UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Gestión de eventos en el desarrollo de aplicaciones con MetaTalk

Apellidos, nombre	Agustí i Melchor, Manuel (magusti@disca.upv.es)
Departamento	Departamento de Informática de Sistemas y Computadores
Centro	Escola Tècnica Superior d'Enginyeria Informàtica Universitat Politècnica de València

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

1 Resumen de las ideas clave

Este artículo es una breve revisión de las operaciones relacionadas con la gestión de eventos y paso de mensajes entre objetos en el lenguaje *MetaTalk.*

1.1 Objetivos

El objetivo global es exponer la similitud que el desarrollador acostumbrado a los paradigmas de programación dirigida por eventos puede observar en el uso del lenguaje *MetaTalk*. Concretando, nos vamos a centrar en:

- La secuencia de objetos que atraviesa un mensaje.
- Las operaciones propias de gestión de mensajes,
- La reescritura de operaciones explícitas de consulta de eventos (propia de la programación secuencial y también llamada *poolling*) que consumen muchos recursos de la máquina, por las operaciones de respuesta a eventos (o programación dirigida por eventos).

En este documento se exponen pequeños listados de código relacionados con las diferentes órdenes que se exponen para dar pie al usuario a que investigue su uso, ejecutando la aplicación, probando el código y consultando la ayuda incluida dentro de la propia herramienta. No espere largas y detalladas explicaciones del código: sea activo, pruébelo.

2 Introducción

Acostumbrados a utilizar el evento **mouseUp** para realizar una acción cuando se “pincha” con el ratón sobre un objeto, ¿se imagina haciendo un bucle esperando a que pulsen un determinado botón? No sólo la cantidad de código necesaria sería un problema, sino que también casi nada más se ejecutaría en *MetaCard* hasta que acabase el código asociado a ese comportamiento. Estas situaciones son tratadas habitual y eficazmente con una implementación basada en código asociado a eventos.

En *MetaTalk* (*MetaCard*), como lenguaje que es para realizar un estilo de programación dirigida por eventos, las cosas no suceden hasta que no llega un evento que las desencadene. Bien sea este evento propio del sistema, bien sea uno que nos inventemos y que lo “generaremos” con la orden **send**. Un objeto que recibe un evento no reaccionará ante él, si no dispone de un manejador para ese evento.

Los manejadores de eventos pueden recibir información complementaria a través de parámetros. Si ha de “generar” un evento con más de un parámetro, habrá de pasar estos parámetros al manejador separados con comas. Véase un ejemplo de uso en el listado 1.

```
on concatenaCadenas a b
 put a & b
end concatenaCadenas

on mouseUp
 send concatenaCadenas && "cadena1" & "," & "cadena2" to me
end mouseUp
```

Listado 1. Ejemplo de uso de manejadores propios y paso de parámetros.

Pero, ¿qué pasa si, como hemos apuntado, un objeto recibe un evento y no hay ningún manejador dispuesto para él? Vamos a hablar de la propagación de esos eventos que no son manejados: existe una jerarquía, un camino o unas reglas. que regula esta "circulación".

3 Envío de mensajes y cancelación

Es posible utilizar un bucle en el modo usual de la programación secuencial para mantener valores actualizados de una variable o la posición de un objeto. Pero en ese caso, el motor de *MetaCard* estará continuamente ocupado realizando esa tarea. Es posible y recomendable, en determinados casos, reescribir un bucle como el código necesario para tratar esa situación y la generación de un mensaje que, un tiempo después o cuando se produzca el evento físico correspondiente, desencadene de nuevo tal secuencia de instrucciones.

Figura 1: Aspecto de la pila en la que se desarrolla la problemática de sustituir los bucles por envío de eventos.

Veamos el planteamiento de un situación para ir introduciendo los cambios, es un poco forzada (existen otra soluciones, claro) pero es para ilustrar el tema. Se quiere mover simultáneamente un par de objetos gráficos por la pantalla: el "objeto1", de color rosado, y el "objeto2", de color verdoso, que se muestran en la fig. 1. La aproximación más *naif* es la de mover los objetos de su posición de origen a la final, véase listado 2, que lo hace con la orden **move** que utiliza el botón "inciAleatori" de la pila que muestra la fig. 1.

```
# Botón iniciAleatori
on mouseUp
  move graphic "objecte1" to\
      random(the width of this card),random(the height of this card)
  move graphic "objecte2" to\
      random(the width of this card),random(the height of this card)
end mouseUp
```

Listado 2: Código inicial de mover dos objetos.

```
# Código del botón "Inici"
on mouseUp
  move graphic "objecte1" to 86,78
  move graphic "objecte2" to 586,78
end mouseUp

# Código del botón "bucleRepeat"
on mouseUp
  repeat with i = 1 to 25
    move graphic "objecte1" to\
        (the first item of the loc of graphic "objecte1" + 20), \
         the second item of the loc of graphic "objecte1"
    move graphic "objecte2" to\
        (the first item of the loc of graphic "objecte2" - 20), \
         the second item of the loc of graphic "objecte2"
  end repeat
end mouseUp
```

Listado 3: Segunda versión del código para mover dos objetos.

Pero, claro: hasta que no acaba una orden, no puede empezar la siguiente. ¿Solución? Hacer pequeños movimientos y alternar entre los objetos, de manera que parezca que se mueven al tiempo. Establezcamos una posición fija de referencia para comparar y hagamos un bucle que mueva esos dos objetos a la hora. El botón "Inici" envía los objetos a una posición prefijada y el botón "bucleRepeat" implementa la solución expuesta:

Si ha probado el código ... ¿Lo está haciendo, verdad? Bien. Como decía, en la experimentación del código del listado 3, también habrá podido apreciar los "saltitos" que dan los objetos. Aunque hemos dividido la trayectoria en 25 "pasos", todavía se aprecia la alternancia en el movimiento: no hay sensación de simultaneidad.

El caso se puede agravar si el número de objetos fuese mayor. ¿Por qué? Imagine que el número de objetos fuese mayor: habría más tiempo transcurrido entre cada "paso" de un objeto y, por tanto, sería más evidente. Además, piense que nada más en la aplicación se podrá estar ejecutando, puesto que el código de un manejador es un bloque que se ejecuta de un tirón.

Vamos a cambiar de estrategia: vamos a dejar que sean los propios objetos quienes se muevan y de una forma más fluida. El listado 4 muestra el código para el botón “bucleRepeat2” y los círculos (los objetos gráficos “objecte 1” y “objecte 2”) de la fig. 1. Para el “objecte 2”se utiliza el mismo código que para “objecte 1”, pero con el signo cambiado.

```
# Código del botón “bucleRepeat2”
on mouseUp
  send "menejat" to graphic "objecte1"
  send "menejat" to graphic "objecte2"
end mouseUp

# Código del gráfico “objecte1”
local nMoiments
on menejat
  if nMoiments = 25
  then put 0 into nMoiments
  else
    move me rel -20,0
    add 1 to nMoiments
    send "menejat" to me in 1 millisecond
  end if
end menejat
```

Listado 4: Tercera versión del código para mover dos objetos.

3.1 Menos bucles y más eventos

Si está acostumbrado a trabajar con lenguajes imperativos tiene tendencia a programar de una manera, digamos, explícita y secuencial. Aquí revisamos el enfoque de dos aplicaciones, exponiendo una posible solución realizada utilizando el paralelismo que sugiere la metodología de la programación orientada a eventos. El motivo es que es el modo más óptimo para determinadas tareas que se realizan habitualmente en este lenguaje que nos ocupa: *MetaTalk*.

En concreto, vamos a analizar dos ejemplos que se basan en la sucesión de eventos del sistema y del interfaz con el usuario, que se reciben para decidir las acciones a realizar. De modo que sea posible que la llegada de los segundos

pueda interrumpir a los primeros y suceda de una manera "natural". Empezamos con un caso donde no parece que haya necesidad de discutir que las cosas suceden por eventos. Después pasaremos a uno que reescribiremos como una sucesión eventos que pueden llegar en cualquier momento.

3.1.1 Caso de estudio: un reloj

Un caso, eminentemente sencillo, es el de disponer un reloj (puede ser bastante sencillo como el de la fig. 2) que mantenga la hora actualizada. Para implementarlo vamos a programar la sucesión de eventos y el tratamiento correspondiente a los mismos. Lo que queremos ver aquí es que es muy fácil actuar al mismo que aquel, para por ejemplo pedir que se detenga en cualquier momento.

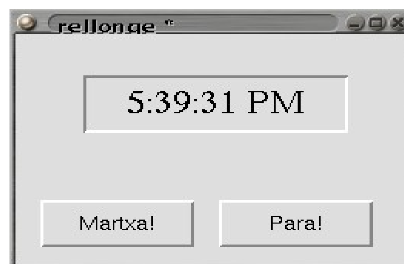


Figura 2: Aspecto de la pila rellonge.mc para probar el envío y la cancelación de eventos.

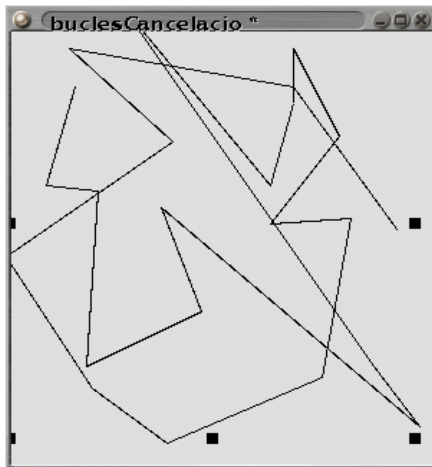
```
# Código del botón "Martxza!"
on mouseUp
  send "actualitza" to me in 1 second
end mouseUp

on actualitza
  put the long time into fld "display"
  send "actualitza" to me in 1 second
  put the result into timerID
end actualitza

# Código del botón "Para!"
global timerID
on mouseUp
  cancel timerID
end mouseUp
```

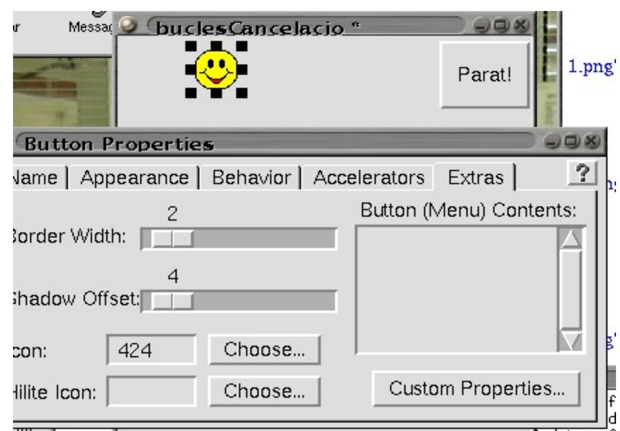
Listado 5: Un posible código para la implementación de un reloj usando eventos.

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Para construir esta aplicación se han dispuesto tres objetos y se ha decorado un poco el tipo de letra que muestra el campo de texto, cosa que dejo a su experimentación. El botón de la izquierda es el encargado de poner en marcha el reloj y mantenerlo actualizado. El listado 5 muestra la implementación sugerida para él.

a) b)





c) d)

Figura 3: Ejemplo de bucle programado con eventos: la primera fila constituye la disposición de los objetos (el punto de partida) y la segunda los dos posibles resultados que queremos obtener.

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

3.1.2 Caso de estudio: una animación

MetaCard presenta sus funcionalidades básicas para realizar animaciones en el tutorial “Tutorial sobre presentaciones” que incorpora la aplicación. Allí se plantea que se puede mover un campo de texto sobre el camino que había dibujado previamente con un gráfico. Sugiero al lector interesado que lo revise si no lo ha realizado ya. ¡Qué digo, no se vaya, vamos a improvisarlo sobre la marcha!

```
# Código de la tarjeta
local puntActual, elCami
on mouseUp
 put the points of graphic  "caminet" into elCami
 set the loc of btn "elProta" to the first line of elCami

 repeat with puntActual = 2 to the number of lines of elCami
    move btn "elProta" to line puntActual of elCami
 end repeat

 answer information "Soc bo: he aplegat!"
end mouseUp

# Código del botón “Parat!"
on mouseUp
  stop moving btn "elProta"
  answer information "Soc millor encara: l'he parat!"
end mouseUp
```

Listado 6: Un posible código para la implementación de la animación de un campo de texto usando eventos.

Creemos una pila y dispongamos sobre esta un gráfico (que yo llamo "caminet") con la herramienta *Graphic tool (polygon)* de la "*Menu Bar*" como el que muestra la fig. 3a. Sobre esta también situaremos, por ejemplo como en la 3b, un par de botones: uno ("elProta", que podemos decorar con alguno de los iconos de MetaCard y quitar el borde y el nombre del botón para que sea más divertido), que pretendemos mover por la ventana y otro ("Parat!") que pretendemos que pueda detener la acción de aquel. Para demostrar si "elProta" llega al final, mostraremos una caja de diálogo con alguna frase al respecto (fig. 3c), mientras que haremos algo similar (fig. 3d) al final del código del botón encargado de detener el proceso. Así todo el mismo será muy visual.

```
# Código de la tarjeta
global idMensatge
global puntActual, elCami
on mouseUp
 put the points of graphic  "caminet" into elCami
 put 1 into puntActual
 set the loc of btn "elProta" to the first line of elCami
 send "pegaUnPas" to me in 10 milliseconds
end mouseUp

on pegaUnPas
 if (puntActual = the number of lines of elCami)
   then send "hasAplegat" to btn "elProta" in 10 milliseconds
   else
     add 1 to puntActual
     move btn "elProta" to line puntActual of elCami
     send "pegaUnPas" to me in 500 milliseconds
     put the result into idMensatge
 end if
end pegaUnPas

# Código del botón “elProta"
 hasAplegat
 answer information "Soc bo: he aplegat!"
end hasAplegat

# Código del botón “Parat!"
global idMensatge
on mouseUp
  cancel idMensatge
  answer information "Soc millor encara: l'he parat!"
end mouseUp
```

Listado 7: Un posible código para la implementación de la animación de un objeto, sobre una ruta prefijada.

Todo puede empezar cuando se pulsa con el ratón sobre la tarjeta y acabará cuando el objeto de interés llegue al final del camino o cuando sea interrumpido, en cada caso el mensaje lo aclarará.

En el tutorial “Tutorial sobre presentaciones” mencionado, el objeto se mueve con el código que muestra el listado 6l y en el que hemos incluido la orden **stop** que hemos visto en la ayuda del **move**: Pruébelo y ya me cuenta …

3.1.3 Caso de estudio: revisando la animación

Pues bien, si no encuentra solución al problema anterior, le planteo una totalmente generalizable a otros casos donde la orden **stop**, ni ninguna otra, pudieran hacer nada por detener el proceso. La idea básica es crear eventos para iniciar las condiciones del bucles, hacer las acciones correspondientes a cada iteración y terminar. En el listado 7 aparecen respectivamente como "mouseUp", "pegaUnPas" y "hasAplegat".

Observe que entre ellas se pasan el turno con la orden **send** con un pequeño retraso, ahí se permite que se pueda interrumpir la cadena. El primero prepara los valores iniciales de las variables y da paso a la primera iteración del "bucle". El evento (bucle) principal se reenvía a si mismo el mensaje hasta que se da la condición de terminación. Entonces se ejecutará la única instrucción que había después del bucle.

4 Conclusiones

En este trabajo se han mostrado los elementos necesarios para la realización de aplicaciones que operan utilizando eventos en MetaCard, bajo el paradigma de desarrollo de aplicaciones multimedia de "tarjetas y guiones" (*cards & script*).

Las explicaciones se han acompañado de código para poder experimentar de forma práctica con la exposición. El lector interesado puede hacerse así una idea práctica de lo expuesto.

5 Bibliografía

[1] *MetaCard*. Disponible en <http://www.metacard..com/>