InterViews Reference Manual
Version 3.0

Draft: April, 1991

Mark A. Linton

Paul R. Calder

John M. Vlissides

Chapter 1

Introduction

InterViews is a system for building and using interactive software. Like most interactive systems, 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.

This manual is organized into nine chapters. Chapter 1 defines basic terms and provides an overview of the system. Chapter 2 describes the use of objects called *glyphs* to define the presentation of a user interface. Chapter 3 defines the objects that a glyph can use to draw on a screen or printer. Chapter 4 describes how glyphs are associated with a window on the screen and how InterViews objects communicate information to an external window manager. Chapter 5 explains how input is handled. Chapter 6 defines how characteristics of a group of objects can be controlled from a common property sheet. Chapter 7 describes typical objects for controlling a user interface, including menus, scrollbars, and buttons. Chapter 8 describes objects for representing and editing documents. Chapter 9 describes the classes that interface to the operating system.

1.1 Notation

We use C++ syntax to describe the different classes of InterViews objects. We assume the reader is somewhat familiar with C++ semantics, but the specification of a class in this manual is *not* identical to its C++ class declaration in a header file. In particular, we do not list the private sections of a class declaration because they are actually part of the implementation, not the interface. We also do not list members that are implicitly part of the implementation rather than the semantics of a classs. For example, C++ destructors are normally public but often simply free storage allocated by the object. Thus, there is no need to list and document destructors as part of a class interface.

A more subtle example is a virtual function implemented by a subclass. We only list the function in the subclass interface if the subclass changes the semantics of the function. We do not include PSFont::width, for example, because PSFont just implements the Font::width virtual function using PostScript font metrics instead of screen font metrics. On the other hand, the Listener class actually extends the semantics of Glyph::pick to put the listener on a target list associated with the hit object. Thus, we do include Listener::pick.

1.1.1 Names

We use identifiers that begin with an upper case letter for types; we use lower case for member functions and parameters. We assume that some mechanism allows us to use any names we wish–there is no need to use a special prefix for all global names (types and classes). Ideally, this capability would be provided by the C++ language, but this is not yet the case. Currently, we use #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.

1.1.2 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, a function 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 functions that do not change the behavior of an object. That is, a member function is const if a call to it could be omitted without changing the affect of subsequent function calls on the object. This definition is consistent with the notion that a compiler could eliminate redundant calls to the same const function.

Using our semantics of const, the transformation matrix identity function 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. Current C++ compilers can only guarantee that a function does not affect an object's behavior if the function does not modify storage.

Note that 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 a member function performs some non-const operation on that object, then the member function should be defined as non-const.

1.1.3 Common Symbols

Several definitions are pervasive throughout the system. 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++, nil is a macro defined to zero. These definitions are automatically defined as a side effect of using any other InterViews classes.

1.1.4 Coordinates

Many classes and functions in a toolkit manipulate coordinates. We define a type **Coord** to represent a position or a distance. The default units for a coordinate are printers points. By default, coordinate values are scaled according to screen fonts, not the dots-per-inch of the screen. Thus, the coordinate scaling factor is 75/72 on systems with 75dpi fonts. The scaling factor for a screen can be customized on a per-screen basis.

The reason for using font resolution instead of screen resolution is that most applications want to show geometry as it would appear relative to text, not at its actual size. The problem is that font metric information is given in terms of the font resolution. For example, consider the character "x" drawn using a 75-dpi font next to a rectangle on a 90-dpi screen. If the rectangle is scaled using 90-dpi, then it will look too small relative to the "x" on a printer.

Ideally, Coord would be a C++ class with well-defined operations and conversions to concrete

```
class Resource {
public:
  Resource();
  void ref();
  void unref();
  static void ref(Resource*);
  static void unref(Resource*&);
};
```

Figure 1.1: Resource class interface

types. However, current C++ compilers do not make it practical to define a Coord class that is represented as a single word. We therefore define Coord as a typedef equivalent to a 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 1pt line might generate one or two pixels, depending on where it is on the screen. Objects that need to generate consistent pixel sizes must explicitly scale to pixel units.

1.1.5 Dimensions

Many composition strategies manipulate coordinates in one dimension. To use a single strategy in any dimension, we define an enumerated type **DimensionName** as *Dimension_X*, *Dimension_Y,* or *Dimension_Z.* It is possible that other dimensions may be defined in the future. The enumeration constant *Dimension_Undefined* is defined after any other dimensions. Thus, one can iterate through all dimensions by stopping at Dimension_Undefined.

1.1.6 Reference counting

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. We do not consider resource really part of the InterViews programming interface; 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 implementations must manage references correctly.

Figure 1.1 shows the Resource class interface. Every resource has a reference 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.A resource should not be deleted like other objects; instead, Resource::unref should be used to unreference an object. When a resource is unreferenced, the count is decremented. If the count is not positive, then the object is destroyed. The static ref and unref functions check if their arguments are nil and if not call the non-static ref or unref member functions, respectively. The static unref sets the pointer to nil if the object is destroyed.

1.2 Overview

The goal of InterViews is to make it easy to compose user interfaces from reusable components. The central class for composition is **glyph** because it defines the geometry of a user interface object. 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 glyph 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.

Every application must create a **world** object to denote the destination display. Currently, a world also provides an application main loop and property information. It is expected in the future that the World class will be split into separate classes for the display, screen, and application control.

The InterViews input model is ‘‘pick-oriented’’. That is, input events are received by the appropriate window, as determined by the window manager (using some focus policy). The receiving window does a pick operation on its top-level glyph. Pick operations have two results: a list of glyphs that were picked, and a list of those glyphs that are interested in handling an input event. A glyph that detects input is called a **listener.** The window searches the list of listeners, stopping when one is found that can handled the event. Listeners don’t actually handle events themselves; they return a **handler** that is responsible for taking the appropriate action.

Chapter 2

Glyphs

A glyph is an object that draws. Glyphs are the basic unit for building the presentation side of a user interface. Glyphs define no storage by default and are passed all contextual information for rendering. Therefore, glyphs may be shared and a glyph structure need not be strictly hierarchical; it may be a directed acyclic graph.

There are three kinds of glyph subclasses. *Primitive* glyphs are leaves–they contain no components. *Composite* glyphs contain several components and typically arrange them in some form. *Monoglyphs* contain a single component, and they alter the component's appearance or behavior.

2.1 Glyph base class

Glyph is a subclass of resource because glyphs can be shared. Resource must be a virtual base class to ensure that only one copy of resource information (i.e., reference count) is managed in the case of multiple inheritance. Figure 2.1 shows the glyph base class definition.

```
class Glyph : virtual public Resource {
protected:
  Glyph();
public:
  virtual void request(Requisition&) const;
  virtual void allocate(Canvas*, const Allocation&);
  virtual void draw(Canvas*, const Allocation&) const;
  virtual void print(Printer*, const Allocation&) const;
  virtual boolean pick(Canvas*, const Allocation&, Hit&, unsigned int depth) const;

  enum BreakType { pre_break = 1, in_break, post_break };
  virtual Glyph* compose(BreakType) const;

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

Figure 2.1: Glyph class interface

2.1.1 Geometry management

Glyph::request asks a glyph to specify its desired geometry in the given requisition. It is desirable that this operation be fast so that a composite glyph need not store the requistions for its components. Glyph::allocate notifies a glyph that a portion of the canvas has been allocated for it. A glyph can assume that allocate will be called before any draw operations are called with the same canvas and allocation.

2.1.2 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. A glyph's draw operation therefore should not be called directly, but rather is implicitly called during screen update.

Glyph::print generates a representation of the glyph graph suitable for printing. The canvas and printer rendering interfaces are identical, and the default 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 rectangle specified in canvas-relative coordinates. The rectangle coordinates are contained in the Hit parameter. Pick returns true if at least one glyph is found inside the rectangle. The depth parameter specifies which level in the Hit object to store the hit glyphs. When pick returns, the Hit parameter contains the glyphs that were hit. In addition the hit object contains a list of **listener** glyphs. Listener is a subclass of glyph that represents an output area sensitive to input. Listener::pick simply puts itself on a hit object's target list.

2.1.3 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). The enumeration values are defined to start at one to avoid equating an uninitialized datum (which will be zero) with a valid enumeration constant.

Glyph::compose on a glyph 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.

2.1.4 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. *Glyph::allotment seems to imply that composite glyphs always store the allocations of their components. This requirement is not intended, but in practice many composites will store the allocations. So, one might find that calling allotment is frequently cheap, but this assumption is not valid in general.*

2.1.5 Requisitions

Glyphs define their geometric needs with a **requisition**, which specifies a **requirement** in each dimension. Figure 2.2 shows the requisition and requirement class interfaces. Composite glyphs typically use a layout object to arrange their components in some manner such as a tile or overlay. Layouts are often dimension-independent; for example, there is no algorithmic difference between left-to-right and top-to-bottom tiling. Requisitions can therefore be specified and retrieved independently in each dimension.

Requisiton::Requisition creates a requisition with undefined requirements. 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 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 coordintaes 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.

2.2 Allocations

The actual size a glyph is given is defined by an **allocation**, which specifies an **allotment** in each dimension. Figure 2.3 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, and alignment is 0.5, then the allotment begins at -4 and ends at 6.

```
class Requisition {
public:
  Requisition();
  int penalty();
  void penalty(int);
  void require(DimensionName, const Requirement&);
  const Requirement& requirement(DimensionName);
};

class Requirement {
public:
  Requirement();
  Requirement(Coord natural);
  Requirement(Coord natural, Coord stretch, Coord shrink, float align);
  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);
  boolean defined();
  void natural(Coord);
  Coord natural();
  void stretch(Coord);
  Coord stretch();
  void shrink(Coord);
  Coord shrink();
  void alignment(float);
  float alignment();
};
```

Figure 2.2: Requisition and requirement

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.

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

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

Figure 2.3: Allocation and allotment class interfaces

2.3 Primitive glyphs

Primitive glyphs do not contain component glyphs–they are leaf nodes in a glyph instance graph Figure 2.4 shows the Character, Space, and Label class interfaces. A **character** glyph draws a single character using the font and foreground color defined by the given style sheet. The character code is effectively an index into the font information.

A **space** is represents character spaces. Fonts typically provide a single space character, but document editors need different sizes of spacing for minimum spacing between words and sentences. Unlike glue, a space object can be picked. A **label** is semantically equivalent to an LRBox containing the individual characters in a string.

A **rule** is a line that is parallel to a coordinate axis and is useful for separating objects visually. For convenience, HRule and VRule subclasses are provided for specifying horizontal and vertical lines, respectively.

A **stencil** draws through a bitmap, using the foreground color defined by the property sheet. A bitmap is logically an array of bits that can be zero or one. For elements that are one, the stencil draws in the foreground color. For elements that are zero, the stencil does not draw. An **image** draws a raster. The natural size and origin of a stencil (image) is defined by the dimensions and

```
class Character : public Glyph {
public:
  Character(long code, const StyleSheet*);
  long code() const;
  const StyleSheet* style() const;
};

class Space : public Glyph {
public:
  Space(unsigned int count, Coord each, const StyleSheet*);
  unsigned int count() const;
  Coord each() const;
  const StyleSheet* style() const;
};

class Label : public Glyph {
public:
  Label(const String&, const StyleSheet*);
  const String& stringl() const;
  const StyleSheet* style() const;
};
```

Figure 2.4: Character, Space, and Label class interfaces

bearings of the bitmap (raster).

Figure 2.5 shows the interfaces to the classes that affect composition. 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.

A **glue** object is blank space with explicit stretchability and shrinkability in a single dimension. For convenience, HGlue and VGlue subclasses are provided for horizontal and vertical glue, respectively. A **shape-of** object has no appearance but the identical shape of another glyph. A **strut** is a gap like glue, but an alignment defined by font information.

2.4 Composite glyphs

A composite glyph contains one or more components. Typically, a composite glyph uses a **layout** object to arrange its components in some manner such as a tile or overlay. A layout is not a glyph, but an object that helps glyphs manage the requests and allocations of a group of components. Figure 2.6 shows the class interfaces for layout and its sub classes. Layout is an abstract class, so the constructor is only available to subclasses. Layout::request computes a total requisition given a set of count requisitions for the individual components. Layout::allocate uses the given allocation and individual requisitions to compute the individual allocations.

An **align** object is a layout that aligns all its components origins in a particular dimension. A **superpose** overlays components using two to five other layouts. A **tile** is a layout that arranges components side-by-side in a particular dimension. The components are arranged in increasing

```
class Discretionary : public Glyph {
public:
  Discretionary(int penalty, Glyph*);
  Discretionary(int penalty, Glyph* no, Glyph* pre, Glyph* in, Glyph* post);
};

class Glue : public Glyph {
public:
  Glue(DimensionName, Coord natural);
  Glue(DimensionName, Coord natural, Coord stretch, Coord shrink, float alignment = 0);
  Glue(const Requisition&);
};
class ShapeOf : public Glyph {
public:
  ShapeOf(const Glyph*);
};

class Strut : public Glyph {
public:
  Strut(const StyleSheet*, Coord natural = 0, Coord stretch = 0, Coord shrink = 0);
  const StyleSheet* style() const;
};

class HStrut : public Glyph {
public:
  HStrut(
    Coord right_bearing, Coord left_bearing = 0,
    Coord natural = 0, Coord stretch = 0, Coord shrink = 0
  );
};

class VStrut : public Glyph {
public:
  VStrut(
    Coord ascent, Coord descent =0, Coord natural = 0, Coord stretch = 0, Coord shrink = 0
  );
};
```

Figure 2.5: Primitive glyphs that specify composition information

order; that is, the first component starts at the lowest coordinate in the dimension. For example, Tile(Dimension_X) tiles left-to-right. TileReversed is just like Tile but places components in decreasing order. For example, TileReversed(Dimension_Y) tiles top-to-bottom.

2.4.1 Boxes

A **box** is a composite glyph that arranges its components according to a given layout. For convenience, subclasses are provided for common layouts. Figure 2.7 shows the class interfaces for box and its predefined subclasses. A box can be constructed either with a list of up to 10 glyph components, or an initial_size estimate on the number of components. The size is not a maximum, but can be used to avoid growing the list dynamically. The base class is given a specific layout to use. LRBox uses a tile layout in the X dimension to arrange its components

```
class Layout {
public:
  virtual void request(unsigned long count, const Requisition*, Requisition& result);
  virtual void allocate(
    const Allocation& given, unsigned long count, const Requisition*, Allocation& result
  );
protected:
  Layout();
};

class Align : public Layout {
public:
  Align(DimensionName);
};

class Superpose : public Layout {
public:
  Superpose(Layout*, Layout*, Layout* = nil, Layout* = nil, Layout* = nil);
};

class Tile : public Layout {
  Tile(DimensionName);
};

class TileReversed : public Layout {
public:
  TileReversed(DimensionName);
};
```

Figure 2.6: Layout classes

left-to-right. TBBox uses a tile-reversed layout in the Y dimension to arrange its components top-to-bottom. Overlay uses a superpose layout to place its components in the same position. The components are ordered from back-to-front.

2.4.2 Decks

A **deck** is a composite glyph where only one of the components is visible at any time. The natural size 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. Figure 2.8 shows the class interface for deck. Deck::card returns the index of the component that is currently visible (or -1 if no component is showing). Deck::flip_to makes the component at the given index the visible one.

2.4.3 Pages

A **page** is a composite glyph that contains a background and any number of foreground components. The location of the components is specified explicitly. Components of a page are drawn when the page is drawn, except for components that are explicitly ‘‘hidden’’ using Page::show. Figure 2.9 shows the class interface for page. Page::left, Page::bottom, Page::right, and Page::top return the respective coordinates of the page’s canvas as defined by the page’s most recent allocation. Page::x and Page::y return the origin of the page. Page::move allocates the component specified by the given index with its origin at the given position.

```
class Box : public Glyph {
public:
  Box(Layout*, unsigned int initial_size);
  Box(
    Layout*, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
    Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
  );
};

class LRBox : public Box {
public:
  LRBox(unsigned int initial_size);
  LRBox(
    Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
    Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
  );
};

class TBBox : public Box {
public:
  TBBox(unsigned int initial_size);
  TBBox(
    Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
    Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
  );
};

class Overlay : public Box {
public:
  Overlay(
    Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil,
    Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil, Glyph* = nil
  );
};
```

Figure 2.7: Box classes

Page::location returns the current origin of the specified component. Page::show turns on or off the allocation and drawing of the specified component. Page::showing returns whether the specified component is currently showing.

2.5 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 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, monoglyph's 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.

```
class Deck : public Glyph {
public:
  Deck();
  int card();
  void flip_to(int index);
};
```

Figure 2.8: Deck class interface

```
class Page : public Glyph {
public:
  Page(Glyph* background);
  Coord left(), bottom(), right(), top();
  Coord x(), y();
  void move(unsigned int index, Coord x, Coord y);
  void location(unsigned int index, Coord& x, Coord& y);
  void show(unsigned int index, Coord& x, Coord& y);
  boolean showing(unsigned int index);
};
```

Figure 2.9: Page class interface

Figure 2.10 shows the MonoGlyph base class interface. The constructor is protected because MonoGlyph is an abstract class. A non-nil glyph specifies the initial body. MonoGlyph::body is overloaded to set and return the body.

2.5.1 Decoration

A common use of monoglyphs is to decorate a glyph. Decoration could also be implemented by putting the body and a glyph that draws the decoration into an overlay. However, an overlay is a composite glyph, so it interprets structure operations as manipulating the components of the overlay. With this approach, one could not pass a decorated glyph to another object that wants to change the structure of the body. It is therefore more convenient to decorate glyphs using a monoglyph.

Figure 2.11 shows the interfaces to the Background and Border classes. A **background** glyph draws a solid color under its body. Background objects free other glyphs from being concerned about refreshing the background when they are drawn and when used near the top of

```
class MonoGlyph : public Glyph {
protected:
  Glyph();
public:
  virtual void body(Glyph*);
  virtual Glyph* body() const;
};
```

Figure 2.10: MonoGlyph class interface

```
class Background : public MonoGlyph {
public:
  Background(Glyph*, const Color*);
  const Color* color() const;
};
class Border : public MonoGlyph {
public:
  Border(Glyph*, const Color*, Coord thickness);
  const Color* color() const;
  Coord thickness() const;
};
```

Figure 2.11: Background and Border class interfaces

an instance graph are more efficient than having each object draw its own background. A **border** draws a thick solid rectangle around the edge of a glyph. Borders do not request additional space; they simply draw underneath the body. To dedicate space to a border, it is necessary to use a **margin** glyph, which allocates additional space around its body. Figure 2.12 shows the Margin class interface. Margin glyphs do not draw in the space. For convenience, HMargin, VMargin, LMargin, RMargin, TMargin, and BMargin subclasses are provided that create horizontal, vertical, left, right, top, and bottom margins, respectively.

A **shadow** draws a half-filled area under its body. Because it does not fill the area, using a shadow as the root glyph in a window implies that the window should not be double-buffered.

```
class Margin : public MonoGlyph {
public:
  Margin(Glyph* body, Coord all);
  Margin(Glyph*, Coord hmargin, Coord vmargin);
  Margin(Glyph*, Coord left, Coord right, Coord bottom, Coord top);
  Margin(
    Glyph*, Coord left, Coord left_stretch, Coord left_shrink,
    Coord right, Coord right_stretch, Coord right_shrink,
    Coord bottom, Coord bottom_stretch, Coord bottom_shrink,
    Coord top, Coord top_stretch, Coord top_shrink
  );
};
```

Figure 2.12: Margin class interface

```
class Shadow : public MonoGlyph {
public:
  Shadow(Glyph*, Coord x_offset, Coord y_offset, const Color*);
  Coord x_offset() const;
  Coord y_offset() const;
  const Color* color() const;
};
```

Figure 2.13: Shadow class interface

```
class LRMarker : public MonoGlyph {
public:
  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();
};

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

Figure 2.14: LRMarker and XYMarker class interfaces

Figure 2.13 shows the Shadow class interface. An **lr-marker** is a glyph that can makr a region of its body. The marking is done by painting a color under the area or a color over 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 region. 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 2.14 shows the LRMarker and XYMarker class interfaces.

A **transform-setter** is a glyph that contains a transformer that it uses to change the canvas coordinate transformation matrix during allocation, drawing, and picking. The new matrix is the old matrix multiplied by the transform-setter's matrix. The natural size of a transform-setter is the transformed natural size of its body. A transform-setter does not decorate its body in the same sense as the other glyphs described here, but it does modify the appearance of its body. A scale-to-fit object is similar to a transform-setter in that it modifies the transformation matrix of its body. Instead of using a specified transformer, scale-to-fit scales its body by the ratio of its allocation to the body's natural size. For example, if the body's natural size were 10x10 and the

```
class TransformSetter : public MonoGlyph {
public:
  TransformSetter(Glyph*, const Transformer&);
  const Transformer& transformer() const;
};

class ScaleToFit : public MonoGlyph {
public:
  ScaleToFit(Glyph*);
};
```

Figure 2.15: TransformSetter and ScaleToFit class interfaces

```
class Center : public Monoglyph {
public:
  Center(Glyph*, DimensionName, float align = 0.5);
  Center(Glyph*, float xalign = 0.5, float yalign = 0.5);
};

class FixedSpan : public MonoGlyph {
public:
  FixedSpan(Glyph*, DimensionName, Coord span);
};
```

Figure 2.16: Center and FixedSpan class interfaces

allocation was 20x20, then the scale-to-fit glyph would scale by a factor of 2x, making the body twice as big. Figure 2.15 shows the TransformSetter and ScaleToFit class interfaces.

2.5.2 Adjusting geometry

Figure 2.16 shows the MonoGlyph subclasses that modify the geometry of the body. A **center** glyph aligns its body within an allocated area. An alignment in a dimension is specified by a floating point value, indicating the fraction of the total area where the center of the component should be positioned. Centering can be performed on both the X and Y dimensions or on a specific dimension. For convenience, HCenter and VCenter subclasses are provided that center horizontally and vertically, respectively. The term ‘‘center’’ is actually somewhat misleading because only an alignment of 0.5 actually centers the body–a horizontal alignment of 1.0 will right adjust the body. Note that a center only changes the alignment of its body, it does not set the position directly. Therefore, if a center glyph is inside a glyph that does not pay attention to alignments, the result will not be centered as expected. In particular, a top-level center glyph will not have any effect because a window does not use alignments to position its glyph.

A **fixed-span** glyph controls the shape and allocation of its body in one dimension. FixedSpan::request simply replies with its specified size ignoring its body. Similarly, FixedSpan::allocate ignores its allocation and passes the body its specified size. A fixed-span is ideal for use around objects that need a default size, but do not wish to specify one themselves. For example, a vertical scrollbar needs a reasonable horizontal size, but would rather not define one itself. By using fixed-spans, other classes need not provide optional constructors that specify sizes.

2.5.3 Compositions

A composition is a glyph that uses a **compositor** to determine suitable breaks between groups of its components. Figure 2.17 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 builds an LRBox for each sublist, while the TBComposition subclass builds a TBBox.

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 lr-boxes for lines into a tb-composition, which in turn puts the tb-boxes for columns into an lr-composition, which puts the resulting 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(
  new TBBox, new SimpleCompositor, /* no separator */ nil, /* width */ 4*72.0
);
Discretionary* ok = new 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

---

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

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

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

Figure 2.17: Composition classes

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

class ArrayCompositor : public Compositor {
public:
  ArrayCompositor(int N);
};

class SimpleCompositor : public Compositor {
public:
  SimpleCompositor();
};

class TeXCompositor : public Compositor {
public:
    TeXCompositor(int penalty);
};
```

Figure 2.18: Compositor classes

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 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 assesing 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 2.18 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.

2.5.4 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.

```
class Patch : public MonoGlyph {
public:
  Patch(Glyph*);
  virtual Allocation& allocation();
  virtual void reallocate();
  virtual void redraw();
  virtual boolean repick(Hit&, unsigned int depth);
};
```

Figure 2.19: Patch class interface

Figure 2.19 shows the Patch class interface. Patch::allocation returns the patch's most recent allocation. 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.

Chapter 3

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 operations for rendering to a Canvas or to a Printer. 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.

3.1 Graphics Attributes

InterViews provides classes that represent graphics attributes such as colors and fonts. The instances are all sharable, meaning the classes are derived Resource. The objects are also device-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.

3.1.1 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 3.1 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

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

Figure 3.1: Brush class interface

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 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.

3.1.2 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 3.2 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 RGB intensities, an alpha value (default is 1.0), and a drawing operation. A color drawing operation need be specified only in 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::find is a static member function that attempts to find the RGB values for a given name on the given display. Typically, it is used to see if a color name is valid. If Color::find returns true, then the resulting intensities can be used to create the desired color object. 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.

InterViews automatically translates an RGB specification to the appropriate pixel value for a display. This approach hides the device-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 application defaults file.

```
class Color : virtual public Resource {
public:
  enum Op { Copy, Xor };
  typedef float Intensity;
  Color(Intensity r, Intensity g, Intensity b, float alpha = 1.0, Op = Copy);
  static boolean find(Wrold*, const char* name, Intensity& r, Intensity& g, Intensity& b);
  virtual boolean distinguished(World*, Color*);
  virtual void intensities(World*, Intensity& r, Intensity& g, Intensity& b);
};
```

Figure 3.2: Color class interface

```
class Font : virtual public Resource {
public:
  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 boolean find(const char* family, int size, const char*& fullname, float& scale);
  virtual const char* name() const;
  virtual const char* encoding() const;
  virtual Coord size();

  typedef int CharCode;
  virtual Coord left_bearing(CharCode);
  virtual Coord left_bearing(const char*, unsigned int len);
  virtual Coord right_bearing(CharCode);
  virtual Coord right_bearing(const char*, unsigned int len);
  virtual Coord ascent();
  virtual Coord ascent(CharCode);
  virtual Coord ascent(const char*, unsigned int len);
  virtual Coord descent();
  virtual Coord descent(CharCode);
  virtual Coord descent(const char*, unsigned int len);
  virtual Coord width(CharCode);
  virtual Coord width(const char*, int);

  virtual int index(const char*, int , float offset, boolean between);
};

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

Figure 3.3: Font and PSFont class interfaces

3.1.3 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 3.3 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::ascent returns the extent above the font’s baseline; Font::descent returns the extent below the font’s baseline. Both functions can be called with a character code, in which case the extent is for the specified character in the font, with no parameters, in which case the extent is for

the largest character in the font, or with a string of a given length, in which case the extent is the maximum of the extents for the characters in the string.

Font::left_bearing returns the left edge of the bitmap associated with a character; Font::right_bearing returns the right edge. Both functions can be called on a specific character or a string of a given length. 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. 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.

3.1.4 Transformer

A transformer object represents a 3x2 matrix for use in translating 2D coordinates. *This will be extended for 3D in the future*. Figure 3.4 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 result of multiplying the matrix and the given matrix. Two operations are necessary becase matrix multiplication is not communative 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.

```
class Transformer {
public:
  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 3.4: Transformer class interface

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;
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).

3.1.5 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 3.5 shows the Bitmap class interface. There are two constructors for bitmaps. One takes the bitmap data, width, height, and origin (often referred to the ‘‘hot spot’’). 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.

The find 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, find returns true and sets the

```
class Bitmap : virtual public Resource {
public:
  Bitmap(char*, 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);
  virtual void poke(boolean set, unsigned int x, unsigned int y);
  virtual void peek(unsigned int x, unsigned int y);
  virtual Coord width();
  virtual Coord height();
  virtual unsigned int pwidth();
  virtual unsigned int pheight();
  virtual Coord left_bearing();
  virtual Coord right_bearing();
  virtual Coord ascent();
  virtual Coord descent();
  virtual void transform(const Transformer&);
};
```

Figure 3.5: Bitmap class interface

```
class Raster : virtual public Resource {
public:
  Raster(unsigned int pwidth, unsigned int pheight);
  virtual Coord width() const;
  virtual Coord height() const;
  virtual unsigned int pwidth() const;
  virtual void peek(
    unsigned int x, unsgned int y,
    Color::Intensity& r, Color::Intensity& g, Color::Intensity& b, float& alpha
  ) const;
  virtual void poke(
    unsigned int x, unsigned int y,
    Color::Intensity r, Color::Intensity g, Color::Intensity b, float alpha
  );
};
```

Figure 3.6: Raster class interface

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.

3.1.6 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. Figure 3.6 shows the Raster class interface. 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 used.

3.2 Canvas

A canvas is a 2-dimensional area on which to draw. The base implementation draws on a portion of the screen, normallycreated 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 3.7shows the canvas operations. For screen canvases, Canvas::window returns the window containing the canvas; otherwise it returns nil. Canvas::width and Canvas::height return the dimensions of the canvas in coordinates.

```
class Canvas {
public:
  Canvas();
  virtual Window* window() const;
  virtual Coord width() const;
  virtual Coord height() const;

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

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

  virtual void push_clipping();
  virtual void clip();
  virtual void clip_rect(Coord l, Coord b, Coord r, Coord t);
  virtual void pop_clipping();

  virtual void damage(Coord left, Coord bottom, Coord right, Coord top);
  virtual void damage_all();
  virtual boolean damaged(Coord left, Coord bottom, Coord right, Coord top) const;
  virtual boolean any_damage() const;
  virtual void repair();
};
```

Figure 3.7: Canvas class interface

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 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 each 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 may be deferred to improve performance. For example, a series of Canvas::characater 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. The World::flush and World::sync operations can be used to synchronize with the display.

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);
```

3.3 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, shown in Figure 3.8.

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 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 is available to subclasses to force any locally-buffered data to be written.

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

Figure 3.8: Printer class interface

Chapter 4

Display Management

The previous chapters covered glyphs and graphics operations. This chapter discusses how to use glyphs on a display. A **window** is an object that can be mapped onto a **screen** and receive input. Associated with a window is the glyph that 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 draw on the root glyph to refresh the canvas and calls pick on the glyph to determine what to do with input events.

A **display** is the unit of window system control; typically it consists of a single screen, keyboard, and a mouse or other pointing devices. A **session** is the flow of control associated with an application. A session can interact with more than one display and can access all the screens associated with a single display. For simplicity, we assume here that sessions use a single display that has a single screen.

Currently, InterViews provides a single class, World, for both session and display management. In the future, World will be split into two classes: one for the display and one for the session information of an application.

4.1 World

Figure 4.1 shows the session operations on a world A world must be constructed before any other InterViews objects are created. The constructor takes a string that specifies the session's class name, command-line arguments, optional argument description, and optional initial properties. The session's instance name comes from the ‘‘-name’’ command-line option if one is given, or the environment variable RESOURCE_NAME if it is defined, or the value of argv[0] with all leading directories stripped. The session class and instance names are used as the root path for customizable properties.

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 the property, a *style* specifying where the associated value is, and a default *value*. Valid styles are OptionPropertyNext (use the next argument as a property:value pair), OptionValueNext (use the next argument as the value), OptionValueImplicit (use the default value), OptionValueIsArg (use argument as value), and OptionValueAfter (use remainder of argument as value). If a command-line argument is found that matches an option description, it will be interpreted and removed from *argv*. The argument count (*argc*) will be modified to reflect the number of uninterpreted command-line arguments.

In addition to the program-specified options, the options listed in Table 4.1 are matched automatically. Most of the options are identical to standard options provided by all X applications. Options specific to InterViews include -dbuf/-nodbuf, to control double-buffering; -visual, to specify a visual class to use for all windows; -shape/-noshape, to control use of non-rectangular windows; and -malloc to turn on whatever memory-management error checking is provided by the system.

```
class World {
public:
  World(const char*, int& argc, char** argv, OptionDesc* = nil, PropertyData* = nil);
  static World* current();
  const char* name() const;
  const char* classname() const;
  int argc() const;
  char** argv() const;

  virtual void run();
  virtual void quit();

  virtual boolean pending() const;
  virtual void read(Event&);
  virtual boolean read(long sec, long usec, Event&);
  virtual void unread(Event&);
  virtual void poll(Event&) const;
  virtual void grab(Handler*);
  virtual void ungrab(Handler*):
  virtual Handler* grabber() const;
  virtual boolean is_grabbing(Handler*) const;
};
```

Figure 4.1: World session-specific operations

The optional property data is a nil-terminated array of pairs <s1,s2>, where s1 is a string that names a property and s2 is a string that represents the value for the property. User defaults will

-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
-fg	same as -foreground
-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
-name	next argument sets the instance name of the session
-nodbuf	do not double-buffer windows
-noshape	do not try to use non-rectangular windows
-reverse	swap foreground and background colors
-rv	same as -reverse
-shape	allow use of non-rectangular windows
-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
-xrm	next argument is "name:value"; sets named attribute to value

Table 4.1: Predefined command-line options

override any property data, and command-line arguments will override user defaults.

4.1.1 Session control flow

Several operations are provided for getting events from the display. Generally, applications need not call these directly–they can simply call World::run at the end of their main function. World::run reads events from the display, handling them as they arrive, and stopping after a call to World::quit.

World::pending returns whether any events are waiting to be read. World::read gets the next event from the display. World::read returns with any event, not necessarily an event understood directly by the toolkit. For example, under X an application will get a DestroyNotify event when a window is destroyed. The Event class defines no toolkit type for this event (it will be "other") and takes no default action. However, it will be returned by World::read.

World::read can optionally be called with a timeout. It returns true if an event is read before the timeout occurs, or false if no event occurred before the specified time expires.

World::unread puts an event back at the front of the input queue associated with the display. Putting an event back is no necessarily the same as not reading it because the target window could be modified (even destroyed) between the time the event is first read and when it is re-read.

World::grab adds the given handler to the list of grabbing handlers associated with the world. The most recent grabber will always get the first opportunity to receive an event (see Chapter 5 for more details of event handling and grabbing). World::ungrab removes the given handler from the grab list. World::is_grabbing returns whether the given handler is on the grab list.

World::current is a static member function that returns the world instance that most recently received an event. For applications that use a single display, this operation trivially returns the one open display. World::current should be avoided where possible, however, because it may cause performance problems for multi-threaded applications.

4.1.2 Display information

Figure 4.2 shows the World operations for accessing display information. World::propsheet returns the root property sheet for the display. Property sheets are the current mechanism for obtaining user/application defaults and will be replaced in the future by style sheets (which will be described in Chapter 7). World::property_value uses the property sheet to lookup the value associated with the given string. World::property_is_on returns whether the named property has a value set of "on" or "On". World::font returns the default font for the display, World::foreground returns the default foreground color, World::background returns the default background color.

World::shaped_windows returns whether the display allows non-rectangular windows. World::double_buffered returns whether windows on the display are double-buffered by default. World::width and World::height return the width and height in coordinates of the screen.

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

4.2 Window base class

The base class window is abstract; it is not instantiated directly. Two subclasses are provided:

```
class World {
public:
  PropertySheet* propsheet() const;
  const char* property_value(const char*) const;
  boolean property_is_on(const char*) const;
  Font* font() const;
  Color* foreground() const;
  Color* background() const;
  boolean shaped_windows() const;
  boolean double_buffered() const;

  virtual void flush();
  virtual void sync();
  Coord width() const;
  Coord height() const;

  void SetHint(const char*);
  void RingBell(int);
  void SetKeyClick(int n);
  void SetAutoRepeat(boolean);
  void SetScreen(int);
};
```

Figure 4.2: World cisplay-specific operations

ManagedWindow, for defining information for a window manager, and PopupWindow for windows that should be mapped outside of window manage control. Because it circumvents a window manager a popup should only be immediate feedback, such as for popup or pulldown menus.

Figure 4.3 shows the interface for the base class. Window::geometry sets or returns the desired geometry for a window. Geometry specifications are strings of the form ‘‘WxH+X+Y’’ where W is the width, H the height, X the left corner, and 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 opposide 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. The Window::geometry get function returns nil if no geometry has been specified.

Window::double_buffered sets or gets the flag requesting that all drawing be performed into an off-screen area and switched to the screen in one operation instead of drawing directly onto the screen. Using X graphics, this implies a copy of the window contents in the server. Windows are double-buffered by default. It is necessary to turn off double buffering on a window that does not draw its contents completely, such as a half-tone shadow. It also might be undesirable to double-buffer if drawing is done into a limited number of planes. For example, if the underlying system can only double-buffer using hardware planes and there are a limited number of planes, then an image might look better drawn with a single buffer.

4.2.1 Cursors

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

```
class Window {
protected:
  Window(Glyph* = nil);
public:
  virtual void geometry(const char*);
  virtual const char* geometry() const;
  virtual void double_buffered(boolean);
  virtual boolean double_buffered() const;
  virtual void cursor(Cursor*);
  virtual Cursor* cursor() const;

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

  virtual void map();
  virtual void unmap();
  virtual void raise();
  virtual void lower();
  virtual void world(World*);
  virtual World* world() const;

  virtual void receive(Event&);
  virtual Handler* target(Event&);
  virtual void repair();
};
```

Figure 4.3: Window base class interface

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.

Figure 4.4 shows the Cursor class interface. 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. The 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>.

```
class Cursor : virtual public Resource {
public:
  Cursor(short x, short y, int* pattern, int* mask, Color* fg, Color* bg);
  Cursor(Bitmap* pattern, Bitmap* mask, Color* fg, Color* bg);
  Cursor(Font*, int pattern, int mask, Color* fg, Color* bg);
  Cursor(int, Color* fg, Color* bg);
};
```

Figure 4.4: Cursor class interface

4.2.2 Window manipulation

Window::place specify the desired screen coordinates for a window. Window::align specifies a desired alignment. The 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 coordinates specify the lower-left corner of the window.

This programming interface is different from the interpretation of the geometry string, which by default refers to the upper-left corner of the window. The reason for this inconsistency is because the geometry string may be specified by a user, so the interpretation must be consistent with other X applications (users are not interested in which toolkit an application uses). The programming interface, in contrast, is consistent with the other InterViews objects, which have a lower-left orientation.

Window::map requests the window to be mapped onto the screen. If no display has been assigned explicitly with Window::world, then the default one is used. Window::map need *not* be synchronous–one cannot assume the window is usable immediately following calling map. Window::unmap requests that the window be unmapped 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 be under control of the user through a window manager.

4.2.3 Input handling

The window systems delivers input events to a particular window, which in turn tries to find a handler to process each event. Window::receive partial processes an event, handling window maintenance events internally. For example, X expose and configure events are handled directly by Window::receive. Window::target determines if there is a suitable handler for an event. Normally, the window does a pick on its root glyph and iterates through the listeners found to see if any are interested in the event. A handler can also ‘‘grab’’ input, which means the window will send it the event directly instead of doing a pick. Most interface components, including buttons and scrollbars, will grab while a pointer button is pressed. Window::target will return nil if no handler is grabbing and either the event does not have pick coordinates or there are no interested listeners.

Window::repair refreshes the damaged area of the window’s canvas. Before blocking for input, the event reader repairs all damaged windows. Therefore, it is not necessary for an application to call Window::repair directly.

4.3 ManagedWindow

A managed window provides operations to specify information for a window manager to use. ManagedWindow is an abstract base class with four 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 4.5 shows the ManagedWindow class interface. Most of the operations on a managed window set or return information associated with the window. ManagedWindow::name specifies a string the window manager is expected to display with the window, typically in a headline bar. ManagedWindow::icon_name specifies a string to display with the window when it is iconified. ManagedWindow::icon_geometry specifies the size of a window’s icon, using the same format as ManagedWindow::geometry. 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.

```
class ManagedWindow : public Window {
public:
    virtual void name(const char*);
    virtual const char* name() const;
    virtual void icon_name(const char*);
    virtual const char* icon_name() const;
    virtual void icon_geometry(const char*);
    virtual const char* icon_geometry() const;
    virtual void icon_bitmap(Bitmap*);
    virtual Bitmap* icon_bitmap() const;
    virtual void icon_mask(Bitmap*);
    virtual Bitmap* icon_mask() const;

    virtual void icon(Window*);
    virtual Window* icon() const;

    virtual void iconic(boolean);
    virtual boolean iconic();

    virtual void iconify();
    virtual void deiconify();
};
```

Figure 4.5: ManagedWindow class interface

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 conjuction with icon_name, icon_geometry, icon_bitmap, or icon_mask. An icon window cannot have an icon window or icon information.

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 reverse s the operation, unmapping the icon and mapping the original window. It does not make sense to iconify or deiconify an icon window (indeed, IconWindow probably should not be a subclass of ManagedWindow).

Figure 4.6 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 a group leader. TransientWindow is a subclass to TopLevelWindow that can additionally be associated with a primary window with TransientWindow::transient_for.

4.4 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

```
class ApplicationWindow : public ManagedWindow {
public:
  ApplicationWindow(Glyph*);
};

class TopLevelWindow : public ManagedWindow {
public:
  TopLevelWindow(Glyph*);
  virtual void group_leader(ManagedWindow*);
  virtual ManagedWindow* group_leader();
};

class TransientWindow : public TopLevelWindow {
public:
  TransientWindow(Glyph*);
  virtual void transient_for(ManagedWindow*);
  virtual void ManagedWindow* transient_for();
};

class IconWindow : public ManagedWindow {
public:
  IconWindow(Glyph*);
};
```

Figure 4.6: ManagedWindow subclasses

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();
}
```

Chapter 5

Input Handling

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. A **handler** is an object that processes events. A **listener** is a glyph that defines the mapping from events to handlers in a portion of a window. Input events can also be *grabbed* by a handler, meaning the handler will get all events regardless of the input pointer position.

5.1 Events

Input events arrive in a single stream (per display) to an application. The event class interface is shown in Figure 5.1. The event class defines two event-specific types: Event_Type is the kind of event (mouse motion, button down, etc.). InterViews receives all window system events, but only defines event types for device input; otherwise the type is other_event . Event_PointerButton specifies a button on a pointing device, where any is used to indicate any one of the buttons.

Event::world returns the display from which the event originated. An uninitialized event can be bound to a specific display for reading. Event::pending tests if any events are waiting to be read. Event::read gets the next event from the display associated with the event, waiting if necessary. Event::read can also be passed a timeout, specified by two values, one seconds and one microseconds. If no event arrives before the timeout, then Event::read returns false. Event::unread puts the event at the front of the input queue. Event::poll does not read an event, but fills the event with the current state of the pointer and modifier keys.

Event::handler returns a handler for the event or nil if none can be found. Event::handler uses the pointer location in the event to do a pick on the root glyph associated with the target window of the event. A side effect of the pick determines the top-most handler that is interested in the given event.

A handler may also “grab” input events by calling Event::grab. More than one handler may grab at a time, making a list of grabbers ordered from most recent to oldest grab. Event::ungrab removes a handler from the list. Event::grabber returns the first grabber in the list. Event::is_grabbing tests whether the given handler is on the list.

Event::handle process es an event by first checking if any the grabbers in order to see if one will handle the event. If there are no grabbers or none handle the event, then Event::handle calls Event::handler to find a handler and passes the even t to the handler if one is found.

Event::time returns a timestamp in milliseconds for the event. Event::pointer_x and Event::pointer_y return the pointer location for the event. If no location is associated with the event, then 0.0 is returned. Pointer locations are guaranteed to be provided for motion, pointer button, and keyboard events. Event::pointer_button returns the Event_PointerButton for a down or up event; it returns none for all other events. Event::*_is_down returns whether the corresponding 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 generally be avoided. Event::mapkey finds the string binding for a

```
class Event {
public:
  enum Type { undefined, motion, down, up, key, other_event };
  enum PointerButton { none, any, left, middle, right, other_button };
  virtual void world(World*);
  virtual World* world() const;

  virtual boolean pending();
  virtual void read();
  virtual boolean read(long sec, long usec);
  virtual void unread();
  virtual void poll();

  virtual Handler* handler();
  virtual void grab(Handler*);
  virtual void ungrab(Handler*);
  virtual boolean is_grabbing(Handler*);
  virtual void handler();

  virtual Event_Type type(); const;
  virtual unsigned long time() const;
  virtual Coord pointer_x() const;
  virtual Coord pointer_y() const;
  virtual Event_PointerButton pointer_button() const;
  virtual boolean control_is_down() const;
  virtual boolean meta_is_down() const;
  virtual boolean shift_is_down() const;
  virtual boolean capslock_is_down() const;
  virtual boolean left_is_down() const;
  virtual boolean middle_is_down() const;
  virtual boolean right_is_down() const;
  virtual unsigned char keycode() const;
  virtual unsigned int mapkey(char*, unsigned int len) const;
};
```

Figure 5.1: Event class interface

given key. Event::mapkey is passed a buffer to place the result in and a maximum length 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.

5.2 Handlers

A handler is an object that can be passed a raw event to process. Handler::event is a pure virtual function; subclasses must define an implementation.

Handlers typically translate raw events into abstract events. A **pointer-handler** is a handler subclass that translates pointer events (typically from a mouse) into calls to the appropriate virtual functions. Figure 5.2 shows the class interfaces to handler and pointer-handler.

PointerHandler::sense is called when the handler first receives a motion event. PointerHandler::press is called when a button is pressed. A pointer-handler grabs input events

```
class Handler : virtual public Resource {
public:
  virtual void event(Event&) = 0;
};

class PointerHandler: public Handler {
protected:
  PointerHandler();
public:
  virtual void sense(Event&)
  virtual void press(Event&);
  virtual void drag(Event&);
  virtual void release(Event&);
  virtual void commit(Event&);
  virtual void abort(Event&);
};
```

Figure 5.2: Handler classes

while a button is pressed. PointerHandler::drag is called when the pointer moves while a button is pressed. PointerHandler::release is called when the button is released. Normally, PointerHandler::commit is called after the button is released. However, if a second button is pressed during dragging, PointerHandler::abort will be called instead of committ.

5.3 Listeners

Glyphs specify geometric information and are therefore the basic unit for composing objects to form a user interface. The base glyph class directly supports output; however, glyphs are also useful to handle input. The normal approach is to define a MonoGlyph subclass that interprets input, with the body of the monoglyph being the glyph that defines the object's appearance. For example, a button is a monoglyph with an appearance specified by a **telltale**, which is a glyph that can have several appearances (normal, highlight, chosen). The button responds to input by modifying the telltale appearance and performs an action when appropriate (such as when the mouse button is released).

Separating input and output glyphs simplifies the construction of components from existing parts. For example, a button that display a raster image does not require a new button subclass. One can simply pass an image glyph to the existing button class constructor.

Input glyphs are normally derived from the Listener class. A listener represents an area sensitive to input. Listener::pick puts the listener on the hit object's listener list. Listener::target returns the handler associated with the listener if the event is one that the listener is interested in. Interest in events is specified (and returned) by Listener::motion, Listener::key, and Listener::button. Figure 5.3 shows the listener class interface.

It is common for a glyph to be both the listener and the handler for events. The reason for the separation is to allow one handler to be shared among several listeners. Interactive objects such as buttons, however, are both a listener and a handler. In this case, it is common to use multiple inheritance and define Listener::target to return this if the event is of interest.

```
class Listener : public MonoGlyph {
public:
  Listener(Glyph*, Handler*);
  virtual void motion(boolean);
  virtual boolean motion() const;
  virtual void key(boolean);
  virtual boolean key() const;
  virtual void button(boolean, Event_PointerButton = Event::any);
  virtual boolean button(Event_PointerButton = Event::any) const;
  virtual boolean caught(Event&) const;
  virtual Handler* target(Event&);
  virtual boolean pick(Canvas*, const Allocation&, Hit&, int depth);
};
```

Figure 5.3: Listener class interface

5.4 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 5.4 shows the target class interface.

The target behavior is controlled by the sensitivity parameter to the constructor. A value of never_hit means pick should always return false, always_hit means return true. A value of primitive_hit means pick returns true if the area intersects the target’s allocation, character_hit means return true if the area intersects the target’s allocation in the X dimension, and body_hit means return the result of calling pick on the body.

```
class Target : public MonoGlyph {
public:
  enum TargetSensitivity { always_hit, primitive_hit, character_hit, body_hit };
  Target(Glyph* body, TargetSensitivity);
};
```

Figure 5.4: Target class interface

Chapter 6

Style Sheets

A style sheet contains information for user interface objects. Style sheets have three roles: as a customization mechanism with style information specified by the user, as a document structuring mechanism for describing formatting changes, and as a general mechanism for controlling graphics attributes.

This abstraction is not yet implemented and will primarily affect objects with a concrete user interface ,such as menus, buttons, and scrollbars.

Chapter 7
Controls

A control has a concrete look-and-feel that typically allows the user to control an application.

7.1 Buttons

7.2 Menus

7.3 Adjusters

Chapter 8

Documents

A document is a structured object that can contain text, graphics, images, and some day audio and video. Documents are not currently implemented or fully-specified. The intention is to replace string and text editing objects, such as InterViews 2.6 provides with StringEditor and TextEditor, with components that can manipulate documents uniformly.

Chapter 9

Operating System Interface

This chapter will describe the classes that abstract operating system services.

9.1 Memory

9.2 Directory

9.3 Host