7. ANEXOS

Tutorial para crear un Utility System Engine sencillo

Al igual que las otras máquinas, crear ésta es bastante parecido. El único cambio es el valor de inercia que hay que opcionalmente introducir. Por defecto, éste valor es de 1.3f, es decir, modifica al alza un 30% del valor de la acción en curso. Si se quiere desactivar este efecto, habría que introducir un valor de 1.

Máquina con inercia de 1.3f y sin ser submáquina:

```
UtilitySystemEngine utilEngine = new UtilitySystemEngine();
```

Máquina con inercia de 1.3f siendo submáquina:

```
UtilitySystemEngine utilEngine = new UtilitySystemEngine(true);
```

Máquina con inercia personalizada y sin ser submáquina:

```
UtilitySystemEngine utilEngine = new UtilitySystemEngine(false, 1.0f);
```

Tutorial para crear factores y acciones

Lo primero que hay que saber es cómo crear factores. En esta sección se explica como crear todos los tipos de factores y sus peculiaridades y, posteriormente, cómo crear las acciones a partir de un Utility System.

LeafVariable: es el factor base, el que recibe una referencia del entorno, su valor máximo y mínimo y la normaliza. Crearlo es sencillo, recibe una función que devuelve un valor flotante, el valor flotante máximo y mínimo.

```
Factor nombreLeaf = new LeafVariable(() => valorFlotanteDelEntorno,
flotanteMaximo, flotanteMinimo)
```

En la función flecha se podrían hacer más cosas, es recomendable únicamente usarlo para pasar la referencia al valor del entorno.

LinearCurve: es un factor que recibe otro como argumento y lo modifica linealmente. Sus parámetros sirven para construir una función lineal, éstos son: la pendiente y la ordenada en el origen, los cuales son parámetros opcionales.

```
Factor nombreLinear = new LinearCurve(factor);
```

```
Factor nombreLinear = new LinearCurve(factor, pendiente);
Factor nombreLinear = new LinearCurve(factor, pendiente, ord_origen);
```

ExpCurve: es un factor que recibe otro como argumento y lo modifica exponencialmente. Sus parámetros sirven para construir una función exponencial, éstos son: el exponente, el desplazamiento horizontal y el desplazamiento vertical, siendo los tres opcionales.

```
Factor nombreExp = new ExpCurve(factor);
Factor nombreExp = new ExpCurve(factor, exponente);
Factor nombreExp = new ExpCurve(factor, exponente, despX);
Factor nombreExp = new ExpCurve(factor, exponente, despX, despY);
```

LinearPartsCurve: es un factor que recibe otro como argumento y lo modifica linealmente por partes. Sus parámetros sirven para construir la función lineal por partes, recibe una lista de puntos (deben contener valores normalizados, entre 0 y 1). Primero hay que crear los puntos y añadirlos a una lista y, posteriormente, crear este factor.

```
// Puntos
List<Point2D> points = new List<Point2D>();
points.Add(new Point2D(0, 0));
points.Add(new Point2D(0.2f, 0.4f));
points.Add(new Point2D(0.6f, 0.8f));
points.Add(new Point2D(1, 1));

// Factor
Factor nameParts = new LinearPartsCurve(factor, points);
```

MaxFusion y *MinFusion:* son factores que reciben una lista de factores como argumento y elige el que mayor o menor utilidad otorgue, devolviendo su valor resultado.

```
// Factores
List<Factor> factors = new List<Factor>();
factors.Add(factor1);
factors.Add(factor2);

// MaxFusion
Factor maxFusionFactor = new MaxFusion(factors);
// MinFusion
Factor minFusionFactor = new MinFusion(factors);
```

WeightedSumFusion: es un factor que recibe una lista de factores como argumento y una lista de pesos para modificar el peso asignado a cada factor, devolviendo un único valor fusionado.

```
// Factores
List<Factor> factors = new List<Factor>();
factors.Add(factor1);
factors.Add(factor2);

// Pesos
List<float> weights = new List<float>();
weights.Add(0.4f);
weights.Add(0.6f);

// Weighted Sum 2 opciones
Factor weightFactor = new WeightedSumFusion(factors);
Factor weightFactor = new WeightedSumFusion(factors, weights);
```

Las acciones de utilidad reciben un factor para poder ser creados, por ello se ha decidido colocarlos en esta sección. Se pueden crear tres tipos de acciones sin submáquina:

- Acciones simples: son aquellas que contienen un factor y una referencia a método para ejecutar y nada más.

```
utilitySystem.CreateUtilityAction("nombre", metodo_a_ejecutar,
factor);
```

- Acciones específicas que contienen una transición directa al nodo hoja del árbol de comportamientos, sin método a ejecutar:

```
utilitySystem.CreateUtilityAction("nombre", factor, valorRetornado,
behaviourTree);
```

- Acciones específicas que contienen una transición al nodo hoja del árbol de comportamientos que espera a que una función devuelva un ReturnValues distinto de “Running” y con un método a ejecutar:

```
utilitySystem.CreateUtilityAction("nombre", factor, metodo_a_ejecutar,
func_valorRetornado, behaviourTree);
```

Tutorial para crear submáquinas dentro del Utility System

Cualquier submáquina puede ser creada a partir de dos funciones que tiene el UtilitySystem:

➢ Crear una submáquina entrando a su estado inicial:

```
utilitySystem.CreateSubBehaviour("name", factor, subBehaviour);
```

➢ Crear una submáquina entrando a un estado específico:

```
utilitySystem.CreateSubBehaviour("name", factor, subBehaviour,
stateTo);
```

Tutorial para salir de un UtilitySystem a cualquier estado de un FSM

El método a utilizar es el mismo que se usa para salir de cualquier submáquina de un FSM a un estado del FSM. La sentencia a utilizar sería la siguiente:

```
utilitySystem.CreateExitTransition("nombre", stateFrom, Perception,
stateTo);
```

El estado “stateFrom” hace referencia al estado contenedor del **UtilitySystemEngine**.

Tutorial para salir correctamente de un Utility System submáquina al LeafNode contenedor del Utility System

Hay dos formas de salir al LeafNode contenedor de un Utility System. La primera manera es crear un UtilityAction sin método a ejecutar que salga directamente al LeafNode con el valor retornado asignado por el usuario, por ejemplo:

```
utilitySystem.CreateUtilityAction("Exit", factor_cualquiera,
ReturnValues.Succeed, behaviourTree);
```

La otra manera es crear un UtilityAction con método a ejecutar que espere a que una función devuelva otro valor que no sea ReturnValues.Running. Por ejemplo:

```
private ReturnValues IsRecipeCreated(){
      if(Vector3.Distance(transform.position, tablePosition.position)
< 0.1f) {
            transform.LookAt(pizzaPosition);
            recipeAnimator.ResetTrigger("Reset");
            return ReturnValues.Succeed;
      } else {
```

```
            return ReturnValues.Running;
}}

utilityCurves.CreateUtilityAction("exit_ac1", pepperoniSum, () => {
CreateRecipe(1); }, IsRecipeCreated, behaviourTree);
```

El método *Update()* de **UtilityAction** se encarga de gestionar si se debe hacer la transición o no:

```
public void Update(){
      // Valor devuelto al árbol de comportamientos, una vez devuelva
un valor distinto de Running
      if(this.valueReturned != null){
            ReturnValues returnValue = this.valueReturned();
            if (returnValue != ReturnValues.Running){
                  new Transition("Exit_Action_Transition",
this.utilityState, new PushPerception(this.uCurvesEngine),
this.uCurvesEngine.NodeToReturn,
                              returnValue, this.bt,
this.uCurvesEngine)
                              .FireTransition();
}}}
```

Tutorial para salir de las submáquinas al Utility System

Para salir de cualquier submáquina (a excepción de los buckets) al UtilitySystemEngine, se usa la siguiente sentencia desde la submáquina:

```
subMachine.CreateExitTransition("Exit_Transition", stateFrom,
Perception, utilEngine);
```

La diferencia entre los FSM y los Behaviour Trees es que en estos últimos el nombre de la transición es obligatorio poner “Exit_Transition”. Además, en los Behaviour Trees lo ideal es usar un BehaviourTreeStatusPerception.

Además, el Utility System gestiona si debe cambiar entre una UtilityAction u otra, a pesar de ser la submáquina la que esté activa, de forma que las transiciones de submáquina a supermáquina se realizan dinámicamente también.