8. Anexos

Anexo 1

Guía para la creación de comportamientos de máquinas de estados

Para crear un comportamiento de máquina de estados es tan fácil como crear un objeto del tipo '*BehaviourStateMachine'.* Este objeto tendrá dos constructores, el primero sin parámetros que permitirá crear máquinas de estados independientes, es decir, que no dependen de ninguna super máquina.

El segundo constructor sí necesita de un parámetro con el que indicar si será una sub-máquina o no. Este parámetro puede ser un booleano (*true* = es un sub-máquina, *false* = no es una sub-máquina) pero para mayor legibilidad se le ofrece al usuario utilizar el parámetro definido *IsASubmachine/IsNotASubmachine.* Veamos ejemplos de código:

- **Creación de máquina de estados independiente:**

```
new StateMachineEngine();
new StateMachineEngine(false);
new StateMachineEngine(BehaviourEngine.IsNotASubmachine);
```

- **Creación de submáquina de estados:**

```
new StateMachineEngine(true);
new StateMachineEngine(BehaviourEngine.IsASubmachine);
```

Una vez creada la máquina de estados tan solo habrá que llamar a su método de actualizado en cada iteración del bucle de actualización de nuestra aplicación o juego.

```
myFSM.Update();
```

Nota: este método de actualización tiene que ser llamado tanto para las supermáquinas como submáquinas

Anexo 2

Guía para la creación de estados

Los estados son elementos que permiten realizar las acciones al agente inteligente dentro de las máquinas de estados. Para poder crear estados se necesita haber creado primero una máquina de estados, ya que todos los métodos serán llamados desde la máquina.

El primer estado que hay que crear, ya que es obligatorio para que la máquina funcione correctamente, es el estado de entrada. Que será el primer estado al que entre la máquina una vez se inicialice. Este estado de entrada puede ser de dos tipos, un estado de entrada vacío que no haga ninguna tarea cuando se entre en él y un estado de entrada que sí realice una tarea. Para ello se sobrecarga la misma función, para crear un estado de entrada vacío tan solo habrá que pasarle un *string* con el nombre del estado y para crear un estado de entrada que realice cualquier acción habrá que pasarle también un nombre de estado y el nombre de una función de tipo *void* (si devuelve cualquier otro tipo no funcionará) que será la acción que realizará. El código sería algo así:

- **Creación de un estado de entrada vacío:**

```
myFSM.CreateEntryState("state name");
```

- **Creación de un estado de entrada que realiza una acción:**

```
myFSM.CreateEntryState("state name", myAction);
```

Los siguientes estados que se pueden crear son los estados normales, aquellos a los que se accede a través de una transición desde otro estado y que pueden realizar, o no, una acción. Hay tres formas de crear estos estados, según las propiedades de cada uno, pero todos siguen la misma estructura que los estados de entrada.

Está el estado vacío, el cual no realiza ninguna acción y solo requiere que se le pase un nombre. El estado convencional que realiza una acción requiere de un nombre y una función de tipo *void* (si devuelve cualquier otro tipo no funcionará) que definirá la acción que realizará. Y, por último, se encuentra el estado que realiza una acción a la que se le puede pasar una percepción (más adelante veremos cómo crearlas) como parámetro para almacenar u obtener valores previamente ya calculados.

- **Creación de un estado vacío:**

```
myFSM.CreateState("state name");
```

- **Creación de un estado que realiza una acción:**

```
myFSM.CreateState("state name", myAction);
```

- **Creación de un estado que realiza una acción y se le pasa una percepción cómo parámetro:**

```
myFSM.CreateState("state name", () => myAction(perception));
```

El último tipo de estado que se puede crear es el estado que contiene a una sub-máquina en su interior, con este estado podremos crear máquinas de estados jerárquicas con gran facilidad. Estos estados requieren de un nombre y la sub-máquina a la que van a entrar, esto hará que al entrar en este estado contenedor se ejecute directamente el estado de entrada de la submáquina, si, por el contrario, el usuario quiere entrar en un estado concreto de la sub-máquina tan solo tendrá que añadir como parámetro el estado al que quiere acceder. Los dos métodos para crear estos estados son:

- **Creación de un estado contenedor de una submáquina que entra al estado de entrada:**

```
myFSM.CreateSubStateMachine(“state name”, subBehaviour);
```

- **Creación de un estado contenedor de una submáquina que entra a un estado concreto:**

```
myFSM.CreateSubStateMachine(“state name”, subBehaviour,
                            submachine state);
```

NOTA IMPORTANTE: las submáquinas siempre han de crearse antes que las supermáquinas, ya que necesitan tener la referencia a ellas.

Anexo 3

Guía para la creación de percepciones

Ahora veremos cómo crear percepciones, las percepciones se usan para lanzar las transiciones cuando se cumpla una condición. Según el tipo de condición que queramos evaluar deberemos de usar un tipo de percepción u otro o incluso hacerse una percepción propia. Ahora veremos los diferentes tipos de percepción y para qué sirven cada uno y cómo crear una percepción propia:

- **Percepción de tipo 'push':** esta percepción se lanza manualmente por el usuario, otro agente del entorno o incluso por el propio entorno. Son percepciones que se quieren lanzar cuando ocurra un evento o acción concreta (pulsar una tecla, abrir una puerta…)

```
myFSM.CreatePerception<PushPerception>();
```

para lanzarlas hay que hacerlo por código, lanzando la transición que controlan (ver Anexo 4 para crear transiciones):

```
myFSM.Fire(transition);
      o
myFSM.Fire("transition name");
```

- **Percepción de tipo 'timer':** esta percepción lanza la transición automáticamente una vez pasado el tiempo que se le indique, este tiempo empieza a contar una vez entre dentro del estado salida de la transición. Esta percepción necesita que se le pase como parámetro un valor de tipo *float* que indicará los segundos que deben pasar antes de lanzar la transición.

```
myFSM.CreatePerception<TimerPerception>(segundos);
```

- **Percepción de tipo 'value':** esta percepción lanza la transición automáticamente cuando se cumpla la evaluación o evaluaciones que se le pasen. Estas evaluaciones han de ser comparaciones booleanas del estilo: mayor que, menor que, igual que… si se decide añadir más de una comparación en la percepción, se deberán de cumplir todas a la vez para que la transición se lance. Aunque se llamen 'value' permiten variables de cualquier tipo que permitan realizar una expresión booleana.

```
myFSM.CreatePerception<ValuePerception>(() => myLife <= 0);
myFSM.CreatePerception<ValuePerception>(() => var1 < 0,
                                        () => var2 == true);
```

- **Percepción de tipo 'is in state':** esta percepción comprueba si una máquina de estados se encuentra en un estado en concreto, de ser así, lanza la transición, si no, espera a que la máquina de estados llegue a ese estado. Los parámetros que necesita son: la máquina de estados donde comprobará el estado en el que se encuentra y el nombre del estado al que tiene que llegar para lanzar la transición.

```
myFSM.CreatePerception<IsInStatePerception>(StateMacine to
                look in, “name of the state to look for”);
```

- **Percepción de tipo ‘Behaviour tree status’:** esta percepción es parecida a la anterior, pero comprueba el estado del nodo raíz de un árbol de comportamientos. Lanzará la transición cuando el estado del nodo raíz del árbol de comportamientos sea el mismo que indique el usuario. Los parámetros que necesita son: el árbol de comportamientos que comprobará y el estado al que espera llegar.

```
myFSM.CreatePerception<BehaviourTreeStatusPerception(behav
                    iour tree to check, status to reach);
```

- **Percepción de tipo ‘And’:** este tipo de percepción permite unir dos percepciones de cualquier tipo, haciendo que la transición sólo sea ejecutada cuando las dos percepciones cumplan sus respectivas condiciones a la vez. Esta transición necesita como parámetros dos percepciones que pueden ser de cualquier tipo, incluso creadas por el usuario.

```
myFSM.CreateAndPerception<AndPerception>(perception left,
                                        perception right);
```

- **Percepción de tipo ‘Or’:** este tipo de percepción permite unir dos tipos de percepciones cualesquiera y lanzará la transición cuando tan solo una de ellas se cumpla. Como parámetros necesitará dos tipos de percepciones de cualquier tipo, que incluso pueden ser percepciones creadas por el propio usuario.

```
myFSM.CreateOrPerception<OrPerception>(perception left,
                                       perception right);
```

- **Percepciones creadas por el usuario:** cómo puede haber infinitas percepciones, según el elemento que se quiera comprobar, el usuario podrá crear sus propias percepciones para poder automatizar las transiciones que él quiera. Tan solo deberá seguir unos sencillos pasos:

1. El usuario crearía una clase propia con el nombre de la percepción que quiera y hará que herede de la clase padre **Percepcion**.

```
MyPerception : Perception() { … }
```

2. El siguiente paso será sobrescribir el método *‘Check’* de la clase padre, el cual es necesario para que la percepción se evalúe.

```
public override bool Check() { … }
```

3. El próximo paso es opcional, ya que no todas las percepciones lo necesitarán, es sobrescribir el método *‘Reset’*, si la percepción del usuario no necesita reiniciar ningún

valor y no sobrescribe este método no pasará nada. Solo es necesario cuando el usuario sí quiera reiniciar un valor cuando se lance la transición controlada por su percepción.

```
public override void Reset() { … }
```

4. Este paso también es opcional, dependiendo de las necesidades de la percepción, y será implementar los métodos *'get'* y *'set'* en caso de que la percepción necesite alguno de ellos para almacenar distintos valores propios de la máquina de estados.

```
public type get() { … }
public void set() { … }
```

5. Con estos pasos ya estaría creada la percepción en sí. Ahora solo faltaría añadirla a la máquina de estados correspondiente.

```
MyPerception perception =
myFSM.CreatePerception<MyPerception>(new MyPerception());
```

Anexo 4

Guía para la creación de transiciones en máquinas de estados

Las transiciones se utilizan para pasar de un estado a otro dentro de una misma máquina de estados, son lanzadas por la percepción que tienen asignada, la cual las lanzará cuando se cumpla su condición. Para crear una transición se necesita: el nombre de la transición, el estado desde el que sale, la percepción encargada de lanzarla y el estado al que va. El código se verá tal que así:

- **Creación de una transición:**

```
myFSM.CreateTransition("transition name", state from,
                       perception, state to);
```

Las transiciones anteriores sirven para pasar de un estado a otro, pero siempre que estén dentro de una misma máquina de estados. Si se quiere volver de una submáquina a su supermáquina se deberá usar una transición de salida. Hay varias transiciones de salida dependiendo del tipo de supermáquina a la que se quiera volver. De momento veremos la transición que sólo devuelven a máquinas de estados. Es una transición normal en cuanto a los parámetros que necesita, ya que necesita: el nombre de la transición, el estado de la submáquina desde el que sale, la percepción que la lanzará y el estado de la supermáquina a donde llegará.

- **Creación de una transición de salida, desde cualquier comportamiento a una máquina de estados:**

```
mySubFSM.CreateExitTransition("transition name", state from,
                       perception, supermachine state to);
```

NOTA IMPORTANTE: las transiciones de salida han de ser lanzadas siempre desde la submáquina

Anexo 5

Guía para la creación de árboles de comportamientos

Ahora veremos cómo crear árboles de comportamientos. Tienen una estructura y sintaxis similar a la de las máquinas de estados para intentar mantener una homogeneidad entre todas las distintas técnicas de la librería, salvando las diferencias que hay entre ellas.

Los árboles de comportamientos también se tienen que definir si son submáquinas o no al ser creados, se definen de la misma manera que las máquinas de estados.

- **Creación de árboles de comportamientos independientes:**

```
new BehaviourTreeEngine();
new BehaviourTreeEngine(false);
new BehaviourTreeEngine(BehaviourEngine.IsNotASubmachine);
```

- **Creación de subárboles de comportamientos:**

```
new BehaviourTreeEngine(true);
new BehaviourTreeEngine(BehaviourEngine.IsASubmachine);
```

Cuando se haya creado el árbol de comportamientos y todos los nodos que lo compondrán (explicados en la siguiente guía) se tendrá que definir el nodo raíz del árbol mediante la siguiente función:

```
myBT.SetRootNode(any node);
```

Una vez creado el árbol de comportamientos tendremos que llamar a su método de actualización en cada iteración del bucle de actualización de nuestra aplicación o juego.

```
myBT.Update();
```

Nota: este método de actualización tiene que ser llamado tanto para las superárboles como subárboles

Anexo 6

Guía para la creación de nodos del árbol de comportamientos

Ahora veremos cómo crear los distintos nodos de los que se puede componer un árbol de comportamientos y que tipo de acción realiza cada uno.

- **Nodo secuencia:** este nodo va entrando en orden en sus distinto hijos mientras estos devuelvan un estado de 'éxito', en el momento en el que uno de sus nodos hijo devuelva un estado de 'fallido' parará la secuencia y el resultado final será de 'fallido', si por el contrario consigue pasar por todos los nodos hijos y que estos devuelvan un 'éxito' el estado final será de 'éxito'.
  Los nodos secuencia pueden recorrer sus hijos en el orden en el que se han añadido o bien reordenarlos aleatoriamente para que los recorra en un orden aleatorio.

  Nodo secuencia en el orden en el que se han añadido:

  ```
  myBT.CreateSequenceNode("name", false);
  ```

  Nodo secuencia en orden aleatorio:

  ```
  myBT.CreateSequenceNode("name", true);
  ```

Para añadir nodos hijos al nodo secuencia tan solo habrá que usar el siguiente método, hay que recordar que el orden en el que se añadan los nodos hijos es el orden en el que se realizará la secuencia.

```
mySequenceNode.AddChild(any node);
```

- **Nodo selector:** este nodo también ejecuta sus nodos hijos en orden pero con la diferencia de que para avanzar al siguiente nodo, el nodo hijo tiene que devolver un estado de 'fallido', en el momento en el que el nodo hijo devuelva un 'éxito' la secuencia parará y el nodo secuencia devolverá un estado de 'éxito', si por el contrario, pasa por todos los nodos hijos y todos devuelven un estado de 'fallido' el nodo secuencia finalizará con un estado de 'fallido'.
  Los nodos selectores siempre recorren sus hijos en el orden en el que se han añadido

```
myBT.CreateSelectorNode("name);
```

Para añadir cualquier nodo hijo al nodo selector tan solo habrá que añadirlo mediante la función:

```
mySelectorNode.AddChild(any node);
```

- **Nodo bucle:** este nodo decorador tan solo permite tener un nodo hijo que ejecutará tantas veces cómo se le diga o infinitas veces si no se le pasa ningún número de repeticiones.

  Nodo bucle infinito:

```
myBT.CreateLoopNode("name", child node);
```

Nodo bucle finito:

```
myBT.CreateLoopNode("name", child node, loop times);
```

- **Nodo bucle hasta que falle:** este nodo decorador ejecutará su nodo hijo indefinidamente hasta que devuelva un estado de 'fallido'. En el momento que falle dejará de ejecutarlo y el estado decorador devolverá un estado de 'éxito'.

```
myBT.CreateLoopUntilFailNode("name", child node);
```

- **Nodo 'timer':** este nodo ejecutará su nodo hijo una vez haya pasado el tiempo indicado por el usuario, en segundos.

```
myBT.CreateTimerNode("name", child node, time in seconds);
```

- **Nodo inversor:** el nodo inversor se encarga de invertir el estado que le ha llegado de su nodo hijo, por el contario. Es decir, si le llega un estado de 'éxito' lo invertirá por un 'fallido' y si le llega un estado de 'fallido' lo invertirá por un estado de 'éxito'.

```
myBT.CreateInverterNode("name", child node);
```

- **Nodo de éxito:** este nodo modificará el estado que le llegue de su nodo hijo, sea cual sea, por un estado de 'éxito'. Se puede combinar con un nodo inversor para que modifique el estado por un estado de 'fallido'.

```
myBT.CreateSucceederNode("name", child node);
```

- **Nodo condicional:** este nodo ejecutará su nodo hijo, pero devolverá un estado de éxito o fallido en función de si cumple o no la condición. Esta condición será creada en forma de percepción, que puede ser de cualquier tipo tanto de las ya creadas como de las creadas por el usuario.

```
myBT.CreateConditionalNode("name", child node, condition);
```

- **Nodo hoja:** estos nodos son muy parecidos a los estados de las máquinas de estados, ya que realizan una acción (la función que realizará la acción debe ser de tipo *void*) y devuelven un estado dependiendo de si la han cumplido o no. Estos nodos siempre han de ser nodos hoja de los árboles de comportamiento por lo que nunca podrán tener hijos asignados.
  La función de evaluación para saber si se ha cumplido la acción o no deberá de ser creada por el usuario y deberá devolver uno de los tres estados de retorno (en proceso, éxito o fallido).

```
myBT.CreateLeafNode(“name”, action, evaluation function);
```

En la API los valores de retorno (en proceso, éxito o fallido) son accesibles mediante un enumerador:

```
ReturnValues.Running;
ReturnValues.Succeed;
ReturnValues.Failed;
```

Ahora veremos los nodos hoja que permiten contener en su interior cualquier tipo de comportamiento en forma de submáquina. Estas submáquinas serán lanzadas automáticamente cuando se entre en el nodo hoja que las contiene, y para salir de ellas de vuelta al súper árbol se deberán de usar las transiciones de salida.

- **Crear nodo hoja con una submáquina dentro:** este nodo hoja permite incorporar cualquier submáquina dentro de él. Una vez se entre en este nodo se entrará directamente en el estado de entrada de la submáquina o el nodo raíz, en caso de ser un subárbol.

```
myBT.CreateSubBehaviour(“name”, sub-machine/sub-behaviour tree);
```

- **Crear nodo hoja con una submáquina dentro yendo a un estado en concreto:** este nodo hoja sólo permite incorporar una submáquina de estados, pero el usuario podrá decidir a qué estado quiere ir cuando se entre en la submáquina, sin tener que pasar por el estado de entrada.

```
myBT.CreateSubBehaviour(“name”, sub-state machine, state
to);
```

Anexo 7

Guía para la creación de transiciones en árboles de comportamientos

Aunque todas las transiciones entre nodos de los árboles de comportamientos se lanzan automáticamente, sin que el usuario tenga que realizar ninguna acción, hay unas transiciones en concreto en las que el usuario tiene que intervenir para poder especificar el estado de retorno que tendrá el nodo al que se llega. Estas transiciones son las transiciones de salida de una submáquina al super árbol de comportamientos que la contiene.

- **Transición de salida de una submáquina de estados a un árbol de comportamientos:** esta transición se define cuando se quiere salir de un estado cualquiera, de una submáquina de estados, al nodo contenedor del árbol de comportamientos. Requiere que se le pase como parámetros: el nombre de la transición, el estado desde el que se va a volver, la percepción que lanzará la transición y el estado que obtendrá el nodo contenedor cuando se vuelva a él (exitoso o fallido).

```
mySubBT.CreateExitTransition(“transition name”, state
from, perception, return value);
```

- **Transición de salida de un subárbol de comportamientos a un árbol de comportamientos:** esta transición se define cuando se quiere salir de un subárbol de comportamientos al árbol de comportamientos que lo contiene. En este caso la transición se lanzará automáticamente cuando el nodo raíz del subárbol de comportamientos tenga un estado de ‘éxito’ o ‘fallido’ y este estado será el que devolverá al nodo hoja que lo contiene, en el super árbol de comportamientos. Debido a que todo ya está automatizado tan solo necesita un nombre de transición.

```
mySubBT.CreateExitTransition(“transition name”);
```

NOTA IMPORTANTE: las transiciones de salida han de ser lanzadas siempre desde el subárbol

---

Además de estas guías se ofrecen al usuario algunas escenas con comportamientos básicos y su código fuente, para que puedan ver de primera mano como crear un comportamiento específico y poder tomarlo de referencia si lo necesitan.