Tema 4 - Interacción y sistemas de visualización en RV y RA.

4.3 Sensores software. Sistemas de eventos y scripting.

Germán Arroyo, Juan Carlos Torres

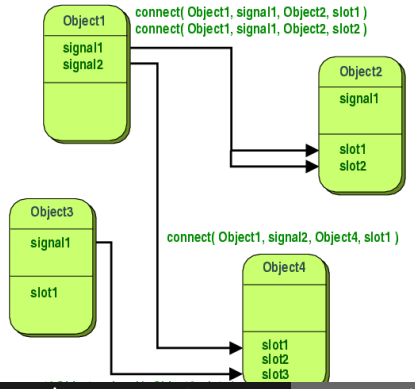
5 de febrero de 2021

Contenido del tema

4.3 Sensores software. Sistemas de eventos y scripting.

Técnicas de programación de sensores software (línea de tiempo, colisiones, fuerzas, UI, AI, etc.):

- Extensión (herencia): Button → mi_boton
- Callbacks (funciones): int event(void *obj_data, …);
- Señales.



Problema de la sincronización

Un simple «if» en el momento inadecuado puede romper un mundo virtual. Necesitamos sincronizar todas las partes (física, visualización, interacción, etc.).

Necesitamos un *observador* que vigile que todo se ejecuta en el orden correcto.

Patrón de diseño: Basado en Observador

El **Observador** (*observer*) es un objeto internamente síncrono (aunque externamente asíncrono) que acepta peticiones de los **sujetos**, las ordena y las ejecuta en el momento correspondiente.

Una vez hecha la petición, el **sujeto** no detiene su ejecución, sino que continúa con sus instrucciones.

Si necesita esperar, puede usar semáforos.

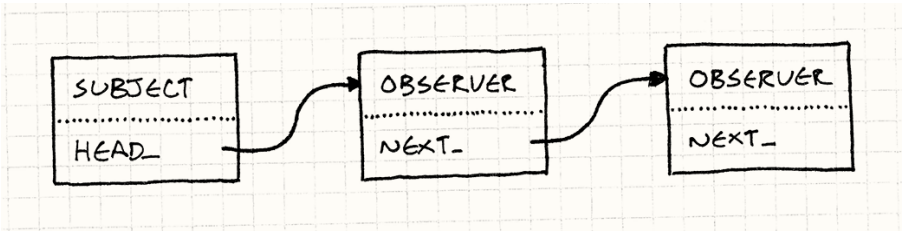
¡Problema!

¡Si un observador intenta cerrar el cerrojo que tiene un sujeto, el mundo virtual puede entrar en un bloqueo (*deadlock*)!

Observadores enlazados

En lugar de que un único observador se encargue de todas las partes, es común tener cada observador dedicado a un aspecto: física, visualización, etc.

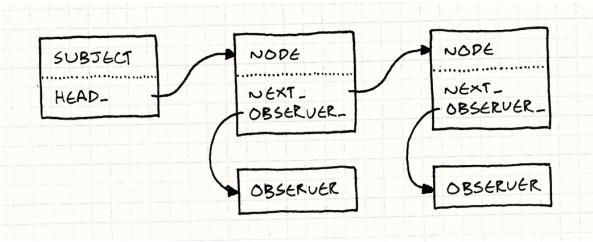
Los *sujetos* mantienen una lista de observadores (conectados bidireccionalmente).



Pool de observadores

Un observador podría estar en más de una de estas listas enlazadas.

Evitar duplicidad: se crea un *pool* de objetos en su lugar.



Problemas de los observadores

Aplicar observadores al problema equivocado: interfaces de usuario.
Borrar observadores: aún puede haber sujetos que lo invoquen.

- ¿Solución? Colector de basura: cada vez que hay un evento especial para un único objeto, se crea y se destruye (¿o no? → ¡se añade otra instancia!).

  The *lapsed listener problem*:
  https://en.wikipedia.org/wiki/Lapsed_listener_problem

¿Entonces?

Simplemente se les notifica cuando algo ha cambiado.
Lenguaje imperativo para modificar algo de la interfaz para reflejar un nuevo **estado**.

Señales

Las señales son una forma de emitir mensajes desde un objeto, mensaje con el que otros objetos pueden reaccionar.

Definición (emisor, solamente 1):

```
signal health_depleted
```

Conexión (emisor, solamente 1):

```
character_node.connect(«health_depleted», self, ”_on_Character_health_depleted”)
```

Recepción (receptores, más de uno):

```
func _on_Character_health_depleted():
…
```


Entity Component System (ECS)

El ECS da preferencia a la composición, antes que a la herencia de propiedades.

Otorga una flexibilidad enorme, ya que cualquier cosa puede ser una **entidad**.

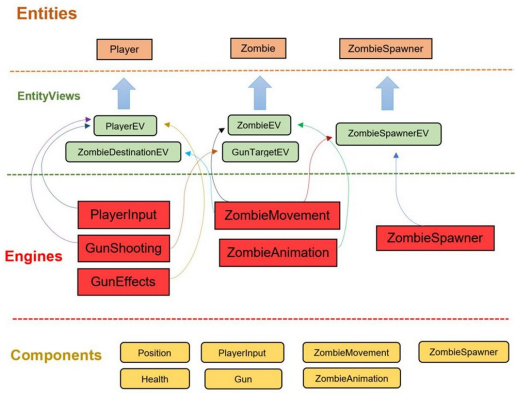


Figura 1: Esquema Entidad-Componente-Sistema.

Componentes de un ECS

Entidad: la entidad es un objeto de propósito general de identidad única (ID único).
Componente: los datos del objeto (desligados de la entidad) para un aspecto de la entidad (etiquetas de comportamiento).
Sistema: cada sistema se ejecuta constantemente, realizando las acciones globales para cada entidad que posea un determinado tipo de componente.

El código se escribe en los sistemas y no en las entidades o en los componentes.

Ejemplo de ECS

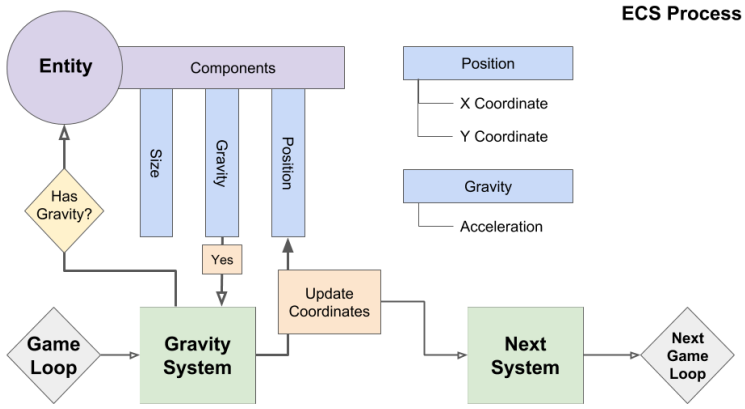


Figura 2: Ejemplo de Entity-Component-System en un entorno 2D.

Ejemplo de sistemas en un ECS

Los sistemas están enlazados, recordando la arquitectura del **Observador**.

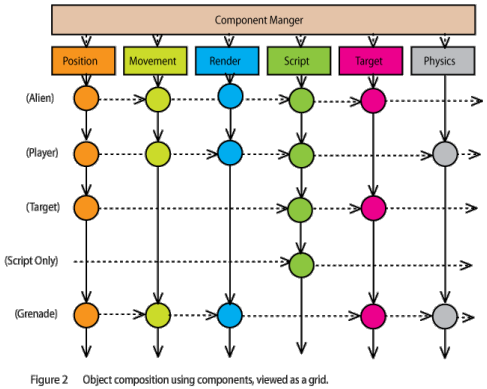


Figure 2 Object composition using components, viewed as a grid.

Figura 3: Ejemplo de conexión síncrona de sistemas.

Implementación mediante grupos (I)

A menudo los grupos funcionan como componentes. Son etiquetas asociadas a un nodo (que hace de entidad).

En el objeto que hace de entidad guardia:

```
add_to_group(«guards»)
```

Opcional: en algún evento podemos conectar etiqueta con *callback*:

```
get_tree().call_group(«guards», «enter_alert_mode»)
```


Implementación mediante grupos (II)

Dos opciones:

El mismo objeto hace de sistema (usando callback anterior):

```
func enter_alert_mode():
…
```

Otro objeto hace de sistema:

```
var guards = get_tree().get_nodes_in_group(«guards»)
```


Modo herramienta (*tool mode*)

Los scripts no se suelen ejecutar dentro del editor.

Solo las propiedades exportadas pueden modificarse.

```
export (int) var gravity
```

¿Y si queremos que se ejecute dentro del propio editor?

```
tool
```