A Tutorial for InterViews Programmers
Part I: The Basics

John Vlissides

Abstract

InterViews is a library of C++ classes that assist in the design and implementation of graphical interfaces. InterViews makes building applications for workstations easier by providing abstractions that hide the details of the underlying window system. The library contains common toolkit objects such as scroll bars, menus, and buttons. It also has mechanisms for composing these elements into complete interfaces without specifying a lot of detail.

This tutorial introduces the basics of using InterViews. It discusses the primitive and library classes and their use in three demonstration programs. The programs are described in depth to give insight into how InterViews objects are used to build interfaces. This tutorial assumes familiarity with C++ and that the reader has access to the InterViews manual documentation.

1 Introduction

InterViews is a software package designed to simplify building graphical interfaces. To make this possible, InterViews provides the following:

1. *An object-oriented design.* User interface components are easily modeled as a set of objects. From a software engineering standpoint, objects are a good encapsulation mechanism, and inheritance makes extending the system easy. A key aspect of InterViews is that it is implemented in an object-oriented language, C++ [3]. The implementation of InterViews thus directly reflects its object-oriented design.

2. *A set of predefined interface objects.* The InterViews library contains a variety of components such as scroll bars, menus, and buttons. All such components are derived from the **Interactor** base class.

3. *A flexible composition mechanism.* Composition objects are used to compose interactors into complete interfaces. The library contains several subclasses of the **Scene** base composition class, each supporting a different style of composition.

4. *Higher-level functionality.* InterViews also provides a structured graphics library, which simplifies the development of applications that support direct manipulation of hierarchically structured graphics.

An overview of the design and philosophy of InterViews appears elsewhere [1]. The overview defines terminology used throughout the documentation and will give you an idea of what InterViews can do.

This tutorial guides you through three programming exercises in which elementary interfaces are created using InterViews. The purpose of the exercises is to help you learn how to use InterViews by example rather than trial and error. The first exercise is a simple program that draws a text string in a window. In the second exercise the program is rewritten to handle input so that the user can click on the string to terminate the program. The third exercise implements a more sophisticated interface using composition.

A description of the basic InterViews primitives and library classes follows the first exercise. The treatment is brief—it's assumed that you have access to the InterViews Reference Manual [2] for detailed information. Because this tutorial covers only the basics, the structured graphics library is discussed in a subsequent tutorial. In both tutorials we assume that you are familiar with C++.

2 Hello Version 1: Simple Output

The first exercise is a simple program to draw the string "hello world" in a window. The code is shown in Figure 1.

The fundamental task in presenting an interface is to put an interactor on the screen. It is the interactor's responsibility to display itself and

Research supported by the Quantum project through a gift from Digital Equipment Corporation.

```
#include <InterViews/message.h>
#include <InterViews/world.h>

void main (int argc, char** argv) {
    World* world = new World("Hello", argc, argv);
    Message* message = new Message("hello world");

    world->Insert(message);
    world->Run();
}
```

Figure 1: Hello Version 1

interpret input properly. Ideally, an interactor that does what we want already exists somewhere. In reality, a composition of predefined interactors will usually suffice. Sometimes, however, we must define specialized interactors, though even in these cases we can often implement the desired behavior by modifying an existing interactor. Rarely will you be forced to define a new interactor from scratch.

For Hello, we can use the **Message** interactor to display the text. The message interactor will take care of redrawing the string when necessary. By default, message interactors center text in the available space; so the string will stay in the middle of the message even after it is resized.

The first two lines in Figure 1 incorporate the declarations of the objects we will use. Then in the main program we declare two objects: a **world** and a message. The world is the root scene into which all interactors are inserted in order to be visible on the display. Think of the world as the “desktop” in which interactors live. The message object is responsible for drawing the text.

Once we have created the objects, all we need to do is insert the message into the world, and it will appear on the display. Then we tell the world to run for an unspecified time so that the message will stay on the display. While the program runs, the user can see the message and manipulate it just like any other window on the desktop.

Hello is so simple that it doesn't even handle input events. That is, it does not interpret mouse clicks, key presses, or any other user actions. However, the underlying window system generates “events” that tell windows to draw themselves, among other things. The InterViews library handles these events for us in the Run function, which is defined in the Interactor base class (thus we could have called it on the message object just as well) and returns only under certain circumstances. Since Hello does nothing to create such circumstances, there is no way for it to exit. Later we'll see how we can make this happen.

3 Primitives

All user interface components are derived from the Interactor class. Scroll bars, menus, buttons, and the like are subclasses of interactor, as is the base composition class, Scene. Interactors typically interpret input events and produce graphical output. Interactor is one of several *primitive* classes in InterViews. Other primitives are used by interactors to manage input, output, screen space, and to abstract the underlying window system. The primitives completely encapsulate device- and system-dependent code, and InterViews library classes and applications are implemented entirely in terms of the primitives.

3.1 Interactor

The operations on interactors that are relevant to this tutorial are shown in Figure 2. The Redraw operation defines the interactor's appearance. It is called automatically whenever a part of an interactor needs to be redrawn, perhaps because it had been obscured by other applications but is now visible. Highlight tells the interactor to draw itself in its entirety, either in its normal appearance (when its argument is false) or in an alternate form to distinguish itself (when its argument is true). For example, the alternate appearance for

```
virtual void Redraw(Coord left, Coord bottom, Coord right, Coord top);
virtual void Highlight(boolean);
virtual void Handle(Event&);
void Read(Event&);
```

Figure 2: Interface to Interactor base class operations

an item in a menu might be white text on a black background rather than the normal black on white. This alternate appearance could be used to indicate that the item is selected. Redraw is called by InterViews library code in response to window manager operations, though it can be (but normally isn't) called explicitly. Highlight, on the other hand, is usually called explicitly.

The Message class used in the Hello program is an example of a simple interactor. Its Redraw operation merely draws a string in the available screen space. The position of the text is computed based on the available screen space when the message is drawn. A highlighted message has its text and background colors reversed.

Interactors handle input events with the Handle operation. Each event is targeted to a particular interactor. Any interactor can read the next event on the global event queue with the Read operation. The Event object stores state that describes the event. Handle and Read can be used to create "event-driven" input handling, in which only one interactor is responsible for reading events and forwarding them to their target. This is done with the following loop:

```
for (;;) {
    Read(e);
    e.target->Handle(e);
}
```

In fact, this is essentially all that the Run operation does. An event-driven organization is good way to structure a program's input handling code when the number of different input states is small. However, programs that have many context-sensitive input behaviors can be difficult to implement in an event-driven model. For example, when a button is pressed in a pop-up menu, it may be desirable to ignore events for targets other than menu items. It is straightforward for the menu to read events directly, discarding those for other targets. In a purely event-driven model, however, the interactor responsible for reading and forwarding events must "know" that it should not forward events for other interactors while the menu is active. Retaining such knowledge of input state can complicate the implementation unnecessarily.

3.2 Scene

Scenes are used to compose interactors into complete interfaces. Composition is a powerful way to provide flexibility without complexity. A wide variety of interfaces can be implemented simply by combining existing interactors through composition.

Scene is an "abstract" class. That means it simply defines the set of operations, or **protocol**, that subclasses should implement; it does not actually implement them itself. Thus, instances of the Scene base class cannot be created. Instead, the InterViews library provides several scene subclasses, each supporting a different style of composition. The only scene operation used in this tutorial is `void Insert(Interactor*)`, which inserts an interactor into the scene.

3.3 World

World is a scene into which interactors are inserted when they are to appear on the screen. Only "top-level" interactors need be inserted into the world; it is unnecessary, for example, to insert the components of a scene into the world once the scene itself has been inserted. Interactors that are inserted into the world are usually scenes that compose the main windows of the application.

3.4 Shape and Canvas

Interactors contain a **Shape** object that defines their desired size. Shapes contain the "natural" height and width of the interactor and the amounts it can shrink and stretch from its natural shape. Scenes often use the shape information in their component interactors to decide how to allocate space among them. When an interactor is inserted into the world, the actual screen space it is allocated represents a compromise between its natural shape and available space. The actual space allocated to an interactor is defined by its **Canvas**. An interactor's graphical output appears on and is clipped to the canvas.

3.5 Painter

Interactors use **Painter** objects in their Redraw operation to perform graphical output on their canvas. Painters provide operations for drawing filled and open lines, circles, rectangles, and other shapes, as well as text. Painters use **paint** objects such as **Color**, **Pattern**, and **Font** to specify how graphics are drawn.

3.6 Sensor

Different interactor subclasses handle input differently. Some interactors handle only mouse button events, for example, while others also handle mouse motion. Interactors that handle input maintain a **Sensor** object. A sensor defines which input events are relevant for an interactor. For instance, if an interactor's sensor is interested in mouse button events, then these will be queued whenever a mouse button is depressed while the mouse cursor is atop the interactor's canvas. These events will have the interactor as their target.

4 Predefined Interactors

Now we will look at some user interface components that are provided by the InterViews library as predefined interactors. We will use these interactors in the remaining exercises.

4.1 MonoScene and Frame

A **MonoScene** is a scene that contains a single component interactor. A monoscene has no appearance and adopts the shape of its component. An application will often define a subclass of MonoScene to serve as its outermost interactor, into which it inserts the other elements of its user interface. Monoscenes represent the simplest form of composition, namely the enhancement of a single component. **Frame** is a MonoScene subclass that draws a line of user-specified width around its component interactor.

4.2 Box and Glue

A **Box** supports side-by-side composition of interactors. Instances of the **HBox** subclass compose interactors horizontally, while instances of the **VBox** subclass compose them vertically. A box allocates its entire canvas to its components, stretching or shrinking them to fill available space. **Glue** is an interactor that is often inserted between components in a box. Glue is simply blank space that is used to do most or all of the stretching and shrinking so that other components in the box don't have to; otherwise, components might have to be distorted excessively to fit.

4.3 Button

Button is a base class for interactors that can set the value of a piece of state. Buttons affect **ButtonState** objects, which contain a value and a list of buttons that can set the value. Buttons can be "pressed," the effect of which can be redefined in subclasses. By default, pressing a button sets the value of the associated buttonstate to the button's value.

We will use the **PushButton** subclass of button provided by the library. Pushbuttons are rounded rectangles containing some text that highlight themselves when they are pressed.

5 Hello Version 2: Simple Input Handling

The first version of Hello is very simple; the message object essentially does all the work. In this exercise, we'll rewrite Hello to handle input. Specifically, a downclick will draw the message in reverse colors, and the subsequent upclick will terminate the program.

The structure of this version of Hello more closely resembles that of a real application. We will derive a **Hello** class from MonoScene and have it handle input events, and we'll insert a frame containing the message object into the hello object to define its appearance.

Figure 3 shows the declaration of the Hello class. The header files include all the classes that will be used in this program. If you wish to try the program, you should place all the code in a single

```
void main (int argc, char** argv) {
    World* world = new World("Hello", argc, argv);
    Hello* hello = new Hello;

    world->Insert(hello);
    hello->Run();
}
```

Figure 4: Hello Version 2 main program

```
#include <InterViews/frame.h>
#include <InterViews/message.h>
#include <InterViews/painter.h>
#include <InterViews/sensor.h>
#include <InterViews/world.h>

class Hello : public MonoScene {
public:
    Hello();
    virtual void Handle(Event&);
};
```

Figure 3: Hello Version 2 header files and class declaration

```
Hello::Hello () {
    input = new Sensor;
    input->Catch(DownEvent);
    input->Catch(UpEvent);
    Insert(
        new Frame(
            new Message("hello world")
        )
    );
}

void Hello::Handle (Event& e) {
    if (e.eventType == DownEvent) {
        Highlight(true);
    } else if (e.eventType == UpEvent) {
        QuitRunning(e);
    }
}
```

Figure 5: Hello Version 2 class definition

file with the includes and class declaration at the beginning.[1]

The Hello class contains a constructor and a Handle operation that is redefined to manage input. The main program (see Figure 4) creates a world object and a hello object. Then it inserts the hello object into the world and calls Run on the hello object (equivalently, it could call Run on the world). Run will return after the hello object calls QuitRunning, a member function defined in the Interactor base class. Control is then transferred back to the main program, which then exits.

The member function definitions appear in Figure 5. The constructor initializes the sensor first and then inserts the frame containing the message into the hello object itself. The input member variable, defined for all interactors, points to the sensor that specifies events of interest. No events are targetted for an interactor unless it initializes input.[2] The `Catch(DownEvent)` call in Figure 5 makes the sensor interested in down events, while `Catch(UpEvent)` makes it interested in up events.

As described earlier, the Run function's default implementation calls the target interactor's Handle function, giving the target a chance to respond to the event. The target of interest in this example is the hello object, and its Handle does one of two things:

1. If the event is a down event, then it calls Highlight, which makes the hello object appear in reversed colors.

2. If the event is an up event, then it calls QuitRunning, which causes Run to return.

[1] Conventionally, class declarations are placed in their own header files, and class definitions are in corresponding `.c` files. In this tutorial we put the code for each exercise in one file for simplicity.

[2] An interactor can also switch the sensor it uses at any time by calling `Interactor::Listen(Sensor*)`.

The Scene class redefines its Highlight operation to call Highlight on its children. The MonoScene and Hello classes inherit this behavior; therefore calling Highlight on the hello object highlights its message.

6 Hello Version 3: Boxes, Glue, and Buttons

The final version of Hello will compose a message and a button using boxes and glue to make a dialog box. The button will dismiss the dialog box and terminate the program. The composition mechanisms will automatically manage the layout of components when the dialog box is resized. A screen dump of the output produced by the program appears in Figure 6.

As in the previous exercise, we will derive a Hello class from MonoScene. The declaration appears in Figure 7. Notice that Handle is no longer redefined, and the class declares a buttonstate object. Moreover, Run is redefined, and an Interior operation is added to help specify the appearance of the dialog box. The main program is identical to the second version's.

The definition of the constructor and Run appears in Figure 8. The constructor initializes the buttonstate to true and inserts a frame containing the interactor defined by the Interior operation into the hello object. The constructor needn't define a sensor because only the button will handle input events. A Handle function is unnecessary for the same reason. The Hello class overrides the default Run function to allow the buttonstate to define when the program should exit. The button's Handle operation will set the buttonstate's value to false when the button is pressed.

The last and most complicated routine in this version of Hello is the Interior operation. It composes a message and a button using boxes and glue to form the dialog box. The code appears in Figure 9. The routine returns a vbox containing two hboxes and three pieces of vglue. The first hbox contains a piece of glue, a message, and another piece of glue; the second hbox contains a piece of glue, a pushbutton, and another piece of glue. The pushbutton is passed the string to display, the buttonstate to use, and the value it will set the buttonstate to when it is pressed. A piece of vglue separates the two hboxes in the enclosing vbox; two other pieces of vglue pad the area above and below

Figure 6: Hello Version 3 output

```
#include <InterViews/box.h>
#include <InterViews/button.h>
#include <InterViews/event.h>
#include <InterViews/frame.h>
#include <InterViews/glue.h>
#include <InterViews/message.h>
#include <InterViews/world.h>

class Hello : public MonoScene {
public:
    Hello();
    void Run();
private:
    Interactor* Interior();
    ButtonState* status;
};
```

Figure 7: Hello Version 3 header files and class declaration

```
Hello::Hello () {
    status = new ButtonState(true);
    Insert(new Frame(Interior()));
}

void Hello::Run () {
    Event e;
    int alive;

    do {
        Read(e);
        e.target->Handle(e);
        status->GetValue(alive);
    } while (alive);
}
```

Figure 8: Hello Version 3 definition of constructor and Run member function

```
Interactor* Hello::Interior () {
    int space = round(.25*inch);

    return new VBox(
        new VGlue(space, 0),
        new HBox(
            new HGlue(space, 0),
            new Message("hello world"),
            new HGlue,
        ),
        new VGlue(2*space),
        new HBox(
            new HGlue(2*space),
            new PushButton(
                "goodbye world",
                status, false
            ),
            new HGlue(space, 0)
        ),
        new VGlue(space, 0)
    );
}
```

Figure 9: Hello Version 3 Interior operation

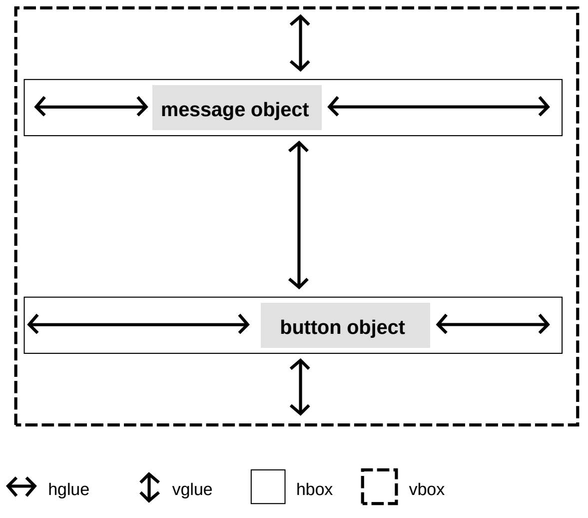

Figure 10: Hello Version 3 schematic

the two hboxes, respectively. The layout of this composition is depicted schematically in Figure 10. Note that the natural size of the glue is defined in terms of inches, a unit represented by a global variable in the InterViews library. Specifying distances in inches instead of pixels makes the code independent of screen resolution.

That completes the implementation of Hello Version 3. An important aspect of the composition is that no explicit position information had to be supplied; we only specified the natural size of the glue separating the components. This *declarative* specification lets us concentrate on the objects and how they are arranged instead of worrying about absolute position and handling resizes manually.

Exercises

1. Change the first version of Hello to use a pushbutton instead of a message, and try it out. Explain why the interface responds to input events now whereas it didn't when it used a message object.

2. In addition to down and up events, sensors can be sensitive to "enter" and "leave" events, which are generated when the user moves the mouse cursor into and out of an interactor's canvas. Modify Hello version 2 to highlight itself when it receives an enter event and return to its original appearance when it receives a leave event.

3. The InterViews library provides a **Dialog** class to simplify building dialog boxes. Rewrite the third version of Hello to use a dialog object. Your version should contain fewer lines of code than the original.

References

[1] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[2] Stanford University. *InterViews Reference Manual, Version 2.6*, 1989.

[3] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1986.