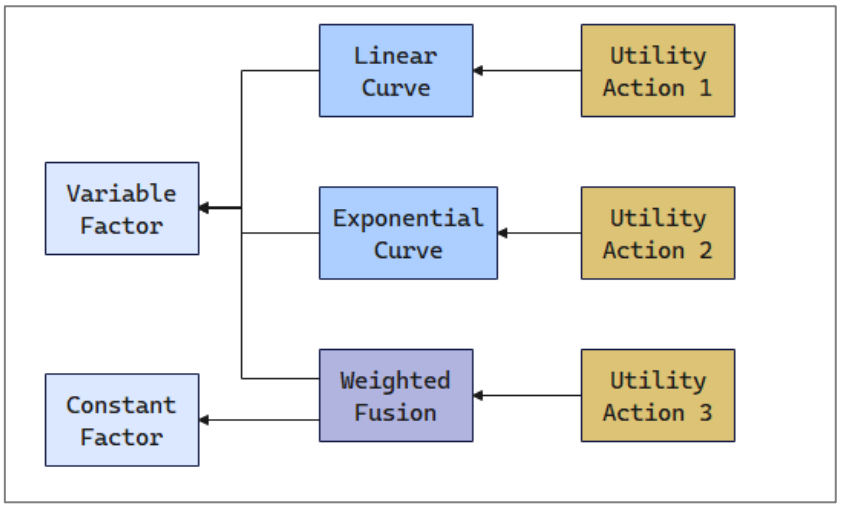
GUIA BÁSICA DE CREACIÓN DE SISTEMAS DE UTILIDAD

Crear y ejecutar un sistema de utilidad

Esta guía muestra los pasos a seguir para crear el siguiente sistema de utilidad.



Para crear un sistema de utilidad se crea un objeto de la clase `UtilitySystem`. Se pasa como parámetro opcional la inercia del sistema, que es un multiplicador para la utilidad del elemento seleccionado para facilitar que este se mantenga estable. Por defecto es 1.3.

```
UtilitySystem us = new UtilitySystem();       // Inercia 1.3
UtilitySystem us = new UtilitySystem(1.5f);   // Inercia 1.5
```

En los sistemas de utilidad se crean primero los factores de izquierda a derecha y luego el resto de los nodos.

Para ejecutar el sistema de utilidad se debe lanzar el método Start al comienzo, y después ejecutar el método Update en cada iteración.

```
UtilitySystem us = new UtilitySystem();  // Al principio de la ejecución se crea el grafo.
us.Start();  // En el primer frame se inicia la ejecución del grafo (p.e. Start en Unity).
us.Update(); // En cada bucle de ejecución se actualiza el grafo (p.e. Update en Unity):
```

***NOTA**: Estos métodos sólo deben ejecutarse directamente en el grafo principal, no en los subgrafos.*

Factores hoja

Los factores hoja sirven para obtener un valor de utilidad del entorno y el resto de factores calculan su utilidad en base a estos. Todos los tipos de factores limitan su valor al intervalo [0 - 1].

Factores hoja variables

Estos factores obtienen su utilidad de una variable o función y normalizan su valor en una escala 0-1.

Para crear un factor variable se usa el método `CreateVariable` del sistema de utilidad y se pasa como parámetros una función que devuelva un valor `float`, y dos valores `floats` que representan el valor mínimo y máximo que puede devolver la función. Por defecto estos valores son 0 y 1.

En el ejemplo inicial se crea un factor variable. La utilidad del factor cambiará cuando cambie el valor de "value".

```
float value = 5f;
VariableFactor v = us.CreateVariable(() => value, 0f, 10f); // Devolverá (value-0)/(10-0) = 0.5
```

Factores hoja constantes

Para crear un factor constante se usa el método `CreateConstant` y se pasa como parámetro el valor deseado. La utilidad del factor siempre será un valor entre 0 y 1, aunque el valor introducido sea mayor o menor.

En el ejemplo inicial se crea un factor constante. Este tipo de factores es útil para crear una "acción por defecto" que se seleccione si el resto no llegan al valor.

```
ConstantFactor v = us.CreateConstant(0.7f); // Devolverá siempre 0.7
```

Factores curva

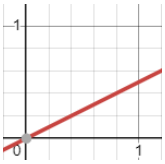
Los factores curva o curvas de utilidad sirven para modificar la utilidad de un factor hijo con una función. Para crearlos se usa el método create curve pasando el nodo factor hijo.

En el ejemplo inicial se crean dos factores curva de tipo lineal y exponencial respectivamente, ambos modificando la utilidad del factor hoja variable.

```
LinearCurveFactor lf = us.CreateCurve<LinearCurveFactor>(variableFactor);
ExponentialCurveFactor ef = us.CreateCurve<ExponentialCurveFactor>(variableFactor);
```

Factor curvo lineal

Modifica la utilidad de su factor hijo con una función lineal. Se puede personalizar la pendiente (por defecto 1) y la ordenada en el origen (por defecto 0) de la función.

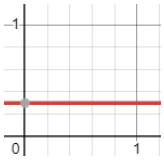


Slope = 0.5

YIntercept = 0

Slope = -0.5

YIntercept = 0.8



Slope = 0

YIntercept = 0.3

```
LinearCurveFactor lf = us.CreateCurve<LinearCurveFactor>(factor);
lf.Slope = -0.5f
lf.YIntercept = 0.8f;
```

Factor curva exponencial

Modifica la utilidad de su factor hijo con una función exponencial. Se puede personalizar el exponente (por defecto 1), el desplazamiento horizontal y el desplazamiento vertical de la función (por defecto 0).

Exponent = 0.5

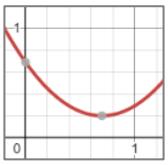
DespX = -0.2

DespY = -0.3

Exponent = 1.8

DespX = 0.5

DespY = 0.4



Exponent = 2

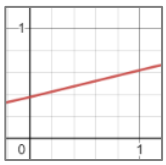
DespX = 0.7

DespY = 0.2

```
ExponentialCurveFactor ef = us.CreateCurve<ExponentialCurveFactor>(factor);
ef.Exponent = 0.5f
ef.DespX = -0.2f;
ef.DespY = -0.3f;
```

Factor sigmoide

Modifica la utilidad de su factor hijo con una función sigmoide. Se personaliza el coeficiente de crecimiento (por defecto 1) y el punto medio (por defecto 0.5).



GrownRate = 1

MidPoint = 0.5

GrownRate = 10

MidPoint = 0.2

GrownRate = -7

MidPoint = -0.3

```
SigmoidCurveFactor sf = us.CreateCurve<SigmoidCurveFactor>(factor);
sf.GrownRate = -7f
sf.MidPoint = -0.2f;
```

Factor curva a puntos

Crea la función con la que modifica la utilidad de su factor hijo mediante puntos que forman una función lineal por partes. Se puede especificar la lista de puntos que forman la función. Para evitar errores estos puntos deben pasarse ordenados en su coordenada x.

P1(0, 0.5)
P2(0.5, 0.8)
P3(1, 0.2)

```
PointedCurveFactor pf = us.CreateCurve<PointedCurveFactor>(factor)
pf.Points = new List<CurvePoint>(){
        new CurvePoint (0,0.4f), new CurvePoint (0.5f,0.8f), new Vector2(1,0.2f)
};
```

Factor curva personalizado

Modifica la utilidad de su factor hijo con una función especificada por el usuario. Se especifica la función con la variable `Function`.

```
CustomCurveFactor cf = us.CreateCurve<CustomCurveFactor>(factor);
cf.Function = (x) => x * x + 0.5f;
```

Factores fusión

Los factores fusión permiten combinar la utilidad de varios factores en uno solo. Para crear un factor fusión se usa el método `CreateFusion`, pasando como parámetro genérico el tipo de nodo fusión y como argumentos los factores hijos. Los factores hijos pueden pasarse por separado o como una lista directamente. En el ejemplo inicial se crea un factor fusión que calcula la suma ponderada de las utilidades de los dos factores hoja.

```
WeightedFusionFactor weightedFusion = us.CreateFusion<WeightedFusionFactor>(factor1, factor2);
weightedFusion.Weights = new float[]{0.7f, 0.3f};
```

MinFusionFactor

Devuelve la utilidad mínima de los factores hijos.

```
MinFusionFactor min = us.CreateFusion<MinFusionFactor>(factor1, factor2, ...);
```

MaxFusionFactor

Devuelve la utilidad máxima de los factores hijos.

```
MaxFusionFactor min = us.CreateFusion<MaxFusionFactor>(factor1, factor2, ...);
```

WeightedFusionFactor

Devuelve la media ponderada de la utilidad de los factores hijos. Se especifican los pesos en la variable `Weights`.

```
WeightedFusionFactor weighted = us.CreateFusion<WeightedFusionFactor>(factor1, factor2, ...);
weighted.Weights = new float[]{0.2f, 0.3f, …};
```

Acciones de utilidad

Las acciones de utilidad son nodos que el sistema de utilidad puede seleccionar para ejecutar una acción. Para crear una acción de utilidad se usa el método `CreateAction` y se pasa como parámetros el factor para calcular su utilidad y la acción que ejecutan.

Como crear acciones:
Para crear la acción hay que añadir el espacio de nombres `BehaviourAPI.Core.Actions` y crear el objeto:

```
FunctionalAction action = new FunctionalAction(StartMethod, UpdateMethod, StopMethod);
```

Los parámetros del constructor de `FunctionAction` son métodos que se ejecutarán cuando el estado comience su ejecución, en cada frame y cuando termine respectivamente. Los métodos `Start` y `Stop` deben devolver `void` y el método `Update` debe devolver `Status`.

Se pueden crear objetos de la clase FunctionalAction especificando solo algunos de los 3 métodos. Si no se especifica el método Update, la acción devolverá siempre Running y no terminará nunca.

En el ejemplo de esta guía se crean tres acciones de utilidad, cada una asignada a un factor concreto.

```
UtilityAction utilityAction1 = us.CreateAction(linearCurve, action1);
UtilityAction utilityAction2 = us.CreateAction(exponentialCurve, action2);
UtilityAction utilityAction3 = us.CreateAction(weightedFusion, action3);
```

Crear subgrafos con sistemas de utilidad.

Crear un subgrafo dentro de un sistema de utilidad

Para ejecutar un subgrafo en un sistema de utilidad hay que crear una acción de tipo `SubsystemAction` pasando como parámetro el subgrafo y asignar la acción a un nodo de tipo `UtilityAction`.

```
UtilitySystem mainUS = new UtilitySystem();
BehaviourGraph subgraph = ...;
SubsystemAction action = new SubsystemAction(subgraph);
UtilityAction uAction = mainUS.CreateLeafNode(factor, action);
```

Salir de un subsistema de utilidad

Hay dos formas de salir de un sistema de utilidad.

- Al acabar de ejecutar un `UtilityAction`, es posible hacer que el sistema de utilidad termine su ejecución con el mismo valor con el que ha terminado la acción. Para ello se pasa el valor `true` al último argumento del método.

```
UtilityAction uAction = us.CreateAction(f, action, true);
```

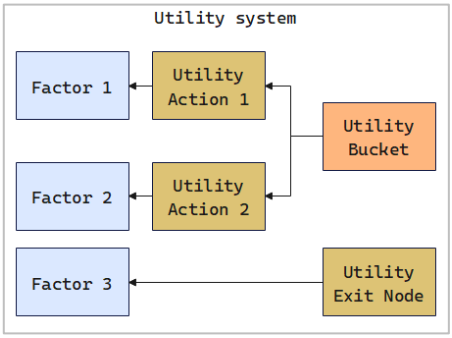
- También se pueden crear nodos que terminen la ejecución del sistema de utilidad cuando son seleccionados. Para crear un nodo de salida se usa el método `CreateExitNode` y se pasa como parámetros el factor para calcular la utilidad y el valor `Status` de salida.

```
UtilityExitNode exitNode = us.CreateExitNode(factor, Status.Success);
```

Extensión: Buckets o grupos de elementos

Los grupos o buckets permiten agrupar acciones, nodos de salida u otros buckets. Para agrupar elementos primero se debe crear el bucket con el método `CreateBucket`. A este método se pasan dos argumentos, la inercia y el umbral de utilidad.

La inercia funciona igual que en los sistemas de utilidad y el umbral determina la utilidad que debe tener una acción del grupo para que este tenga prioridad y se ejecute, aunque otros elementos posteriores de fuera del grupo tengan más utilidad.




```
UtilityBucket bucket = us.CreateBucket(1.3f, 0.3f);
```

Para crear un elemento dentro de un bucket se debe añadir como parámetro el bucket en el método que crea el elemento.

```
UtilityBucket bucket = us.CreateBucket(1.3f, 0.3f);

UtilityAction groupedAction = us.CreateAction(factor, action, false, bucket);
UtilityExitNode groupedExitNode = us.CreateExitNode(factor, Status.Success, bucket);
UtilityBucket subBucket = us.CreateBucket(1.3f, 0.3f, bucket);
```

***NOTA**: El sistema de utilidad escogerá un elemento de aquellos que no tengan grupo asignado. El resto de elementos solo podrán ser escogidos dentro de su propio grupo.* En el ejemplo anterior, el sistema de utilidad escoge entre "*UtilityBucket*" y "*UtilityExitNode*", y si escoge el primero, este escogerá a su vez entre las dos acciones.

Extensión: Optimizar cálculo de utilidad

Es posible optimizar el cálculo de utilidad en los casos es los que sabemos cuándo el valor va a cambiar. Para ello se asigna el valor *false* a la variable *PullingEnabled* para hacer que la utilidad no se recalcule en cada frame.

```
float v = 1f;
UtilitySystem us = new UtilitySystem();
Action action = …
Factor leafFactor = us.CreateLeaf(() => v);
UtilityAction uAction = us.CreateAction(leafFactor, action1);
uAction.PullingEnabled = false;
```

Después solo tenemos que llamar al método UpdateUtility pasando como parámetro el valor *true* cuando queramos que la utilidad se actualice. En el ejemplo anterior, esto sería cada vez que cambia el valor de v.

```
uAction.UpdateUtility(true);
```