Chapter 5: User Interface Layer

5.1 Understanding the UI Layer and its Execution Framework

Interactive applications are implemented and executed using the user interface software layers (collectively UI layer). The UI Layer refers to a set of softwares that operate above the core operating system (and underneath the application). It encapsulates and exposes system functions for the:

- fluid input and output (I/O),
- facilitation of development of I/O functionalities (in the form of an application programming interface/library (API) or toolkit) and
- run-time management of graphical applications and UI elements often manifested as windows or GUI elements (in the form of separate application often called the window manager).

Since most interfaces are graphical, the UI layer uses a 2D (or 3D) graphical system based on which graphical user interface elements are implemented (lower part of Figure 5.1). Thus, to summarize, the UI layer is largely composed of (1) an API for creating and managing the user interface elements (e.g. windows, buttons, menus), and (2) a window manager to allow users to operate and manage the applications through its own user interfaces.

[Figure 5.1]: The user interface software layer for window based multitasking UI.

Figure 5.1 illustrates the UI layer as part of the system software in many computing platforms. The user interacts with the window/GUI based **applications** using various input and output **devices**. At the same time, aside from the general applications, the user interacts with the computer and manages multiple application windows/tasks (e.g. resizing, focusing, cutting and pasting, etc.) using the (background running) **Window Manager**. The window manager is regarded as both an application and API. User applications are developed using the API's that represent abstracted I/O related functionalities of the UI layer such as those for window managing (resizing, iconifying, dragging, copy and paste, etc.), GUI elements and widgets (windows, menus, buttons, etc.), and basic windowing (creating/destroying window, deactivating window, etc.). These API's are abstracted from the even lower level API's for 2D/3D graphics and the operating system. Note that the above architecture can be equally applied to non-window based systems such as those for layer-based[1] (like in the mobile phones). Through such an architecture and abstraction, it becomes much easier to develop and implement interactive applications and their user interfaces.

5.2 Input and Output at the Low Level

[1] "Layer" here refers to a screen-full interaction space, often used in small display computing platforms. Multiple layers (representing multiple interactive processes), while visible one at a time, are possible and switchable, e.g. "swiping" gestures on smart phones.

At the lowest level, inputs and outputs are handled by the **interrupt** mechanism by the system software (operating system). An interrupt is a "signal" to the processor indicating an (usually I/O) event has occurred and requires to be handled. An interrupt signal is interpreted so that the address of its handler procedure can be looked up and be executed while suspending the on-going process for a moment. After the handler procedure is finished, the suspended process resumes again. An arrival of an interrupt is checked at a very fast rate, as part of the processor execution cycle. This practically makes the processor "always" listening to the incoming events and being ready to serve them. The interrupt mechanism is often contrasted to **polling** (also known as the "busy waiting"). In polling, the processor (rather than the I/O device) initiates input or output. In order to carry out I/O tasks, the processor enters a loop, continually checking the I/O device status whether it is ready, and incrementally accomplishes the I/O task. This form of an I/O is deficient in supporting asynchronous user-driven (anytime) I/O and wastes CPU time by blocking other non-I/O processes to move on. At a higher level, the I/O operation is often described in terms of "events" and "event handlers," which is in fact an abstraction of the lower level interrupt mechanism. This is generally called the "event-driven" architecture in which programs are developed in terms of events (such as the mouse clicks, gestures, keyboard input, etc.) and their corresponding handlers. Such information can be captured in a form of a table and used for efficient execution. Figure 5.2 shows the rather complicated interrupt mechanism abstracted into form of a simple event-handler table.

[Figure 5.2] The complex interrupt mechanism abstracted as event-to-handler table.

5.3 Processing the Input and Generating Output

5.3.1 Events, UI Objects and Event Handlers

The diagram in Figure 5.1 is the nominal software architecture for realizing interactive computing systems. How does it work exactly? In other words, how are the user/device input processed and how does the application (with the help of the UI layer) generate output? Central to its overall interworking are **events, UI objects,** and **event handlers.**

The most basic UI object in today's visually oriented UI system would be the **window** (or layer[1]). A window is a rectangular portion of the screen associated with a given application whch is used as a space and channel for interacting with the application. Other UI objects include those such as the buttons, menus, icons, forms, dialog boxes, text boxes and so forth. These are often referred to as "**GUI objects**" or "**widgets**". Most typically, GUI based interactive applications would have a "top window" with all other UI objects or widgets which are logically and/or spatially subordinate to it (Figure 5.3). With the current operating systems mostly supporting concurrency, separate windows/widgets for concurrent applications can co-exist, overlap with one another, and switched for the current "focus." That is, when there are multiple windows (and one mouse/keyboard), the user carries out an action to designate the active or current window ***in***

focus to which the input events will be channeled. Two major methods for focusing are (1) click-to-type and (2) move-to-type. In the former, the user has to explicitly click on the window before making input into it (regardless of the mouse position, the last object that was clicked on will be the one in focus), and in the latter, the window for which the mouse cursor is over becomes in focus. The move-to-type method is generally regarded less convenient because of the unintended focus change due to accidental mouse movements.

While not all UI layers are modeled and implemented in an object oriented fashion, many recent ones are. Thus we can think of generic or abstract object classes for a window and other UI objects and widgets and organized hierarchically (see Figure X). Moreover, we can designate the background screen space as the default "root" system window (which becomes automatically activated upon system start) onto which children application windows and GUI elements (e.g. icons, menus) are put on. The background also naturally becomes the top window for the window manager process.

[Figure 5.3] The "root" (background) window activated by default upon system start and also serving as the top window for the window manager process.

Whether it is the root (background) window, application (top) window or GUI widget, as an interaction channel or object, it will receive input from a user through input devices such as the keyboard, mouse, etc. The physical input from the user/devices is converted into an **"event"** (e.g. by the device drivers and operating system) which is simply data containing information about the user intent or action. Aside from the event value itself (e.g. which key was pressed), an event usually contains additional information such as its type, a time stamp, the window to which it was directed, coordinates (e.g. in case of a mouse or pen event). These events are put into a queue by the operating system (or the windowing system) and "dispatched" (or dequeued) e.g. according to the current focus (to be directed to the target program or process) and to invoke its corresponding handler.

Note that an event does not necessarily correspond exactly just to an individual physical input. The stream of raw inputs may be filtered and processed to form/find meaningful input "sequence" from the current "context." For instance, a sequence of raw inputs may form a meaningful event such as "double-click," commands with modifiers (e.g. alt-ctrl-del), "mouse-enter/exit" (e.g. detection of mouse cursor leaving a particular window).

Figure 5.4 shows the "two tier" event queuing system in a more detail. There is the system-level event queuing system that dispatches the events at the top application level. Each application or process also typically manages its own event queue and dispatching them to its own UI objects. The proper event is "caught" by the UI object as it traverses down the application's hierarchical UI

structure e.g. from top to bottom. Figure 5.4 and 5.5 illustrate this process. Then the **event handler** (also sometimes called the callback function) associated with the UI object is activated in response to the event that is "caught."

[Figure 5.4] The event queuing at the top application level.

Figure 5.5: The event being dispatched to the right UI object handler for a given application (organized as a set of UI objects and associated event handlers in a hierarchical manner) from the application event queue.

The events do not have to necessarily be generated externally by the interaction devices, but sometimes internally for special purposes (these are sometimes called the **pseudo-events**). For instance, when a window is resized, in addition to the resizing itself, the internal content of the window must be redrawn and the same goes for the other windows occluded or newly revealed by it (after resizing). Special pseudo-events are enqueued and conveyed to the respective applications/windows. In the case of resizing/hiding/activating and redrawing of windows, it is the individual application's responsibility, rather than the window manager's, to update its display contents because only the respective applications have the knowledge as how to update its

content. Thus a special "redraw" pseudo-event is sent to the application with the information of which region is to be updated (see Figure 5.6). The window content might need to be redrawn not because of the window management commands such as resizing and window activation but due to the needs of the application itself. The application itself can generate special pseudo-events for redrawing parts of its own window. More generally, UI objects can generate pseudo-events for creating chain effects e.g. when a scroll bar is moved, both the window content and the scroll bar position have to be updated [1].

Figure 5.6: Exposing a window and redrawing it by enqueing a special "redraw" event with the update area information. The event is matched to a proper redraw handler for the given application.

5.3.2 Event driven Program Structure

Based on what we have discussed so far, the event driven program structure generally takes the form of the following structure. The first initialization part of the program creates the necessary UI objects for the application and declares the event handler functions and procedures for the created UI objects. Then, the program enters a loop that automatically keeps dequeing an event from the application event queue, and invoking the corresponding handler (i.e. dispatching the

event). Often the development environment hides this part of the program so that the developer does not have to worry about such system level implementations. However, depending on the development toolkits (see Chapter 6), the user may have to explicitly program this part as well.

,

Figure 5.7: The event driven program structure: UI object creation and event handler set up followed by the event processing loop either provided by the underlying programming environment system / operating system (above) or by explicit user programming (below).

5.3.3 External Output

Interactive behavior that is purely computational will simply be carried out by executing the event handler procedure. However, response to an event is often manifested by explicit visual, aural, haptic/tactile output as well. In many cases, the event handlers only compute for the response behavior, and changes in data needed or new output in a chosen modality (e.g. visual, aural, haptic, tactile, etc.). A separate step for refreshing the display based on the changed screen data is called as last part of the event processing loop. Analogous processes will be called for sending commands to output devices of other modalities as well (last line in Figure 5.7). Sometimes, with multimodal output, the outputs in different modalities need to be synchronized (e.g. output visual and aural feedback at the same or nearly the same time). However, not many interactive

programming frameworks or toolkits offer provisions for such a situation.

While internal computation takes relatively little time (well, in most cases), processing and sending the new/changed data to the display devices can take significant time. For instance, a heavy use of 3D graphic objects can be computationally expensive (e.g. on a mobile device without a graphics subsystem) and this can become a bottleneck in the event processing loop, resulting in an abated interactivity.

5.4 Summary

In this chapter, we looked at the inner-workings of the general underlying software structure (UI Layer or UI Execution Framework) on which interactive programs operate. Most UI frameworks operate in similar ways according to an event driven structure. The hardware input devices generate events that are conveyed to the software interfaces (i.e. UI objects), and they are processed to produce output by the event handling codes. The UI layer sitting above the operating system provides the computational framework for such an event-driven processing model and makes useful abstractions of the lower OS details for easier and intuitive interactive software and interface development. The next chapters introduce toolkits and development frameworks that make the interface development even more convenient and faster.

Reference

[1] Olsen, Dan, *Developing user interfaces – interactive technologies.* Morgan Kaufman, 1998.