides a way to specify an X visual, either on the command-line with the ‘‘-visual’’ flag, or with a ‘‘visual’’ X resource defined to the desired visual type. For example, on displays that support TrueColor (which means pixel values can be computed directly from RGB values) but for which the default visual is not TrueColor, a user could run an application with ‘‘-visual TrueColor’’ or define ‘‘*app*visual:TrueColor’’ in the

```
typedef float ColorIntensity;
typedef unsigned int ColorOp;

interface Color : Resource {
  enum { Copy, Xor };
  Color(
    ColorIntensity r, ColorIntensity g, ColorIntensity b,
    float alpha = 1.0, ColorOp = Copy
  );
  Color(const Color&, float alpha = 1.0, ColorOp = Copy);
  static const Color* lookup(Display*, const String& name);
  static const Color* lookup(Display*, const char*);
  boolean distinguished(Display*, Color*);
  void intensities(
    Display*, ColorIntensity& r, ColorIntensity& g, ColorIntensity& b
  ) const;
  const Color* brightness(float adjust) const;
};
```

Figure 6.2: Color class interface

application defaults file.

### 6.1.3 Font

A font defines a mapping between character codes and their appearance on the screen. PSFont is a subclass of Font that uses PostScript metrics for character widths, if the metrics are available on the system. Figure 6.3 shows the Font and PSFont class interfaces. The Font constructor is given the full name of the font and a scaling factor. If the font is used on a display that does not recognize the name, then a default font will be used. Font::find can be used to compute a valid fullname for a font from a given font family name, desired point size, and font style (such as italic or bold). If a font is available that matches all but the point size, Font::find will return the font with scale set to the ratio of the desired point size to the actual point size.

Font::name returns the full name of the font. Font::encoding returns the character set identification, such as ‘‘iso8859’’ for ISO Latin. Font::size returns the point size of the font.

Font::font_bbox, Font::char_bbox, and Font::string_bbox return information about the overall font, a specific character in the font, or a string of characters. Each operation returns a FontBoundingBox object, which has operations to return detailed information.

FontBoundingBox::ascent returns the extent above the font’s baseline; FontBoundingBox::descent returns the extent below the font’s baseline. FontBoundingBox::left_bearing returns the left edge of the bitmap associated with a character; FontBoundingBox::right_bearing returns the right edge.

Font::width on a single character returns the width of a character’s bitmap and on a string it returns the sum of the widths of the individual characters.

```
interface FontBoundingBox {
   Coord left_bearing() const, right_bearing() const;
   Coord width() const, ascent() const, descent() const;
   Coord font_ascent() const, font_descent() const;
};

typedef long FontCharCode;

interface Font : Resource {
   Font(const String&, float scale = 1.0);
   Font(const char* fullname, float scale = 1.0);
   static boolean find(
      const char* family, int size, const char* style,
      const char*& fullname, float& scale
   );
   static const Font* lookup(const char*);
   static const Font* lookup(const String&);

   const char* name() const;
   const char* encoding() const;
   Coord size();

   void font_bbox(FontBoundingBox&) const;
   void char_bbox(FontCharCode, FontBoundingBox&) const;
   void stringt_bbox(const char*, int, FontBoundingBox&);
   virtual Coord width(FontCharCode);
   virtual Coord width(const char*, int);
   virtual int index(const char*, int, float offset, boolean between);
};

interface PSFont : Font {
   PSFont(
      const char* psname, Coord size, const char* encoding, float scale
   );
};
```

Figure 6.3: Font and PSFont class interfaces

Font::index returns the index of the character in a string that would be offset coordinates 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 will return an index equal to the length of the string.

### 6.1.4 Transformer

A transformer object represents a 3x2 matrix for use in translating 2D coordinates. Figure 6.4 shows the Transformer class interface. The transformer constructor with no parameters creates an identity matrix. The other constructor takes the explicit matrix values as parameters. Transformer::identity returns whether the matrix is currently the identity matrix.

Transformer::premultiply and Transformer::postmultiply set the matrix to be the result of multiplying the matrix and the given matrix. Two operations are necessary because matrix multiplication is not commutative for 2D coordinates. Premultiply means the current matrix is on the left-hand side of the multiplication, postmultiply means the current matrix is on the right. Transformer::invert sets the matrix to its inverse.

Transformer::translate modifies the matrix to add dx to the x coordinate and dy to the y coordinate. Transformer::scale modifies the matrix to multiply the x and y coordinates by sx and sy, respectively. Transformer::rotate modifies the matrix to rotate x and y coordinates by a given angle in degrees. Transformer::skew modifies the matrix to skew coordinates by sx and sy.

Transformer::transform multiplies the given coordinates by the matrix to compute transformed coordinates. The coordinates can either transformed in place stored in specific out parameters (tx, ty). Transformer::inverse_transform performs the inverse mapping; taking transformed coordinates and returning the original coordinates.

The following example shows how to use transformers:

```
Transformer t;      // start with identity
t.rotate(90.0);
t.translate(20.0, 10.0);
t.scale(0.5, 0.5);
float x = 1.0, y = 0.0;
float tx, ty;
```

```
interface Transformer : Resource {
  Transformer();
  Transformer(
    float a00, float a01, float a10, float a11, float a20, float a21
  );
  boolean identity() const;
  void premultiply(const Transformer&);
  void postmultiply(const Transformer&);
  void invert();
  void translate(float dx, float dy);
  void scale(float sx, float sy);
  void rotate(float angle);
  void skew(float sx, float sy);
  void transform(floatx, floaty);
  void transform(float x, float y, floattx, floatty);
  void inverse_transform(floatx, floaty);
  void inverse_transform(float tx, float ty, floatx, floaty);
  void matrix(floata00, floata01, floata10, floata11, floata20, floata21);
};
```

Figure 6.4: Transformer protocol

```
t.transform(x, y, tx, ty);
// now tx = 10.0, ty = 5.5
```

Although the transformation is a single step, one can think of it as individual steps for each of the rotate, translate, and scale steps. First the given point (1.0,0.0) is rotated to (0.0,1.0), then it is translated to (20.0,11.0), finally it is scaled to (10.0,5.5).

### 6.1.5 Bitmap

A bitmap is a two-dimensional array of boolean values. A bitmap is useful for stenciling; that is, drawing through a mask that allows some pixels to be drawn but prevents others from being changed. The Stencil class can be used to put a bitmap in a glyph graph.

Figure 6.5 shows the Bitmap class interface. There are two constructors for bitmaps. One takes the bitmap data, width, height, and origin. The other constructor creates a bitmap for a given character in a font, optionally scaling by a given factor. In this case, the bitmap width and height will reflect the actual size of the character glyph and the bitmap origin will be the same as the character origin.

Bitmap::open operation tries to open a file containing a bitmap definition in the format produced by the X bitmap program. If the file is found and is a valid format, open returns true and sets the bitmap information.

Bitmap::peek and Bitmap::poke are used to read and write at specified positions in the bitmap. Bitmap::width and Bitmap::height return the width and height of the bitmap in coordinates, while Bitmap::pwidth and Bitmap::pheight return the number of bits defined in each dimension.

Treating the bitmap origin as (0,0), Bitmap::left_bearing, Bitmap::right_bearing, Bitmap::ascent, and Bitmap::descent return the left, right, top, and bottom coordinates of the bitmap, respectively. For example, a 16x16 bitmap with its origin at (7,5) would have a left_bearing of -7, a right_bearing of 9, an ascent of 12, and a descent of -5.

```
interface Bitmap : Resource {
  Bitmap(
    void*, unsigned int width, unsigned int height, int x0 = -1, int y0 = -1
  );
  Bitmap(Font*, int code, float scale = 1.0);
  static Bitmap* open(const char* filename);
  void poke(boolean set, unsigned int x, unsigned int y);
  void peek(unsigned int x, unsigned int y);
  Coord width() const, height() const;
  unsigned int pwidth() const, pheight() const;
  Coord left_bearing() const, right_bearing() const;
  Coord ascent() const, descent() const;
};
```

Figure 6.5: Bitmap protocol

### 6.1.6 Raster

A raster is a color image specified by a two-dimensional array of colors. The Image class can be used to put a raster in a glyph graph. The TIFFRaster class provides a single operation, load, for reading a TIFF image file and creating a raster for it. If the file is not readable or not a valid TIFF file, TIFFRaster::load will return nil.

Figure 6.6 shows the Raster and TIFFRaster class interfaces. The raster constructor is given the size of the array of colors. Raster::width and Raster::height return the dimensions of the raster in coordinates, while Raster::pwidth and Raster::pheight return the dimensions of the array. A raster's origin is always the lower left corner.

Raster::peek and Raster::poke read and write the color array, accessing colors in terms of the RGB intensities and an alpha value. Peek and poke operations are guaranteed to be cheap; that is, any processing (especially interaction with the window system) will be deferred until the raster is next displayed.

## 6.2 Canvas

A canvas is a 2-dimensional area on which to draw. The base implementation draws on a portion of the screen, normally created by a window object rather than directly by an application. The Printer subclass uses the same rendering operations to generate PostScript to a file. Thus, it is possible to write a single drawing routine that can be used to generate screen or printer output.

Figure 6.7 shows the canvas and printer operations. For screen canvases, Canvas::window returns the window containing the canvas; otherwise it returns

```
interface Raster : Resource {
  Raster(unsigned int pwidth, unsigned int pheight);
  Coord width() const, height() const;
  unsigned int pwidth() const, pheight() const;
  void peek(
    unsigned int x, unsigned int y,
    ColorIntensity& r, ColorIntensity& g, ColorIntensity& b, float& alpha
  ) const;
  void poke(
    unsigned int x, unsigned int y,
    ColorIntensity r, ColorIntensity g, ColorIntensity b, float alpha
  );
};
interface TIFFRaster {
  static Raster* load(const char* filename);
};
```

Figure 6.6: Raster and TIFFRaster protocols

nil. Canvas::width and Canvas::height return the dimensions of the canvas in coordinates.

The canvas rendering operations are similar to the PostScript drawing operations. Canvas::new_path, Canvas::move_to, Canvas::line_to, Canvas::curve_to, and Canvas::close_path are used to define a list of coordinates on which to perform a drawing operation. Canvas::move_to sets the position in the path, and Canvas::line_to extends the path to a new position. Canvas::curve_to also extends the path, but with a Bezier curve between the old and new positions. Canvas::close_path closes the path. Canvas::stroke draws along the current path with a given brush and color. Canvas::fill draws inside the path with a given color. Canvas::clip restricts subsequent drawing to be inside the path. Clipping is cumulative; that is, two consecutive clip operations will result in a clipping region that is the intersection of the paths specified by the two requests.

Canvas::line, Canvas::rect, Canvas::fill_rect, and Canvas::clip_rect are provided for convenience. Canvas::line is equivalent to stroking a path with two points, Canvas::rect strokes a rectangular path, Canvas::fill_rect fills a rectangular path, and Canvas::clip_rect restricts subsequent output to be within a rectangular path.

Drawing operations are typically batched to improve performance. For example, a series of Canvas::character operations might be combined into a single request on many graphics systems. An application cannot determine if or when a particular operation has completed. No synchronization operations are defined on a canvas, as several canvases may be active at the same time. Display::flush or Display::sync can be used to wait until the display starts or finishes drawing, respectively.

As an example of the drawing operations, the following code draws a filled triangle with corners (x1,y1), (x2,y2), and (x3,y3):

```
canvas->new_path();
canvas->move_to(x1, y1);
canvas->line_to(x2, y2);
canvas->line_to(x3, y3);
canvas->close_path();
canvas->fill(color);
```

## 6.3 Printer

A printer is a 2-D drawing surface like a canvas, but that generates output for hardcopy or previewing. The printer class normally generates PostScript text to a file; other printer formats may be available at a particular site.

Printer is a subclass of Canvas with different implementations for the drawing operations. Thus, a printer can be passed to an operation expecting a canvas. The printer class also provides a few additional operations.

The printer constructor takes a pointer to an output stream where the print representation will be written. Printer::resize specifies the boundaries of the printed page. Printer::comment generates text that will appear in the output

```
interface Canvas {
  Window* window() const;
  Coord width() const, height() const;

  PixelCoord to_pixels(Coord) const;
  Coord to_coord(PixelCoord) const;
  Coord to_pixels_coord(Coord) const;

  void new_path();
  void move_to(Coord x, Coord y);
  void line_to(Coord x, Coord y);
  void curve_to(Coord x, Coord y, Coord x1, Coord y1, Coord x2, Coord y2);
  void close_path();
  void stroke(const Color*, const Brush*);
  void rect(Coord l, Coord b, Coord r, Coord t, const Color*, const Brush*);
  void fill(const Color*);
  void fill_rect(Coord l, Coord b, Coord r, Coord t, const Color*);
  void character(
    const Font*, int ch, Coord width, const Color*, Coord x, Coord y
  );
  void stencil(const Bitmap*, const Color*, Coord x, Coord y);
  void image(const Raster*, Coord x, Coord y);

  void push_transform(), pop_transform();
  void transform(const Transformer&);
  void transformer(const Transformer&);
  const Transformer& transformer() const;
  void push_clipping(), pop_clipping();
  void clip();
  void clip_rect(Coord l, Coord b, Coord r, Coord t);

  void damage(const Extension&);
  void damage(Coord l, Coord b, Coord r, Coord t);
  boolean damaged(const Extension&) const;
  boolean damaged(Coord l, Coord b, Coord r, Coord t) const;
};

interface Printer : Canvas {
  Printer(ostream*);
  void resize(Coord left, Coord bottom, Coord right, Coord top);
  void comment(const char*);
  void page(const char*);
  void flush();
};
```

Figure 6.7: Canvas and printer protocols

stream, but will not show on the printed page. Printer::page generates information about the current page. This operation will not result in any printed output, but is used by previewers. Printer::flush forces any locally-buffered data to be written.

# Chapter 8

# Styles

User interface toolkits traditionally have coupled the management of style attributes such as color and font with the composition of objects in a window. This coupling is too rigid and inefficient for many applications because attributes are *logical* information, whereas composition is a *physical* organization. For example, a document logically contains text and graphics organized into chapters, sections, subsections, and paragraphs. Physically, the document contains lines, columns, and pages. The font of a string of characters in the document is independent of whether there is a line break within the string or not, thus the style information is orthogonal to the layout.

InterViews provides a *style* class for organizing user interface attributes. A style is similar to an environment in a text formatting system such as Scribe. Styles may be nested hierarchically, and attributes defined in an outer style are visible in an inner style if not otherwise defined. A style consists of an optional name, an optional list of prefixes for wildcard-matching, a collection of attributes (name-value pairs), a collection of styles nested inside the style, and a parent style.

## 8.1 Defining a style

Figure 8.1 shows the style class operations for creating and accessing simple style information. When a style is created, its name and parent style may be specified. The default parent style is nil. Style::name sets or gets the style's name. Style::parent gets the style's parent. The parent cannot be set directly, but can be changed by appending the style to its (new) parent.

Style::append and Style::remove add and delete a style from the list of styles nested inside another style. Style::children returns the number of nested styles. Style::child returns the indexed child in the list. Style::find_style returns the nested style with the given name or nil if there is none.

Style::attribute adds a <*name,value*> pair to the list of attributes in the style. If an attribute is already defined with the name, the value will be updated unless the specified priority is lower than the already-defined priority of the attribute Style::remove_attribute deletes the named attribute from the style's attribute list. Style::attributes and the get form of Style::attribute can be used to retrieve all the attributes defined on a style. The order of the list is arbitrary.

## 8.2 Finding an attribute

The style class provides two overloaded functions for finding an attribute value

```
interface Style : Resource {
  Style();
  Style(const String& name);
  Style(Style* parent);
  Style(const String& name, Style* parent);

  void name(const String&);
  const String* name() const;
  void alias(const String&);
  long alias_count() const;
  const String* alias(long) const;
  Style* parent() const;

  void append(Style*);
  void remove(Style*);
  long children() const;
  Style* child(long) const;

  void attribute(const String& name, const String& value, int priority = 0);
  void remove_attribute(const String& name);
  long attributes() const;
  boolean attribute(long, String& name, String& value) const;

  void add_trigger(const String& , Action*);
  void remove_trigger(const String&, Action* = nil);
  void add_trigger_any(Action*);
  void remove_trigger_any(Action*);

  boolean find_attribute(const String&, String& value) const;
  boolean find_attribute(const String&, long&) const;
  boolean find_attribute(const String&, double&) const;
  boolean find_attribute(const String&, Coord&) const;
  boolean value_is_on(const String&);
};
```

Figure 8.1: Style protocol.

given the name. Style::find_attribute takes two parameters and returns a boolean value that is true if the attribute is found and false otherwise. The first parameter is the desired name, which can either be passed as a String object or a *const char**. The second parameter is a reference to where the value of the attribute should be stored if found. If the parameter is a string, then the value is simply copied directly. If it is a long or double, then value string is converted to a number. If the result parameter is a Coord, then the value string is converted to a number and multiplied by the units specified in the value string after the number. The unit specification can be “in” for inches, “cm” for centimeters, “mm” for millimeters, “em” for the width in points of the character “m” in the style’s font, and “pt” for points.

The other function for finding an attribute is Style::value_is_on. This function is equivalent to calling Style::find_attribute and testing if the value string is “on” or “true”. The test is case-insensitive.

## 8.3 Wildcard matching

Attribute names may contain "*" characters to specify wildcard matching. A name of the form A*B will match an attribute B in a nested style named A. Wildcard names also may begin with a "*", which matches in any descendant style. Thus, *A*B will match an attribute B in any descendant style named A. Because attributes are inherited, specifying the wildcard name *B is identical to specifying the normal attribute B.

In addition to a name, styles may have a list of associated aliases. Style::alias prepends a string to the list. Wildcard matches search using a style's name first, then search using the style's aliases in the reverse order in which they are defined. Aliases are typically used for indicating a subclass relationship and allowing styles to inherit attributes specified for a superclass.

For example, suppose the root style defines the following attributes:

```
*Mover*autorepeat:off
*UpMover*autorepeat:on
```

Consider descendant styles S and T: S's aliases are UpMover and Mover; T's aliases are DownMover and Mover. Style::find_attribute for "autorepeat" will return "on" for S, "off" for T.

The wildcard matching algorithm is compatible with the X resource manager to support the same user customization functionality. Wildcard attributes typically are defined only on the root style, as loaded from the window system, application defaults files, or command-line arguments.

## 8.4 Using styles with glyphs

Glyphs that draw typically contain the specific style information they need to render. For example, a character glyph contains the font and color it uses to draw. Higher level glyphs, such as a slider for scrolling, contain a style from which they construct their components. When styles support trigger routines to detect attribute value changes, these higher-level components will be able to reconstruct their contents automatically.

Figure 8.2 shows a function that builds a vertical scrollbar by creating a box containing an up-mover (button with up-arrow), a scroller (slider), and a down-mover (button with down-arrow). The function creates a new style and gives it the prefixes VScrollBar and ScrollBar for customization. If the attribute "mover_size" is defined on the style, then its value will override the default (15.0).

## 8.5 Summary

User interface geometry and attribute management are two different problems that are best solved independently. Whereas glyphs define a physical organization with a directed acyclic graph, styles define a logical organization with a strict hierarchy. Both structures are simple and the connection between them is straightforward.

Styles provide a unified framework for managing user-customizable attributes, document formatting information, and structured graphics state. Style prefixes support wildcarding and allow the decoupling of the implementation class hierarchy from the logical class hierarchy offered to the user for the purposes of customization. This approach makes applications simpler to develop, more consistent to use, and easier to integrate.

```
Glyph* vscroll_bar(Adjustable* a) {
  WidgetKit& kit = *WidgetKit::instance();
  const LayoutKit& layout = *LayoutKit::instance();
  kit.begin_style("VScrollBar");
  kit.alias("ScrollBar");
  Style* s = kit.style();
  Coord mover_size = 15.0;
  s->find_attribute("mover_size", mover_size);
  Glyph* sep = layout.vspace(1.0);
  return kit.inset_frame(
    layout.vbox(
      layout.v_fixed_span(up_mover(a, s), mover_size),
      sep,
      new VScroller(a, s),
      sep,
      layout.v_fixed_span(down_mover(a, s), mover_size)
    ),
    s
  );
}
```

Figure 8.2: Using styles to build glyphs.

# Chapter 9

# WidgetKit

WidgetKit defines operations for creating user interface objects with a concrete look-and-feel. Typically, an application uses a widget kit to create specific components, such as pulldown menus, push buttons, and scrollbars. WidgetKit isolates the application from specific look-and-feel issues and the details of how a concrete component is implemented. Many components are defined using instances of existing classes. For example, a push button with a string label is created using a button, label, and two bevel objects.

InterViews provides a base widget kit class for creating common user interface objects. Subclasses are provided for implementing concrete objects based on the Motif (default) and OpenLook user interfaces.

WidgetKit::instance is a static member function that returns a default kit. If a kit has not yet been created, WidgetKit::instance creates one using the session's style to determine which kit subclass to construct.

## 9.1 Style management

WidgetKit maintains a current style for use in customizing widgets. The initial style is the root style for all windows. WidgetKit defines the operations shown in Figure 9.1 for querying and modifying the style.

WidgetKit::style sets or gets the current style. When the style is set or changed via a Style::attribute call, WidgetKit::style_changed is called to allow WidgetKit subclasses to recompute information associated with the current style (such as colors for shading).

WidgetKit::begin_style sets the current style to a newly-created style that is a child of the current style. The given string is the name of the new style. WidgetKit::alias adds an alias name for the current style. Widget::end_style

```
void style(Style*);
Style* style() const;
void begin_style(const String&), end_style();
void alias(const String&);
void push_style(), pop_style();
void style_changed(Style*);
const Font* font() const;
const Color* foreground() const;
const Color* background() const;
```

Figure 9.1: WidgetKit operations for style management

```
Cursor* hand_cursor() const;
Cursor* lfast_cursor() const;
Cursor* lufast_cursor() const;
Cursor* ufast_cursor() const;
Cursor* rufast_cursor() const;
Cursor* rfast_cursor() const;
Cursor* rdfast_cursor() const;
Cursor* dfast_cursor() const;
Cursor* ldfast_cursor() const;
```

Figure 9.2: WidgetKit cursors

returns the current style to what it was before the call to WidgetKit::begin_style. WidgetKit::push_style and pop_style save and restore the current style on a stack.

WidgetKit::font, foreground, and background return specific attribute information for the current style. These operations are equivalent to (though potentially faster than) finding the string value of a style attribute and then looking up the resource with the given name. For example, WidgetKit::font is the same as finding the attribute named "font" and calling Font::lookup on the attribute's value.

## 9.2 Common cursors

WidgetKit provides operations to retrieve commonly-used cursors that might be shared among several widgets. Figure 9.2 shows the currently-defined operations. WidgetKit::hand_cursor is the outline of a small hand. WidgetKit::lfast_cursor is a double-arrow pointing to the left that is typically used in continuous rate scrolling. Similarly, the other "fast" cursors are double-arrows pointing in various directions.

## 9.3 Bevels

On color displays, it is often desirable to frame objects with a beveled look to give a 3D appearance. WidgetKit provides the following three functions for creating beveled frames:

```
Glyph* inset_frame(Glyph*) const;
Glyph* outset_frame(Glyph*) const;
Glyph* bright_inset_frame(Glyph*) const;
```

WidgetKit:inset_frame uses dark shading in the upper left and light shading in the lower right to make the contents of the frame appear to be recessed. WidgetKit::outset_frame reverses the shading to make the contents appear to project out of the frame. WidgetKit::bright_inset_frame is like inset_frame but uses a brighter background color and is thinner.

## 9.4 Labels

A label is a string of text with the current style's font and color. WidgetKit provides two operations for creating labels:

```
Glyph* label(const char*) const;
Glyph* label(const String&) const;
```

## 9.5 Buttons

A button is an input handler that can perform an action when pressed. Buttons manipulate a *telltale state* object so that views (which are typically part of the button's body) can reflect the current state visually. The button itself is a view so that it can damage the canvas when appropriate, freeing other views from the need to store update information.

A telltale state can be part of a *telltale group*. When one member of a group is chosen, then the currently chosen state becomes unchosen. Telltale groups are useful for implementing radio buttons.

Figure 9.3 shows the button, telltale state, and telltale group protocols. Button just provides operations to access the associated telltale state and action. TelltaleState defines a set of flags that define the current state. TelltaleState::set and TelltaleState::test modify and query the current state, respectively. TelltaleState::join and TelltaleState::leave_group allow the state to be associated with a group.

Figure 9.4 shows the WidgetKit operations that return buttons. Push button, default button, and palette button typically have a similar appearance. Neither a push button or a default button can be chosen, whereas a palette button can. A default button might have a different appearance to indicate to the user that it is the common choice.

A check box is a toggle button: choosing it when already chosen will cause it to become unchosen. A radio button must belong to a telltale group so that within the group only one button is chosen at any given time.

For application-specific actions, it is necessary to define action callbacks for the relevant application classes. In the case of quitting the application WidgetKit::quit can be used to return an action that calls Session::quit.

## 9.6 Menus

A menu is similar to a group of related buttons, called menu items. Like a button, the look of a menu item is dependant on a telltalestate. Menu items can have associated actions that are executed when the item is chosen. Menus items can also have associated nested menus, in which case the submenu is opened when the

```
interface Button : ActiveHandler, Observer {
  Button(Glyph*, Style*, TelltaleState*, Action*);
  TelltaleState* state() const;
  Action* action() const;
};

typedef unsigned int TelltaleFlags;

interface TelltaleState : Resource, Observable {
  TelltaleState(const TelltaleFlags = 0);
  enum {
    is_enabled, is_visible, is_enabled_visible, is_active, is_enabled_active,
    is_visible_active, is_enabled_visible_active, is_chosen, is_enabled_chosen,
    is_visible_chosen, is_enabled_visible_chosen,
    is_enabled_active_chosen, is_active_chosen,
    is_visible_active_chosen, is_enabled_visible_active_chosen,
    is_running, is_choosable, is_toggle,
    max_flags
  };

  TelltaleFlags flags() const;
  void set(const TelltaleFlags, boolean);
  boolean test(const TelltaleFlags) const;
  void join(TelltaleGroup*);
  void leave_group();
};

interface TelltaleGroup : Resource {
  void update(TelltaleState*);
  void remove(TelltaleState*);
};
```

Figure 9.3: Button, TelltaleState, and TelltaleGroup protocols.

item is highlighted. Figure 9.5 shows the WidgetKit menu operations and the menu item and menu protocols.

WidgetKit::menubar creates a horizontal menu with a small amount of white space between items. WidgetKit::pulldown and WidgetKit::pullright create vertical menus. The top of a pulldown menu will be aligned to the bottom of the menubar, while a pullright's top will be aligned to the top right of its containing item.

WidgetKit::menubar_item creates an item centered horizontally with a small amount of white space on each side. WidgetKit::menu_item creates a left-adjusted item, WidgetKit::check_menu_item creates a toggle item that looks like a check box when chosen. WidgetKit::radio_menu_item creates an item that looks like a radio button. WidgetKit::menu_item_separator returns an item for visually separating other items (such as a horizontal line).

```
Button* push_button(const String&, Action*) const;
Button* push_button(Glyph*, Action*) const;
Button* default_button(const String&, Action*) const;
Button* default_button(Glyph*, Action*) const;
Button* palette_button(const String&, Action*) const;
Button* palette_button(Glyph*, Action*) const;
Button* check_box(const String&, Action*) const;
Button*  check_box(Glyph*, Action*) const;
Button* radio_button(TelltaleGroup*, const String&, Action*) const;

Glyph* push_button_look(Glyph*, TelltaleState*) const;
Glyph* default_button_look(Glyph*, TelltaleState*) const;
Glyph* palette_button_look(Glyph*, TelltaleState*) const;
Glyph* check_box_look(Glyph*, TelltaleState*) const;
Glyph* radio_button_look(Glyph*, TelltaleState*) const;
```

Figure 9.4: WidgetKit button operations

## 9.7 Adjusters

Scrollbars and mover buttons are examples of interactive objects that “adjust” the view shown by another object. WidgetKit provides the operations shown in Figure 9.6 to create common adjusters.

WidgetKit::hscroll_bar and WidgetKit::vscroll_bar return controls for scrolling a view in the horizontal and vertical dimensions, respectively. WidgetKit::panner returns a control for scrolling two adjustables at once. One adjustable is controlled by the horizontal position of the panner, one by the vertical position. Typically, the same adjustable is passed to both parameters when creating a panners.

```
Menu* menubar() const, * pulldown() const, * pullright() const;
MenuItem* menubar_item(const String&) const;
MenuItem* menubar_item(Glyph*) const;
MenuItem* menu_item(const String&) const;
MenuItem* menu_item(Glyph*) const;
MenuItem* check_menu_item(const String&) const;
MenuItem* check_menu_item(Glyph*) const;
MenuItem* radio_menu_item(TelltaleGroup*, Glyph*) const;
MenuItem* menu_item_separator() const;

Glyph* menubar_look() const, pulldown_look() const, pullright_look() const;
Glyph* menubar_item_look(Glyph*, TelltaleState*) const;
Glyph* menu_item_look(Glyph*, TelltaleState*) const;
Glyph* check_menu_item_look(Glyph*, TelltaleState*) const;
Glyph* radio_menu_item _look(Glyph*, TelltaleState*) const;
Glyph* menu_item_separator_look() const;

interface MenuItem : Observer {
  MenuItem(Glyph*, TelltaleState*);
  Glyph* body() const;
  TelltaleState* state() const;
  void action(Action*);
  Action* action() const;
  void menu(Menu*, Window* = nil);
  Menu* menu() const;
  Window* window() const;
};

interface Menu : InputHandler {
  Menu(Glyph*, Style*, float x1, float y1, float x2, float y2);
  void append_item(MenuItem*);
  void prepend_item(MenuItem*);
  void insert_item(GlyphIndex, MenuItem*);
  void remove_item(GlyphIndex);
  void replace_item(GlyphIndex, MenuItem*);
  GlyphIndex item_count() const;
  MenuItem* item(GlyphIndex) const;
  void select(GlyphIndex);
  GlyphIndex selected() const;
};
```

Figure 9.5: WidgetKit menu operations, menu, and menu item protocols.

```
Glyph* hslider(Adjustable*) const;
Glyph* hscroll_bar(Adjustable*) const;
Glyph* vslider(Adjustable*) const;
Glyph* vscroll_bar(Adjustable*) const;
Glyph* panner(Adjustable*, Adjustable*) const;

Stepper* enlarger(Adjustable*) const;
Stepper* reducer(Adjustabel*) const;
Stepper* up_mover(Adjustable*) const;
Stepper* down_mover(Adjustable*) const;
Stepper* left_mover(Adjustable*) const;
Stepper* right_mover(Adjustable*) const;

Glyph* slider_look(DimensionName, Adjustable*) const;
Glyph* scroll_bar_look(DimensionName, Adjustable*) const;
Glyph* panner_look(Adjustable*, Adjustable*) const;
Glyph* enlarger_look(TelltaleState*);
Glyph* reducer_look(TelltaleState*);
Glyph* up_mover_look(TelltaleState*);
Glyph* down_mover_look(TelltaleState*);
Glyph* left_mover_look(TelltaleState*);
Glyph* right_mover_look(TelltaleState*);
```

Figure 9.6: WidgetKit adjuster operations

# Chapter 10

# DialogKit

Whereas WidgetKit provides operations for creating common look-and-feel components such as scrollbars, menus, and buttons, DialogKit is creates higher-level dialog objects. The current implementation of DialogKit provides only field editors and file choosers, but in the future this kit will provide operations for creating confirmers, quit dialogs, and information messages. Figure 10.1 shows the operations provided by DialogKit.

## 10.1 Field editor

Many application need simple editors for entering or browsing data. A field editor is suitable for incorporating into other components, such as a dialog box. Figure 10.2 shows the field editor class interface.

Clicking inside the editor (or calling FieldEditor::edit) initiates an edit. Subsequent keyboard events, regardless of the pointer location, are interpreted as editing operations on the text. Clicking outside the editor terminates the editing.

Text is selected with the pointer or with the keyboard. A single click of the left 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 current selection, the replaced text is discarded, and the selection becomes an insertion point following the inserted character. The commands defined are:

character-left (^B)
character-right (^F)
beginning-of-text (^A)
end-of-text (^E)
erase (^H or DEL)
delete (^D)
select-all (^U)
select-word(^W)

Strings that are too long to fit into the editor can be scrolled horizontally. Clicking the middle button inside the editor initiates “grab-scrolling”. While the button is down the editor scrolls the text to follow the pointer, giving the appearance that the user is dragging the text. Clicking the right button engages “rate-scrolling”, a joy-stick-like scrolling interface in which the scrolling rate increases as the user drags the pointer away from the initial click location.

The field editor constructor is passed a sample string, used to compute its

```
interface DialogKit {
   static DialogKit* instance();
   FieldEditor* field_editor(
      const String& sample, Style*, FieldEditorAction* = nil
   ) const;
   FileChooser* file_chooser(
      constString& dir, Style*, FileChooserAction* = nil
   ) const;
};
```

Figure 10.1: DialogKit operations

natural size, a style for customization, and a field editor action to execute when editing returns. Editing normally completes when a carriage return or tab is entered; abnormally when an escape character is entered.

FieldEditor::field sets the contents of the string being edited. FieldEditor::select sets the insertion point or subrange within the edit string. FieldEditor::edit initiates an edit. Specifying a string and selection range is short-hand for first calling FieldEditor::field and FieldEditor::select. FieldEditor::text returns the current value of the string being edited. The caller must copy the string to save the contents, as it will be modified by a subsequent edit.

```
interface FieldEditorAction Resource {
   void execute(FieldEditor*, boolean accept);
};

interface FieldEditorCallback(T) : FieldEditorAction {
   FieldEditorCallback(T)(T*, void (T::*)(FieldEditor*, boolean accept));
};

interface FieldEditor : InputHandler {
   FieldEditor(const char* sample, Style*, FieldEditorAction* = nil);
   FieldEditor(const String& sample, Style*, FieldEditorAction* = nil);

   void field(const char*);
   void field(const String&);
   void select(int pos);
   void select(int left, int right);
   void edit();
   void edit(const char*, int left, int right);
   void edit(const String&, int left, int right);
   const char* text() const;
   void text(String&) const;
};
```

Figure 10.2: Field editor class interface.

```
interface DIalog : InputHandler {
  Dialog(Glyph*, Style*);
  boolean post_for(Window*, float = 0.5, float = 0.5);
  boolean post_at(Coord x, Coord y, float = 0.5, float = 0.5);
  boolean run();
  void dismiss(boolean accept);
};
```

Figure 10.3: Dialog protocol

# 10.2 Dialog

A dialog is an object that can be posted temporarily and grabs input until it is dismissed. Figure 10.3 shows the Dialog protocol. Dialog::post_for creates a transient window for the dialog and aligns over the given window using the given x and y alignment. The default is for the dialog to be centered over the window.

Dialog::post_at creates a top-level window aligned around the given position. Dialog::run blocks until Dialog::dismiss is called, and Dialog::run returns the value of the parameter that is passed to Dialog::dismiss.

# 10.3 FileChooser

A file chooser is a dialog subclass that allows the user to select a file in a directory and returns the file name. Figure 10.4 shows the FileChooser protocol.

```
interface FileChooserAction : Resource {
  void execute(FileChooser*, boolean accept);
};
interface FileChooserCallback(T) : FileChooserAction {
  FileChooserCallback(T)(T*, void (T::*)(FileChooser*, boolean accept));
};
interface FileChooser : Dialog {
  const String* selected() const;
  void reread();
};
```

Figure 10.4: FileChooser protocol

# Chapter 11

# LayoutKit

Composite glyphs usually manage the physical layout of their children. LayoutKit provides operations for creating objects that are useful in managing the arrangement of one or more glyphs. The LayoutKit objects are modelled after the Knuth's TeX document processing system.

## 11.1 Boxes

A box is a polyglyph that uses a layout object to arrange its components. A layout object is not a glyph, but helps a glyph manage the requests and allocations of component glyphs. Figure 11.1 shows the Layout protocol and the LayoutKit operations that create boxes.

Layout::request is given an array of requisitions for the individual components and computes a single requisition for the result. Layout::allocate is given the requisitions of the components and the overall allocation, returning the allocations of the individual components.

A box can be constructed either with a list of up to 10 initial glyph components, or an initial size estimate on the number of components. The size is not a maximum, but can avoid growing the list dynamically.

LayoutKit::hbox returns a box that tiles its components in the X dimension left-to-right and aligns the component origins in the Y dimension. LayoutKit::vbox returns a box that tiles top-to-bottom and aligns in the X dimension. LayoutKit::overlay aligns in both the X and Y dimensions, drawing the components in back-to-front order.

LayoutKit::hbox returns a box with its X origin at the left side of the box; LayoutKit::vbox returns a box with its Y origin at the top of the box. Sometimes it is more convenient to have the origin correspond to the origin of the first component. For example, a column of text might want to have its Y origin correspond to the base line of the first line of text. When the aligment should correspond to the first component's alignment, one can use LayoutKit::hbox_first_aligned or LayoutKit::vbox_first_aligned.

A *deck* is a polyglyph similar in some ways to an overlay box. However, instead of overlaying its components, a deck is a polyglyph where only one of the components is visible at any time. Figure 11.2 shows the deck protocol and the LayoutKit operations to create decks.

Deck::flip_to sets which component should currently be visible; Deck::card returns the current top. The natural size of a deck is the maximum size of the natural sizes of its components. A deck can be stretched to be as large as its largest fully-stretched component.

```
interface Layout {
   void request(GlyphIndex count, const Requisition*, Requisition& result);
   void allocate(
      const Allocation& given, GlyphIndex count, const Requisition*,
      Allocation* result
   );
};

PolyGlyph* box(Layout*, GlyphIndex size = 10) const;
PolyGlyph* hbox(GlyphIndex size) const;
PolyGlyph* hbox(
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
) const;
PolyGlyph* vbox(GlyphIndex size) const;
PolyGlyph* vbox(
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
) const;
PolyGlyph* hbox_first_aligned(GlyphIndex size) const;
PolyGlyph* hbox_first_aligned(
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
) const;
PolyGlyph* vbox_first_aligned(GlyphIndex size) const;
PolyGlyph* vbox_first_aligned(
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
) const;
PolyGlyph* overlay(
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
) const;
```

Figure 11.1: Layout protocol and LayoutKit box operations

## 11.2 Glue

Boxes stretch or shrink their components to fit the available space. However, many components are rigid. For example, characters in text typically do not stretch or shrink. Instead, we prefer to stretch or shrink the white space between words. In TeX, this flexible white space is referred to as "glue".

LayoutKit provides the set of operations to create glue object shown in Figure 11.3. LayoutKit::glue is the most general operation. It can either be passed a specific dimension, natural size, stretchability, shrinkability, and alignment, or it can be passed a complete requisition.

LayoutKit::hglue and LayoutKit::vglue can create glue that is horizontally or vertically stretchable, respectively. The requirement in the minor dimension is undefined. If no parameters are passed to LayoutKit::hglue or vglue, then a natural size of zero is assumed. If no stretchability is specified, then the glue is

```
interface Deck : PolyGlyph {
   GlyphIndex card() const;
   void flip_to(GlyphIndex);
};

Deck* deck(GlyphIndex size) const;
Deck* deck(
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
   Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
) const;
```

Figure 11.2: Deck protocol and LayoutKit operations

assumed to be infinitely stretchable.

LayoutKit::hspace and LayoutKit::vspace create glue with a given size that is rigid. LayoutKit::shape_of returns glue that has the same requisition as another glyph. LayoutKit::shape_of_xy returns glue that takes its X requirement from one glyph and its Y requirement from another glyph.

LayoutKit::spaces returns glue that correspond to a given number of spaces in

```
Glyph* glue(
   DimensionName, Coord natural, Coord stretch, Coord shrink, float alignment
) const;
Glyph* glue(const Requisition&) const;
Glyph* hglue() const;
Glyph* hglue(Coord natural) const;
Glyph* hglue(Coord natural, Coord stretch, Coord shrink) const;
Glyph* hglue(Coord natural, Coord stretch, Coord shrink, float alignment) const;

Glyph* vglue() const;
Glyph* vglue(Coord natural);
Glyph* vglue(Coord natural, Coord stretch, Coord shrink) const;
Glyph* vglue(Coord natural, Coord stretch, Coord shrink, float alignment) const;

Glyph* hspace(Coord) const;
Glyph* vspace(Coord) const;
Glyph* shape_of(Glyph*) const;
Glyph* shape_of_xy(Glyph*, Glyph*) const;

Glyph* spaces(int count, Coord each, const Font*, const Color*) const;
Glyph* strut(
   const Font*, Coord natural = 0, Coord stretch = 0, Coord shrink = 0
) const;
Glyph* hstrut(
   Coord right_bearing, Coord left_bearing = 0,
   Coord natural = 0, Coord stretch = 0, Coord shrink = 0
) const;
Glyph* vstrut(
   Coord ascent, Coord descent = 0,
   Coord natural = 0, Coord stretch = 0, Coord shrink = 0
) const;
```

Figure 11.3: LayoutKit operations for creating spacing glyphs

```
MonoGlyph* center(Glyph*, float x = 0.5, float y = 0.5) const;
MonoGlyph* center_dimension(Glyph*, DimensionName, float align) const;
MonoGlyph* hcenter(Glyph*, float x = 0.5) const;
MonoGlyph* vcenter(Glyph*, float y = 0.5) const;

MonoGlyph* fixed(Glyph*, Coord x, Coord y) const;
MonoGlyph* fixed_dimension(Glyph*, DimensionName, Coord) const;
MonoGlyph* hfixed(Glyph*, Coord x) const;
MonoGlyph* vfixed(Glyph*, Coord y) const;
MonoGlyph* flexible(Glyph*, Coord stretch = fil, Coord shrink = fil) const;
MonoGlyph* flexible_dimension(
   Glyph*, DimensionName, Coord stretch = fil, Coord shrink = fil
) const;
MonoGlyph* hflexible(
   Glyph*, Coord stretch = fil, Coord shrink = fil
) const;
MonoGlyph* vflexible(
   Glyph*, Coord stretch = fil, Coord shrink = fil
) const;
MonoGlyph* natural(Glyph*, Coord x, Coord y) const;
MonoGlyph* natural_dimension(Glyph*, DimensionName, Coord) const;
MonoGlyph* hnatural(Glyph*, Coord) const;
MonoGlyph* vnatural(Glyph*, Coord) const;
```

Figure 11.4: LayoutKit operations that adjust alignment

the current font. Unlike other glue objects, which despite being generally referred to as “white space” do not actually have an appearance, spaces are drawn in the given color.

# 11.3 Alignment

In addition to laying out a collection of glyphs, it is often desirable to modify the positioning of a single glyph. LayoutKit provides operations to adjust the layout of a glyph. These operations return a monoglyph.

Figure 11.4 shows the operations that affect the requisition of a glyph. LayoutKit::center , center_dimension, hcenter, and vcenter change the origin of a glyph as it appears in a requisition. When the glyph body is allocated, it is given the origin it requested. Thus, the name “center” is somewhat misleading as these monoglyphs merely return a glyph that asks to be centered at a particular position, they do not actually change the origin themselves.

LayoutKit::fixed, fixed_dimension, hfixed, and vfixed change a glyph to appear rigid even if it is flexible. Fixed monoglyphs are ideal for specifying sizes that are otherwise undefined, such as the initial height of a file chooser. The inverse functionality–making a glyph flexible that may be rigid–is provided by the LayoutKit::flexible, flexible_dimension, hflexible, and vflexible operations. The LayoutKit::natural, natural_dimension, hnatural, and vnatural operations are

```
MonoGlyph* margin(Glyph*, Coord) const;
MonoGlyph* margin(Glyph*, Coord hmargin, Coord vmargin) const;
MonoGlyph* margin(
  Glyph*, Coord lmargin, Coord rmargin, Coord bmargin, Coord tmargin
) const;
MonoGlyph* margin(
  Glyph*,
  Coord lmargin, Coord lstretch, Coord lshrink,
  Coord rmargin, Coord rstretch, Coord rshrink,
  Coord bmargin, Coord bstretch, Coord bshrink,
  Coord tmargin, Coord tstretch, Coord tshrink
) const;
MonoGlyph* hmargin(Glyph*, Coord) const;
MonoGlyph* hmargin(Glyph*, Coord lmargin, Coord rmargin) const;
MonoGlyph* hmargin(
  Glyph*,
  Coord lmargin, Coord lstretch, Coord lshrink,
  Coord rmargin, Coord rstretch, Coord rshrink
) const;
MonoGlyph* vmargin(Glyph*, Coord) const;
MonoGlyph* vmargin(Glyph*, Coord lmargin, Coord rmargin) const;
MonoGlyph* vmargin(
  Glyph*,
  Coord bmargin, Coord bstretch, Coord bshrink,
  Coord tmargin, Coord tstretch, Coord tshrink
) const;
MonoGlyph* lmargin(Glyph*, Coord) const;
MonoGlyph* lmargin(Glyph*, Coord nat, Coord stretch, Coord shrink) const;
MonoGlyph* rmargin(Glyph*, Coord) const;
MonoGlyph* rmargin(Glyph*, Coord nat, Coord stretch, Coord shrink) const;
MonoGlyph* bmargin(Glyph*, Coord) const;
MonoGlyph* bmargin(Glyph*, Coord nat, Coord stretch, Coord shrink) const;
MonoGlyph* tmargin(Glyph*, Coord) const;
MonoGlyph* tmargin(Glyph*, Coord nat, Coord stretch, Coord shrink) const;
```

Figure 11.5: LayoutKit margin operations

similar to the fixed operations in that they change the natural size, but they do not affect the flexibility.

LayoutKit also provides a set of operations to put a margin around a glyph. These operations are shown in Figure 11.5. LayoutKit::margin is overloaded to specify a fixed margin around the entire glyph, distinct horizontal and vertical margins, separate left, right, bottom, and top margins, or flexible margins on each side. LayoutKit::hmargin specifies horizontal margins; LayoutKit::vmargin specifies vertical margins. LayoutKit::lmargin, rmargin, bmargin, and tmargin specify left, right, bottom, and top margins, respectively.

# Chapter 12

# DocumentKit

In Chapter 11, we described the LayoutKit class, which provides operations for creating layout objects. The DocumentKit class will provide operations for creating document objects that use LayoutKit objects to produce formatted documents. The DocumentKit class has not yet been implemented, so for now we describe several objects that can be useful for building document editors in conjunction with the LayoutKit objects.

A *discretionary* can take on one of several appearances depending on whether a break occurs on it. A common use of a discretionary is for white space in a line of text, where the white space becomes zero-width glue if a break occurs on the discretionary. The penalty associated with a discretionary defines the relative cost of breaking. Currently, operations to create discretionaries are defined on the LayoutKit.

An *lr-marker* is a glyph that can mark a region of its body. The marking is done by painting a color under the area or a color on top of the area (or both). The region is a shape normally associated with text selections. The LRMarker constructor takes two colors, either of which can be nil if the associated underlay or overlay drawing is not desired. LRMarker::mark paints the given region. If y1 and y2 are the same, then the mark region is a single rectangle. Otherwise, the region is defined as starting at (x1,y1) and filling a height of h1 to (x1,right()), then filling (left(),y1+h1) to (right(),y2) and filling a height of h2 from (left(),y2) to (x2,y2). LRMarker::unmark restores the area to its unmarked appearance. An *xy-marker* is simliar to an lr-marker, but it only paints a rectangular area. While an lr-marker is most useful for selecting text, xy-markers are useful for selecting an item in a list or table. Figure 12.1 shows the LRMarker and XYMarker class interfaces.

## 12.1 Compositions

A *composition* is a glyph that uses a *compositor* to determine suitable breaks between groups of its components. Figure 12.2 shows the class interfaces for the composition and its subclasses. The list of components is broken into sublists that are put into separate composite glyphs and then inserted into the body. The LRComposition subclass uses LayoutKit::hbox_first_aligned to create each sublist, while the TBComposition subclass uses LayoutKit::vbox_first_aligned.

Compositions can be used to break paragraphs into lines, lines into columns, or columns into pages. A document editor might create an lr-composition for characters that puts the resulting hboxes for lines into a tb-composition, which in turn puts the vboxes for columns into an lr-composition, which puts the resulting

```
interface LRMarker : MonoGlyph {
   LRMarker(Glyph*, Color* overlay, Color* underlay);
   void mark(
      Coord left, Coord right, Coord x1, Coord y1,
      Coord h1, Coord x2, Coord y2, Coord h2
   );
   void unmark();
};

interface XYMarker : MonoGlyph {
   XYMarker(Glyph*, Color* overlay, Color* underlay);
   void mark(Coord left, Coord bottom, Coord right, Coord top);
   void unmark();
};
```

Figure 12.1: LRMarker and XYMarker protocols

lr-boxes for pages into a deck.

Compositions also can be used just as easily for arranging buttons in a box, where one wants the buttons to ‘‘wrap-around’’ if there are too many to fit horizontally. The code to do this could look as follows:

```
LRComposition* c = new LRComposition(
   layout.vbox(), new SimpleCompositor, /* no separator */ nil, /* width */ 4*72.0
);
Discretionary* ok = layout.discretionary(0, nil, nil, nil, nil);
for (unsigned int i = 0; i < nbuttons; i++) {
   c->append(button[i]);
   c->append(ok);
}
c->repair();
```

The composition constructor takes a body in which to insert sublists, a compositor to determine where to break, a separator to be inserted at each break (unless it is nil), the dimension to use, the width in which the sublists must fit, and optionally the initial size of the list of components. The initial size is not a maximum, but the list by default starts at a small size and is dynamically reallocated as needed. For large lists, it can be more efficient to specify an estimate of the list size. Composition::repair updates the composition to reflect changes to its structure. Composition::item returns the index of the sublist (in other words, the component of the body) containing the component specified by the given index. For example, this function could be used to return the line that contains a particular character. Composition::beginning_of returns the index of the component that starts the sublist specified by the given item. For example, this function could be used to return the character that starts a particular line. Composition::end_of is like beginning_of except it returns the end of the sublist. Composition::margin causes space to be left at the beginning and end of the specified item in the body. For example, this function could be used to put margins on a line of text. Composition::view guarantees that breaks are computed

```
interface Composition : MonoGlyph {
  Composition(
    Glyph*, Compositor*, Glyph* separator, DimensionName,
    Coord width, GlyphIndex size
  );
  void repair();
  GlyphIndex item(GlyphIndex);
  lyphIndex beginning_of(GlyphIndex);
  GlyphIndex end_of(GlyphIndex);
  void margin(GlyphIndex, Coord begin, Coord end);
  void view(GlyphIndex first, GlyphIndex last);
};

interface LRComposition : Composition {
  LRComposition(
    Glyph*, Compositor*, Glyph* sep, Coord, GlyphIndex = 10
  );
};

interface TBComposition : Composition {
  TBComposition(
    Glyph*, Compositor*, Glyph* sep, Coord, GlyphIndex = 10
  );
};
```

Figure 12.2: Composition classes

for the components between indices *first* and *last* inclusively. By restricting the viewing area, this operation can eliminate the computation of breaks for components that are not visible.

A compositor computes the breaks based on assessing the penalty for a possible breaks. Three subclasses are provided that use different levels of sophistication in determining breaks. An *array-compositor* positions breaks every N elements, where N is specified in the constructor. A *simple-compositor* finds a simple set of breaks quickly. It is analogous to a line-at-a-time text formatter. A *TeX-compositor* finds breaks using Knuth's TeX algorithm.

Figure 12.3 shows the interfaces to the compositor classes. Compositor::compose uses the natural, stretch, shrink, penalty, and span information for each component as input parameters. The return value is the number of breaks found. The breaks array contains the positions of the breaks; that is, break[i] is the index of the component where the *i*th break occurs.

```
interface Compositor {
  virtual int compose(
    Coord* natural, Coord* stretch, Coord* shrink,
    int* penalties, int component_count,
    Coord* spans, int span_count,
    int* breaks, int break_count
  );
};

interface ArrayCompositor : Compositor {
  ArrayCompositor(int N);
};

interface SimpleCompositor : Compositor {
  SimpleCompositor();
};

interface TeXCompositor : Compositor {
    TeXCompositor(int penalty);
};
```

Figure 12.3: Compositor classes

See iv/src/examples/preview.

# Appendix A

# Operating System Interface

This appendix describes the classes that abstract operating system services. These classes do not yet cover the entire range of operating system operations, but they offer higher-level abstractions in some cases, relief from name space concerns in other cases, and greater application portability.

The classes are presented below in alphabetical order. A *directory* is a list of files or other directories. A file is a list of bytes. Both directories and files are accessed by a string name. A *list* is an ordered sequence of objects, parameterized by the element type. The math class provides an interface to common mathematical operations on concrete types (e.g., ints and floats). The memory class provides operations on variable-length arrays of bytes. A *string* is a variable-length list of characters. A *table* is an associative map from a key type to a data type.

List and table are generic classes, parameterized by other types. Because few C++ implementations currently support a generic mechanism (though the language defines a template construct), it is necessary to define list and table classes using preprocessor macros and must be instantiated explicitly.

## A.1 Directory

Figure A.1 shows the directory class interface. The static member functions Directory::current and Directory::open return a pointer to a Directory. Directory::current looks in the default directory for the application context, Directory::open tries to find the directory with the given name. If the name cannot be found or access is denied to the directory, Director::open returns nil.

```
interface Directory {
   static Directory* current();
   static Directory* open(const String&);
   int count() const;
   const char* name(int i) const;
   int index(const char*) const;
   boolean is_directory(int index) const;
   void close();
   static String* canonical(const String&);
   static boolean match(const String& name, const String& pattern);
};
```

Figure A.1: Directory class interface.

Directory::count returns the number of entries (files and directories), including any special system entries such as “.” and “..” on Unix. Directory::name returns the name of the specified entry. Directory::index returns the index of the entry that matches the given name or -1 if no match is found. Directory::close discards the information associated with the directory.

## A.2 File

Figure A.2 shows the file class interface and subclasses for input handling. The base class File is abstract and defines no data access functions. Currently, only input files are implemented. File::name returns the string name of the file. File::length returns the number of bytes in the file. File::close releases any information associated with the file.

File::limit sets an upper bound on the size of a buffer to allocate for file storage. By default, a file attempts to allocate contiguous storage for its entire contents. This approach can allow for simpler application code and can be implemented very efficiently on systems with memory-mapped files.

The contents of an input file can be accessed but not modified. InputFile::open returns nil if the named file cannot be found or is not readable. InputFile::read allocates a data area for the file contents, sets the start parameter to the beginning of the area, and returns the length of the area. If no storage limit has been specified and the file is on disk (as opposed to a terminal or pipe), then read will return the entire file.

## A.3 List

Figure A.3 shows the list generic class interfaces. The implementation of lists uses a dynamic array with an insertion gap, meaning that large lists are very

```
interface File {
    const char* name() const;
    long length() const;
    void close();

    void limit(unsigned int buffersize);
};

interface InputFile : File {
    static InputFile* open(const char* name);
    int read(const char*& start);
};
```

Figure A.2: InputFile protocols.

```
interface List(T) {
   List(T)(long initial_size = 0);

   long count() const;
   T item(long index) const;
   T& item_ref(long index) const;
   void prepend(const T&);
   void append(const T&);
   void insert(long index, const T&);
   void remove(long index);
   void remove_all();
};

interface ListItr(ListType) {
   ListItr(ListType))(const ListType&);

   boolean more() const;
   T cur() const;
   T& cur_ref() const;
   void next();
};

interface ListUpdater(ListType) {
   ListUpdater(ListType)(ListType&);

   boolean more() const;
   T cur() const;
   T& cur_ref() const;
   void remove_cur();
   void next();
};
```

Figure A.3: List and iterator class interfaces.

space-efficient. The time efficiency depends on the distribution of insertions–if the insertion position moves frequently, the list will do excessive copying.

The list constructor takes an optional initial size for the dynamic array. For lists that are known to be large, specifying an initial size avoids the cost of growing the array dynamically. List::count returns the number of elements in the list (not the size of the array). List::item returns the indexed item.

List::prepend adds an item at the beginning of the list, List::append at the end, and List::insert before an indexed item. List::remove deletes the item specified by the given index. List::remove_all deletes all the items in the list.

ListItr is a class for iterating through the elements of a list, parameterized explicitly by the list type and implicitly by the element type. The constructor is given the target list. ListItr::more returns true if additional elements are available in the iteration. ListItr::cur returns the current iteration’s item. ListItr::cur_ref returns a reference to the item. ListItr::remove_cur deletes the current item from the list. ListItr::next moves the iteration to the next item in the list. No modifications should be made to the list during an iteration (except in the last step), as this may cause unexpected or erroneous results.

As an example, consider a list of pointers to glyphs. We could declare such a list in a header or source file as follows:

```
declarePtrList(GlyphList,Glyph)
```

Only one source file may contain the expansion of the implementation:

```
implementPtrList(GlyphList,Glyph)
```

A loop to iterate over all the glyphs in a list could be written as follows:

```
for (ListItr(GlyphList) i(list); i.more(); i.next()) {
    Glyph* g = i.cur();
    // do something with g
}
```

A list makes no assumptions about its element type. In particular, destroying a list of pointers will not destroy the objects that are the targets of the pointers.

## A.4 Math

Figure A.4 shows the math class interface. One cannot create a “math object”; all the member functions are static. Math::min and Math::max return the minimum and maximum of two or four numbers, respectively. For the sake of brevity only the floating point definitions are shown, but Math::min and Math::max are also defined for ints, longs, unsigned ints, unsigned longs, and doubles.

Math::abs returns the absolute value of a number. Math::round returns the integer nearest to a floating point value. Math::equal compares two floating point numbers and returns whether they are within a given epsilon (the third parameter) of each other.

## A.5 Memory

Figure A.5 shows the memory class interface. Like the math class, the memory class consists solely of static member functions. Memory::copy writes a specified number of bytes from one memory location to another. Memory::compare determines if a specified number of bytes at one memory location is identical to those at another location. If so, Memory::compare returns 0. Otherwise, it returns a non-zero value. Memory::zero sets a specified number of bytes to zero starting at a given memory location.

In certain circumstances, memory operations are faster than a loop over a set of elements. Memory::zero and Memory::copy are useful for implementing a dynamic array, quickly clearing or copying data when the array grows.

```
interface Math {
   static float min(float a, float b);
   static float max(float a, float b);
   static float min(float a, float b, float c, float d);
   static float max(float a, float b, float c, float d);

   static int abs(int);
   static long abs(long);
   static double abs(double);
   static int round(float);
   static int round(double);
   static boolean equal(float x, float y, float e);
   static boolean equal(double x, double y, double e);
};
```

Figure A.4: Math class interface.

## A.6 String

Figure A.6 shows the string class interface. The purpose of the string class is to provide a convenient set of operations for manipulating variable-length character arrays, *not* to manage storage. The base class does not allocate or free any storage associated with the characters.

Three string constructors are available. The first, with no arguments, creates an uninitialized string that should be assigned to another string before use. The second, with a character pointer, sets the string's data to the given pointer. The string's length is computed from the pointer under the assumption that the data is null-terminated. The third constructor takes a character pointer and explicit length. It does not assume the data is null-terminated.

String::string returns a pointer to the character data, which may not be null-terminated. String::length returns the number of characters in the string. String::null_terminated returns whether the string is already known to be null-terminated (it does not attempt to find a null). String::hash returns a value for the string data suitable for indexing the strings into a hash table.

The string class provides operators for assignment and comparison. The second operand for these operations can be a string or a character pointer. In the latter case, the data is assumed to be null-terminated. String::case_insensitive_equal tests for equality ignoring the case of the characters in the strings.

```
interface Memory {
   static void copy(const void*, void* to, unsigned int nbytes);
   static int compare(const void*, const void*, unsigned int nbytes);
   static void zero(void*, unsigned int nbytes);
};
```

Figure A.5: Memory class interface.

The subscript operator allows access to individual characters. It is an error to pass a negative index or an index greater than or equal to the length of the string. String::substr returns a new string representing the part of the original string begining at the *start* parameter and continuing for *length* characters. If *start* is negative, the beginning position is the end of the string offset by *start*. If *length* is –1, then the remainder of the string is included. String::substr does not copy the data, it simply creates another string that points into the same data as the original string. String::left and String::right are short-hand for accessing the beginning or end of a string. String::set_to_substr, String::set_to_left, and String::set_to_right are convenience for changing a string to a particular substring instead of creating a new string.

String::search returns the index of the occurrence of the given character after

```
interface String {
  String();
  String(const char*);
  String(const char*, int length);

  const char* string() const;
  int length() const;
  boolean null_terminated() const;
  unsigned long hash() const;

  String& operator =(const String&);
  boolean operator ==(const String&) const;
  boolean operator !=(const String&) const;
  boolean operator >(const String&) const;
  boolean operator >=(const String&) const;
  boolean operator <(const String&) const;
  boolean operator <=(const String&) const;
  boolean case_insensitive_equal(const String&) const;

  char operator[](int index) const;
  String substr(int start, int length) const;
  String left(int length) const;
  String right(int start) const;
  void set_to_substr(int start, int length);
  void set_to_left(int length);
  void set_to_right(int start);

  int search(int start, char) const;
  int index(char) const;
  int rindex(char) const;

  boolean convert(int&) const;
  boolean convert(long&) const;
  boolean convert(float&) const;
  boolean convert(double&) const;
};
```

Figure A.6: String class interface.

the given starting position. If the starting position is negative, it is treated as an offset from the end of the string and the search is made right-to-left. String::index and String::rindex are short-hand for searching from the beginning and end of the string, respectively.

String::convert attempts to interpret the string as a number and sets its parameter to the value. If the conversion is successful, String::convert returns true.

Three string subclasses of string are provided, all of which have the same constructors and operations as the base class. CopyString is a subclass that copies the string data when constructed and frees the storage when deleted. When the copy is made, a null is appended to ensure the data is null-terminated. NullTerminatedString is a subclass that guarantees its data is null-terminated. If constructed with a normal string, it will copy the data much like a copy-string. However, if the given string is already a copy-string, then no copy is made. NullTerminatedString is useful for passing string data to external C functions, such as printf.

The third string subclass is UniqueString, which uses a table to map identical strings to the same data. Comparing unique strings for equality is fast because the implementation can compare pointers instead of the string data. Unique strings are not null-terminated.

## A.7 Table

Figure A.7 shows the table class interface. Table is a generic class that is parameterized by a key type and a value type. The constructor is given a size for the hash table implementation. For good access performance, the size should be roughly twice the expected number of keys.

Table::insert stores a <key,value> pair. Table::find searches an entry with the given key. If such an entry exists, Table::find sets *value* and returns true. Otherwise, it leaves the parameter unmodified and returns false. Table::remove deletes a <key,value> pair from the table if one exists. Table::find_and_remove combines the find and remove operations in a single call.

If the same key is inserted more than once, Table::find will return the most recently inserted value. Similarly, Table::remove will delete the most recently inserted pair.

TableIterator allows one to iterate over all the <key,value> pairs defined in a table. TableIterator is parameterized explicitly by the table type, implicitly by the key and value types. TableIterator::cur_key and TableIterator::cur_value return the current entry information. TableIterator::more tests if additional entries are defined. TableIterator::next moves to the next entry in the table.

```
unsigned long key_to_hash(Key);

interface Table(Key,Value) {
  Table(Key,Value)(int hash_table_size);

  void insert(Key, Value);
  boolean find(Value&, Key);
  void remove(Key);
  boolean find_and_remove(Value&, Key);
};

interface TableIterator(Table(Key,Value)) {
  TableIterator(Table(Key,Value))(Table(Key,Value)&);

  Key& cur_key();
  Value& cur_value();
  boolean more();
  boolean next();
};
```

Figure A.7: Table class interface.