A Not-Entirely Gentle Introduction to InterViews

Mark Roseman (roseman@cpsc.ucalgary.ca)
Department of Computer Science, University of Calgary

June 10, 1992

Abstract

InterViews is a C++ class library designed for building graphical interfaces on top of X11. It was implemented by Mark Linton's group at Stanford University. This document and the examples contained herein are largely based on InterViews 3.1α, in particular the InterViews Reference Manual dated May 26, 1992. Because this is an alpha-version, details will change when the "real" release becomes available.

1 What is InterViews?

InterViews is a C++ class library completely encapsulating X11. Unlike many other X toolkits, you don't need to know about the underlying X stuff (Xlib etc.) — honestly! InterViews provides a number of higher level components for constructing X based interfaces.

Key features include:

- high level support of common interface components
- new components easily defined by inheritance
- nice support for combining interface components using $\TeX$ composition model
- look-and-feel independence, e.g. same application can have a Motif or Open Look interface
- support for TIFF images
- some simple container classes
- some lower level operating system stuff (sockets, files, etc.)
- support for graphical editors (Unidraw)

1.1 Evolution of InterViews

Currently, the system is at version 3.1α. There was a major rethinking done between version 2.6 and 3.0, the main implication being that there are two "models" of doing interfaces still present in the system. The 2.6 stuff is being phased out as components are being built to support the 3.0 standards. We'll keep mostly to the 3.0 stuff. There are some widgets however that haven't been redone in 3.0, though there's usually ways to get at them.

2 Hello world

Here's "hello world" done in InterViews:

```
 1  #include <IV-look/kit.h>
 2  #include <InterViews/background.h>
 3  #include <InterViews/session.h>
 4  #include <InterViews/window.h>
 5
 6  int main(int argc, char** argv) {
 7      Session* session = new Session("Hello world", argc, argv);
 8      WidgetKit& kit = *WidgetKit::instance();
 9     session->run_window(
10          new ApplicationWindow(
11              new Background(
12                  kit.label("Hello world"),
13                  kit.background()
14              )
15          )
16      );
17  }
```

Line 7 instantiates a `Session` object, which connects up to the X server and runs the main event dispatching loop. You always do this to start (one per program).

Line 8 sets the variable `kit` to be a pointer to the `WidgetKit` object (of which there is only one — the `instance()` method always returns a pointer to that object). The `WidgetKit` is used to create standard user interface components, such as buttons, menus, etc..

Line 9 tells the `Session` object to run a window, i.e. take a window and handle all events etc. This won't terminate until a call to `Session::quit()` (or a crash), and since there isn't such a call anywhere here, this program will keep running. The one parameter to `Session::run_window` is the window to work with (specified in lines 10–15).

Line 10 creates the window, which is in this case an `ApplicationWindow` (as opposed to say a `Popup` window). The one parameter for the `ApplicationWindow` constructor takes a *glyph* object. A glyph is the basic interface building block in InterViews — specific objects are subclassed from `glyph`.

Lines 11–14 create the glyph that will be displayed in the window. In this case the glyph is a `Background` glyph (which displays a particular color background behind some other glyph). The glyph in front is a `label` (pulled from `kit.label()`), and the background color is the default background color, pulled from `kit.background()`.

3 Compiling and running programs

To actually compile and run the program, stick it in a file "hello.c" in some directory. Set your `PATH` variable to include the directory `/home/grouplab/src/iv3.1/iv/installed/bin/SUN4`. Check that `which ivmkmf` points to the `ivmkmf` in that directory, and not the one in `/usr/local/X11/bin` — use a link if not.

Type the following into a file called "Imakefile":

```
#ifdef InObjectCodeDir

Use_libInterViews()
ComplexProgramTarget(test)

MakeObjectFromSrc(hello)

#else

MakeInObjectCodeDir()

#endif
```

Do a `setenv CPU SUN4`, type `ivmkmf` and then `make Makefiles`. Finally, type `make` to compile your program.

4 More Widgets

The following give some more widgets defined in `WidgetKit` that can be used. The "names" of the widgets are the member functions of the `WidgetKit` used to create them.

4.1 Bevels

`inset_frame(Glyph*)` creates a frame making the contents look recessed

`outset_frame(Glyph*)` creates a frame making the contents look raised

`bright_inset_frame(Glyph*)` is similar to `inset_frame`

4.2 Labels

`label(char*)` creates a text label

4.3 Buttons

`push_button(char*, Action*)` creates a push button having a text string as a label, and will do a callback (via the Action) when pressed — more later on callbacks

`push_button(Glyph*, Action*)` creates a push button like above but specifying some arbitrary glyph (e.g. an image) as the button contents

`default_button(char*, Action*)` creates a default button (e.g. "Okay") `default_button(Glyph*, Action*)` default button with arbitrary contents

`palette_button(char*, Action*)` button that can be toggled on or off

`palette_button(Glyph*, Action*)` ...

`check_box(char*, Action*)` creates a check box

`check_box(Glyph*, Action*)` ...

`radio_button(TelltaleGroup*, char*, Action*)` creates a radio button, where all radio buttons in a group share the same `TelltaleGroup`

4.4 Menus

`Menu* menubar()` returns a menubar... add menu items to this menubar using `append_item(MenuItem*)`

`Menu* pulldown()` returns a pull down menu

`Menu* pullright()` returns a pull right menu

`MenuItem* menubar_item(char*)` returns, e.g. a File menu

`MenuItem* menubar_item(Glyph*)` ...

`MenuItem* menu_item(char*)` returns a standard text menu item; associate a callback with it by using `MenuItem::action(Action*)` or attach a submenu with `MenuItem::menu(Menu*)`

`MenuItem* menu_item(Glyph*)` ...

`MenuItem* check_menu_item(char *)` returns a checkable menu; use `MenuItem::state()->test(is_chosen)` to check if set

`MenuItem* check_menu_item(Glyph*)` ...

`MenuItem* radio_menu_item(TelltaleGroup*, Glyph*)` returns a radio button like menu item

`MenuItem* menu_item_separator()` returns a separator line

4.5 Adjusters

Adjusters are used to adjust some parameter, typically a variable or perhaps the view on a surface. You can attach `Observer` objects to adjusters which will be notified when the adjuster changes.

`hslider(Adjustable*)` returns a horizontal slider

`hscroll_bar(Adjustable*)` returns a horizontal scrollbar

`vslider(Adjustable*)` returns a vertical slider

`vscroll_bar(Adjustable*)` returns a vertical scrollbar

`panner(Adjustable*,Adjustable*)` returns a 2-dimensional panner

5 Laying out widgets

InterViews follows a TEX model for composing interfaces from individual widgets, using boxes and glue, etc. This is made easier using a `LayoutKit` object, which like with the `WidgetKit` object, you can get at via `LayoutKit::instance()`. The following gives some of the common operations from the `LayoutKit`.

`hbox(Glyph*, ...)` creates a horizontal box containing a number of glyphs

`vbox(Glyph*, ...)` creates a vertical box containing a number of glyphs

`overlay(Glyph*, ...)` creates a box where a number of glyphs are overlayed

`deck(Glyph*, ...)` creates a box containing a number of glyphs where only one is visible; which is visible can be retrieved by `Deck::card()` and changed by `Deck::flip_to(int)`

`hglue()` returns a piece of horizontal glue

`vglue()` returns a piece of vertical glue

As an example, this program will put up a window containing the label "good" above the label "bye," separated by a piece of glue.

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

int main(int argc, char** argv) {
    Session* session = new Session("Himom", argc, argv);
    WidgetKit& kit = *WidgetKit::instance();
    const LayoutKit& layout = *LayoutKit::instance();
    return session->run_window(
        new ApplicationWindow(
            new Background(
                layout.vbox(
                    kit.label("good"),
                    layout.vglue(),
                    kit.label("bye")
                ),
                kit.style()->background()
            )
        )
    );
}
```

6 Dialog objects

While the `WidgetKit` object provides some low level interface components, the `DialogKit` aims to provide higher level components. Not much has been implemented yet.

`field_editor(char*, Style*)` creates a field editor, used to edit a string; the first parameter is a sample string, the second can be obtained via `Session::instance()->style()`; the resulting string can be obtained via `FieldEditor::text()`

`file_chooser(char*, Style*)` creates a dialog box for selecting a file, the first parameter being the directory to begin looking in; `FileChooser::selected()` returns the name of the selected file

The InterViews sample application "doc" contains a lot of code which will eventually migrate into the main libraries (see source code in `/home/grouplab/src/iv3.1/iv/src/bin/doc`). The "DialogMgr" contains some higher level components for doing some standard sorts of dialog boxes. The code snippet below shows an example of presenting a dialog box to get a single line of input.

```
mgr = new DialogManager();
if ((s = (char*) mgr->ask( nil, "Prompt string", "Initial string")) != nil) {
  // do something with s
}
```

7 Actions

As mentioned above, InterViews uses `Action` objects to do callbacks for things like buttons, menus, etc. These are done (for now) via preprocessor macros, although this will eventually get changed to use templates. The code fragment below shows how to create a callback to a particular method of an object.

```
class App {
  void mymethod();
};

declareActionCallback(App);
implementActionCallback(App);

void App:mymethod() { /* handle the callback */ }

main() {
  App* a = new App;
  session->run_window(
     new ApplicationWindow(
        kit.inset_frame(
          layout.hbox(
               kit.push_button("Mymethod",
                         new ActionCallback(App)(a, &App:mymethod)
               ),
               layout.hglue(),
               kit.push_button("Quit", &Session::instance()->quit)
          )
       )
    )
  );
}
```

8 Changing the Look and Feel - WidgetKit Revisited

`WidgetKit` provides a convenient way of creating standard interface objects like buttons, menus, etc. Usually this is the most convenient way to create them. If you want to customize such objects, you can instantiate them directly, without using `WidgetKit`. For example, while you could get a button by using the method:

```
WidgetKit::push_button(Glyph*, Action*);
```

you could also instantiate a button more directly, using:

```
Button::Button(Glyph*, Style*, TelltaleState*, Action*);
```

The `Style` and `TelltaleState` control the appearance and behavior (or look and feel) of the button. Styles contain attributes for such things as fonts and colors, while a telltale handles several different appearances for an object (such as pushed or not pushed). The `WidgetKit` operation for creating a button uses a particular style and telltale. Typically, all operations for a `WidgetKit` will have a comparable look and feel.

More interestingly, this suggests that by simply changing the `WidgetKit` to instantiate objects with a different style, you could use the same code in your program to create an entirely different look and feel. In fact, InterViews does this to support several different styles. The `Session` object interprets command line arguments (or X defaults), and can also control exactly what `WidgetKit` is used when you call `WidgetKit::instance()`. The table shows the possible styles.

Option	X Default	Description	Notes
-motif	*gui:Motif	Motif style	
-smotif	*gui:SGIMotif	very nice customization of Motif	current default
-openlook	*gui:OpenLook	Open Look style	not implemented yet
-monochrome	*gui:monochrome	monochrome version of Motif	not in 3.1α

9 More on Windows

So far the examples have shown using a single `ApplicationWindow`. Here's a few more things you can do.

Running multiple windows. To use multiple windows, create them (e.g. `Window* w = new ApplicationWindow(Glyph*)`) and then call `w->map()` rather than `Session::run_window()`. Use `Session::run_window()` for the first window, or `map` the first followed by doing `Session::run()`.

TopLevel windows. Usually, you only have one `ApplicationWindow`, which interprets things like geometry requests on the command line, etc. If you want more than one window, the remainder are usually of class `TopLevelWindow`.

Transient windows. You can create transient windows, which are usually treated differently by the window manager, for things like dialogs, etc. Use objects of type `TransientWindow` for this.

Popup windows. Popup windows are mapped onto the screen directly, without using the window manager. As with the other windows, the constructor for class `PopupWindow` takes a glyph as its only parameter. Popups are placed at a particular location, as shown in the following snippet.

```
PopupWindow* popup = new PopupWindow(g); // create a popup from some glyph
popup->place( x_location, y_location );  // location to place popup
popup->align(0.0, 1.0);   // specify alignment, here top left corner
popup->map();   // place popup on screen
```

10 Designing Your Own Glyphs

The standard widgets provided by `WidgetKit` and other bits of the InterViews library provide a reasonable set of general widgets. However, often you need to design your own for doing things that are application specific. These are designed as a subclass of `Glyph`.

10.1 Glyph protocol

A number of routines will typically need to defined.

`request()` asks a glyph to specify its desired size

`allocate()` informs a glyph an area has been allocated

`draw()` displays the glyph on a `Canvas`

`undraw()` tells a glyph its allocation is no longer valid

`pick()` identifies which glyphs intersect a particular point

A `PolyGlyph` is a glyph containing multiple glyphs, such as boxes. A `MonoGlyph` contains a single other glyph, but provides an easy mechanism to customize one small part of that glyph. Glyphs tend to be lightweight objects which don't maintain information about their allocation, which saves memory and allows them to be shared. However, a `Patch` can be wrapped around any other glyph and will maintain allocation information, providing the ability to for instance redraw an object easily.

10.2 Drawing Operations

Drawing is done on a `Canvas`. The `Canvas` class provides methods to draw lines, curves, strokes, rectangles, fills, characters, bitmaps, images, etc. as well as doing clipping and damage. The class uses objects of type `Brush` (containing a pattern), `Color` (specified as RGB or lookup via name), `Font`, `Transformer` (for doing transformations on `Canvas` operations), `Bitmap`, and `Raster`.

10.3 Input Handling

Input is handled by a subclass of the `InputHandler` glyph. This glyph surrounds any other glyph, and provides methods which can be customized for particular uses. These methods are automatically called for events including mouse movement, press, drag, release, double clicks, and keystrokes. An `ActiveHandler` subclass defines methods for mouse enter and leave events.

11 Other Bits

This section gives a bit of info on some other assorted classes, libraries, etc. that are a part of InterViews. This is just an overview, details are available elsewhere.

11.1 Operating System Support

Support is provided (by way of classes) in a number of areas:

Directory provides support for directories in a file system

File is an abstract class which can be extended to work with particular types of files

Math provides an interface to some standard mathematical functions

Memory provides some standard operations on arrays of bytes

String provides operations on arrays of characters

List provides a parameterized class for dealing with lists of objects, including iterating through the list

Table provides similar support for hash tables

11.2 Unidraw

Unidraw is a framework for designing domain specific graphical editors. These could include drawing editors, circuit diagrams, interface builders, etc. Unidraw was built on top of the 2.6 version of InterViews, and hence doesn't use glyphs, so is a bit harder to use. Unidraw is a very rich framework, allowing powerful editors to be specified using the following abstractions.

Components are graphical representations of elements in a domain. They are divided into **subject** and **view** objects.

Tools support direct manipulation of components, using animation and other visual effects.

Commands define operations on components; they are similar to normal messages, but also carry state (for undo, among other things).

External representations are used to convey domain-specific information outside the editor, for instance a PostScript representation.

11.3 Communications

InterViews manages a Dispatch library, which acts as a front end to normal Unix sockets. A user-defined `IOHandler` can be linked to a socket, so that particular events on the socket will result in the `IOHandler` being notified. This is managed by a `Dispatcher` object.

The Dispatch library also provides mechanisms to send high level application messages to other programs. `RpcWriter` objects are created to send specific messages, while `RpcReader` objects contain methods which are called when specific messages are received. `RpcPeer` objects combine an `RpcWriter` and an `RpcReader` to allow easily connecting to remote sockets.

11.4 TIFF Images

There is a library containing routines to read, display, and manipulate TIFF image format files. This is apparently not InterViews specific, but there are some added hooks for use with InterViews.

11.5 Sample Applications

A number of sample applications are provided with the InterViews distribution.

alert pops up a dialog box with a message in it

dclock is a clock program

doc is a WYSIWYG document editor and the prime example of glyphs code; much of the document handling etc. here will probably end up in the toolkit itself

ibuild is an interface builder (created with Unidraw) which unfortunately generates InterViews 2.6 code (no idea when it will do glyphs)

idraw is an excellent drawing editor built using Unidraw

logo displays the InterViews logo in a window

mailbox monitors your incoming mail

12 Further Work

A number of future modules are planned for InterViews, including:

- add more widgets to `WidgetKit`
- add Open Look style support
- audio, video, and other multimedia objects
- a `FigureKit` for doing structured graphics (replacing Unidraw?)