Tema III
Gestión de Eventos (R-1.1)

Programación en Entornos Interactivos.

14 de marzo de 2011

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Alicante

Resumen

Preliminares. ¿Qué es un evento?. La estructura XEvent general. Máscaras y tipos de eventos. Gestión de eventos con Xt. Gestión de la cola de eventos. Gestión de "Timeouts". Uso de "Workprocs". Otras fuentes de entrada. Ejemplo.

Preliminares

- El servidor **X** se comunica con los clientes por medio de eventos.
- Un **toolkit** consigue que los *Widgets* gestionen ellos solos los eventos más comunes, aún así, hay ocasiones en las que un programador debe trabajar con los eventos por sí mismo.
- Si entendemos bien los eventos tal y como son generados por el servidor **X**, entonces podremos utilizar el binomio **XToolkit+Widget Set** de un modo más eficiente.

¿Qué es un evento? (I)

- Un evento es una notificación enviada por el servidor **X** a un cliente debido a algún cambio. Un evento siempre está relacionado con alguna ventana - *source window* del evento -.
- El servidor **X** notifica de la ocurrencia de un evento a todos los clientes interesados en él.
- Las aplicaciones sólo reciben los eventos que quieren.

¿Qué es un evento? (II)

- Si ningún cliente ha solicitado el evento producido para una ventana *source window*, el servidor **X** lo propaga por la jerarquía de ventanas, hasta encontrar alguna ventana para la que algún cliente lo haya solicitado, o encuentra una ventana que prohibe la propagación del evento. La ventana a la que finalmente se notifica el evento, se denomina *event window*.
- Si en el proceso anterior se llega a la *ventana raíz*, el evento se descarta.
- Los eventos se depositan en una cola, de la cual son retirados por los clientes con la función de *Xt Intrinsics*: **XtNextEvent**.
- A este tipo de eventos los llamaremos Eventos de Bajo Nivel.

¿Qué es un evento? (III)

- Esta función rellena una estructura de tipo **XEvent**:
  - Tipo del evento: **type**.
  - El **display** donde sucedió el evento.
  - La ventana asociada al evento: **window**.
  - El número de serie de la última petición atendida por el servidor: **serial**.
  - Un indicador de si el evento fue generado por el servidor o por un cliente: **send_event**.
- La estructura **XAnyEvent** permite el acceso a los campos comunes de los distintos tipos de eventos: `event.xany.window`.

La estructura XEvent general (I)

```
typedef union _XEvent {
    int type;                /* must not be changed */
    XAnyEvent                xany;
    XKeyEvent                xkey;
    XButtonEvent             xbutton;
    XMotionEvent             xmotion;
    XCrossingEvent           xcrossing;
    XFocusChangeEvent        xfocus;
    XExposeEvent             xexpose;
    XGraphicsExposeEvent     xgraphicsexpose;
    XNoExposeEvent           xnoexpose;
    XVisibilityEvent         xvisibility;
    XCreateWindowEvent       xcreatewindow;
    XDestroyWindowEvent      xdestroywindow;
    XUnmapEvent              xunmap;
    XMapEvent                xmap;
    XMapRequestEvent         xmaprequest;
    XReparentEvent           xreparent;
    XConfigureEvent          xconfigure;
    XGravityEvent            xgravity;
    XResizeRequestEvent      xresizerequest;
```

La estructura XEvent general (II)

```
1  ...
   XConfigureRequestEvent xconfigurerequest;
3  XCirculateEvent        xcirculate;
   XCirculateRequestEvent xcirculaterequest;
5  XPropertyEvent         xproperty;
   XSelectionClearEvent   xselectionclear;
7  XSelectionRequestEvent xselectionrequest;
   XSelectionEvent        xselection;
9  XColormapEvent         xcolormap;
   XClientMessageEvent    xclient;
11 XMappingEvent          xmapping;
   XErrorEvent            xerror;
13 XKeymapEvent           xkeymap;
   long                   pad[24];
15}XEvent;
```

Máscaras y tipos de eventos (I)

- Una máscara de eventos indica al servidor **X** los tipos de eventos de los que debe informar a un cliente cuando se produzcan.
- Esta máscara se emplea en la función de Xlib: **XSelectInput()**. Con esta función, un cliente indica al servidor **X** los eventos de los que quiere ser informado para cada ventana:
  `XSelectInput(d,window,ButtonPressMask|ButtonReleaseMask)`
- Una misma máscara puede dar lugar a que lleguen distintos tipos de eventos, no uno solo.
- Algunas constantes que representan máscaras: `KeyPressMask`, `ButtonPressMask`, `EnterWindowMask`, etc...

Máscaras y tipos de eventos (II)

Los tipos de eventos se pueden agrupar en diversas categorías:

Keyboard events	**Pointer events**
Crossing events	**Focus events**
Exposure events	**Structure control events**
State notification events	**Colormap notification events**
Communication events	

Gestión de eventos con Xt

- **Xt** '*oculta*' muchos aspectos de la gestión de eventos al programador.
- **Xt** permite que cada *widget* gestione los eventos más comunes de manera automática...
- Esto implica que muchas aplicaciones no necesiten tratar directamente con los eventos de bajo nivel, aunque si en una aplicación es necesario se puede hacer.
- Un cliente puede recibir notificación de un tipo de evento particular *instalando* un manejador o '*handler*' mediante la llamada a **XtAddEventHandler()**:

```
XtAddEventHandler(wgt,PointerMotionMask,FALSE,
                  handler_function,data)
```

- Es posible definir varios *manejadores* para un mismo evento, todos ellos son invocados cuando es necesario.

Gestión de la cola de eventos

- La forma habitual de *consumir* y *redirigir* los eventos a los *widgets* apropiados en **Xt** es llamando a las funciones: **XtMainLoop()** o **XtAppMainLoop()**.
- Pero en Xlib tenemos funciones para tratar directamente con la cola de eventos, funciones que están '*duplicadas*' en la capa de **Xt**: **XtPending()** y **XtPeekEvent()**.
- **XtPending()** es útil cuando la aplicación debe realizar otras tareas si no hay eventos en la cola -*ejemplo poco eficiente*-:

```
    if (XtPending()){
2     XEvent event;
      XtNextEvent(&event);
4     XtDispatchEvent(&event);
    }
6   else { /* Hacer algo  si no hay eventos */ }
```

Gestión de Timeouts (I)

- Se pueden considerar como una extensión de la noción de *evento*, de manera que una aplicación puede ejecutar otras tareas ayudándose del mecanismo de gestión de eventos.
- Un `TimeOut` es una función que será llamada automáticamente cuando haya transcurrido un intervalo de tiempo.
- Esto se consigue con:
  `XtAddTimeOut(interval, proc, data)`
  la cual devuelve un dato de tipo `XtIntervalId`.
- El **intervalo** se da en milisegundos.
- El prototipo de la función **proc** es:
  `void proc(caddr_t data, XtIntervalId *id)`.

Gestión de Timeouts (II)

- Cuando se produce el `TimeOut`, **Xt** invoca a la función -*callback*- y elimina esa función, de manera, que ya no se llamará más.
- Un cliente puede evitar que una función asociada a un `TimeOut` se ejecute cuando se acabe el tiempo llamando a: **XtRemoveTimeOut(id)**.
- Este '*id*' debe ser el que devolvió la función `XtAddTimeOut`.
- Es posible instalar `TimeOuts` cíclicos.

Uso de Workprocs

- Un `Workproc` es un *callback* invocado por **Xt** cuando no hay eventos pendientes.
- Un `Workproc` sólo tiene un único parámetro, que representa cualquier dato que pueda recibir.
- Un `Workproc` debería devolver **true**, si queremos que se elimine el *callback* después de ser llamado, o **false** si queremos que siga invocándose de forma cíclica.
- Para registrar un `Workproc` usaremos: **XtAddWorkProc(proc, data)**. Esta función devuelve un `ID` que identifica al `Workproc` y que podemos usar luego en:
- **XtRemoveWorkProc(id)**, para eliminarlo cuando queramos.

Otras fuentes de entrada

- Una aplicación puede recibir 'datos' de más sitios aparte de la cola de eventos, por ejemplo de un fichero.
- Esto se consigue con: `XtInputId XtAddInput(fd,cond,proc,data)`, donde:

  fd : Es un número válido de fichero en Unix.
  cond : Es una condición que indica cuándo llamar al callback...
  proc : Es el callback a invocar
  data : Son datos adicionales que podemos pasar al callback.
- El prototipo del callback que representa **proc** es: **void io_cb(caddr_t, int *fd, XtInputId *id)**.
- Al igual que en ocasiones anteriores, podemos eliminar uno de estos callbacks con la función: **XtRemoveInput(XtInputId id)**.

Ejemplo

```
/* Ejemplo sencillo con XtIntrinsics */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <sys/types.h>

void main(int argc, char *argv[]) {
   Widget topLevel;
   Arg wargs[10];
   int n;

   topLevel = XtInitialize(argv[0],"Memo 1.0",
                           NULL,0,&argc,argv);
   /* Mas codigo...*/
   XtRealizeWidget(topLevel);

   XtMainLoop();
}
```