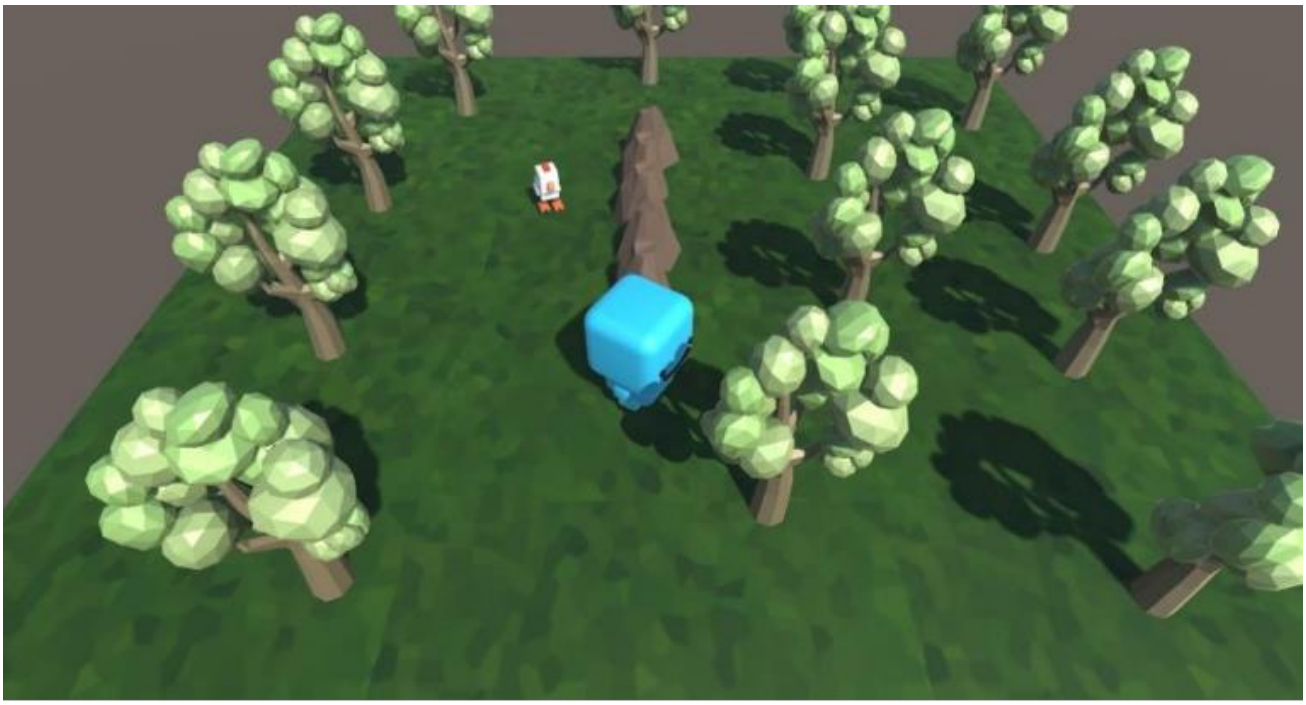
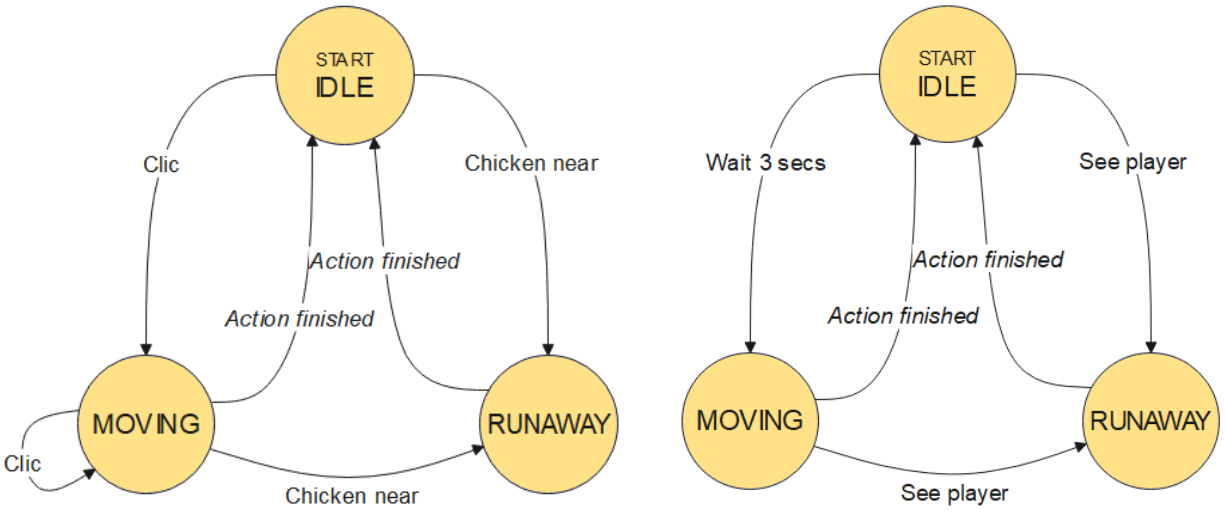
Demos

1. Demo de máquinas de estados: El chico contra la gallina

En esta escena aparece un personaje controlable por el usuario y una gallina en un pequeño escenario. El jugador puede mover el personaje haciendo clic en un punto del escenario siempre que no se acerque demasiado a la gallina, en cuyo caso saldrá corriendo a una dirección aleatoria. La gallina se moverá aleatoriamente por el escenario hasta que el personaje entre en su radio de visión, momento en el que empezará a perseguirle por unos segundos.



1.1 Diagramas de máquinas de estados



1.2 Explicación de la implementación de los personajes

Para este primer ejemplo se va a explicar como funcionan las transiciones entre estados y las distintas formas de crearlas (con percepciones pull, percepciones push, automáticas cuando termina una acción, etc.)

El sistema de comportamiento del personaje del jugador usa una única percepción que comprueba la distancia al objeto de la gallina. Esta percepción se crea con el método "CheckDistanceToChicken", que se ejecuta cada vez que se lanza el método `Check` de la percepción:

```
chickenNear = new ConditionPerception(CheckDistanceToChicken);
```

Las transiciones al estado "Moving" se lanzan con percepciones push. Para que no se comprueben desde el estado origen el valor de StatusFlags debe ser "*None*".

```
var idleToMoving = fsm.CreateTransition("idle to moving", idle, moving, statusFlags:
StatusFlags.None);
var movingToMoving = fsm.CreateTransition("moving to moving", moving, moving, statusFlags:
StatusFlags.None);

_click = new PushPerception(movingToMoving, idleToMoving);
```

Esta percepción push se lanza cuando el jugador hace click en el escenario:

```
protected override void OnUpdate()
{
   if (Input.GetMouseButtonDown(0))
   {
        _click.Fire();
   }
   base.OnUpdate();
}
```

Cuando la acción de moverse o huir termina, se pasa automáticamente al estado Idle. Para hacer esto, las transiciones se crean sin percepción para que se lancen siempre que se comprueben. Por último, para que solo se comprueben cuando la acción haya terminado, sin importar si con éxito o fallo, se pasa el valor *Finished* a el campo `StatusFlags`.

```
fsm.CreateTransition("moving to idle", moving, idle, statusFlags: StatusFlags.Finished);
fsm.CreateTransition("runaway to idle", flee, idle, statusFlags: StatusFlags.Finished);
```

Las transiciones al estado "Runaway" usan la percepción creada al inicio:

```
fsm.CreateTransition("idle to runaway", idle, flee, chickenNear);
fsm.CreateTransition("moving to runaway", moving, flee, chickenNear);
```

En el caso del personaje de la gallina, se usan dos percepciones. La primera de ellas es de tipo UnityTimerPerception y comprueba que el estado origen de la transición que la usa lleva activo 3 segundos. La segunda es de tipo CustomPerception y usa el método "CheckWatchTarget" para comprobar si el jugador está en el radio de visión de la gallina.

```
var chickenNear = new ConditionPerception(CheckWatchTarget);
var timeToStartMoving = new UnityTimePerception(3f);
```

Al igual que en el caso anterior, las transiciones al estado Idle se lanzan cuando las acciones de los estados origen han terminado.

```
fsm.CreateTransition("moving to idle", moving, idle, statusFlags: StatusFlags.Finished);
fsm.CreateTransition("runaway to idle", chasing, idle, statusFlags: StatusFlags.Finished);
```

Para pasar al estado moving se usa la percepción timer:

```
var idleToMoving = fsm.CreateTransition("idle to moving", idle, moving, timeToStartMoving);
```

Por último, para pasar al estado chasing, se usa la percepción custom:

```
fsm.CreateTransition("idle to runaway", idle, chasing, chickenNear);
fsm.CreateTransition("moving to runaway", moving, chasing, chickenNear);
```

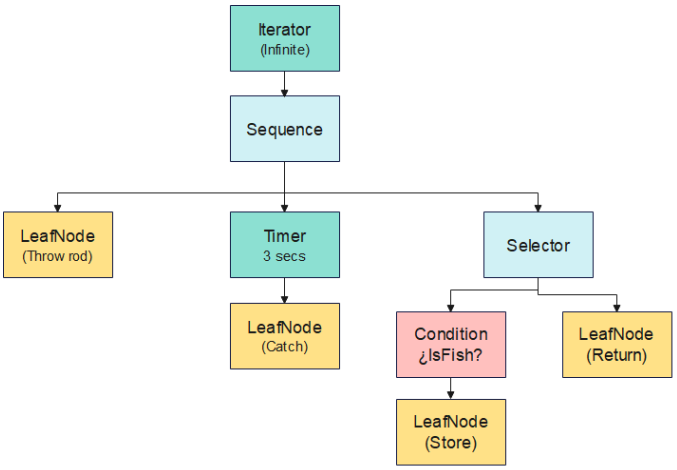
2. Demo de árbol de comportamiento y decoradores: Pesca

En esta escena hay un personaje pescando. El personaje lanzará la caña y cuando pase cierto tiempo y una presa pique tirará del sedal. La captura podrá ser un pez, en cuyo caso se guardará en la cesta de al lado del pescador, o una bota, en cuyo caso se devolverá al agua.

El árbol ejecuta una secuencia de tres pasos de forma infinita usando un decorador de tipo IteratorNode.

Primero ejecuta un nodo hoja que ejecuta una acción custom que consiste en lanzar la caña al agua. Después ejecuta un decorador de tipo UnityTimerDecorator que espera 3 segundos para ejecutar su nodo hoja hijo, con otra acción custom que genera aleatoriamente una bota o un pez en la punta de la caña. El tercer paso es un nodo selector que primero comprueba si la captura es un pez usando un nodo condición con una percepción custom. Si lo es ejecuta el nodo hoja hijo que deja el pez en la cesta. Si no se cumple el nodo condición devolverá false y el selector pasará a su siguiente nodo hijo, que realiza una acción para devolver la captura al agua.

2.1 Diagrama del árbol de comportamiento



2.2 Explicación de la implementación

Este ejemplo se centra en como funcionan las acciones en los nodos hojas y como controlar el flujo de ejecución usando los distintos tipos de decoradores.

Lo primero a destacar es que el para que el personaje ejecute su ciclo de acciones infinitamente, el nodo raíz del árbol es un nodo iterador cuyo valor en Iterations es -1. Este es el valor por defecto.

```
var loop = bt.CreateDecorator<IteratorNode>("loop", seq).SetIterations(-1);
```

El nodo hijo del iterador es un nodo secuencia con tres nodos hijos que representan las subtareas del personaje. El primer valor booleano debe ser false para que la secuencia no sea aleatoria.

```
var seq = bt.CreateComposite<SequencerNode>("seq", false, throwTheRod, timer, sel);
```

El primero hijo es un nodo hoja con una acción que lanza un método "`Start Throw`" al comenzar la ejecución. Este método lanza una corrutina en el objeto *"_rod*". Para comprobar cuando ha terminado esta corrutina, se usa el método "`IsThrown`". El método update de la acción comprueba este método y convierte el valor booleano devuelto en valor de Status, de forma que devolverá Running hasta que la corrutina se complete, cuando devolverá Success y terminará la ejecución.

```
var throwTheRod = bt.CreateLeafNode("Throw rod", new FunctionalAction(StartThrow, () =>
_rod.IsThrown() ? Status.Success : Status.Running));
```

El siguiente nodo hijo es un timer de 3 segundos que cuando termina ejecuta un nodo hoja con una acción. Este nodo hoja es idéntico al anterior, pero cambia la corrutina del objeto "_rod".

```
var catchSomething = bt.CreateLeafNode("Catch something", new FunctionalAction(StartCatch, () =>
_rod.IsPickedUp() ? Status.Success : Status.Running));
var timer = bt.CreateDecorator<UnityTimerDecorator>("Timer", catchSomething).SetTotalTime(3f);
```

El último hijo es un selector. En la primera rama del selector se ejecuta un nodo condición que comprueba si el objeto atrapado es un pescado. Este nodo devuelve Failure directamente sin ejecutar el nodo hijo, si la condición no se cumple, lo que haría que el selector pasara a la siguiente rama. Si la condición se cumple, el nodo condicional ejecutaría su nodo hijo y devolvería el resultado de su ejecución directamente.

Tanto la acción de devolver la captura al agua o guardarla en la cesta devuelve Success directamente en la primera iteracción.

```
var returnToWater = bt.CreateLeafNode("Return to water", new FunctionalAction(
DropCaptureInWater, () => Status.Success));
var storeInBasket = bt.CreateLeafNode("Store in basket", new FunctionalAction(
StoreCaptureInBasket, () => Status.Success));

var sel = bt.CreateComposite<SelectorNode>("sel", false, check, returnToWater);
```

Cuando la secuencia se completa, el nodo secuencia devuelve Success y el nodo iterador reinicia su ejecución.

3. Demo de árbol de comportamiento: Entrar en la casa

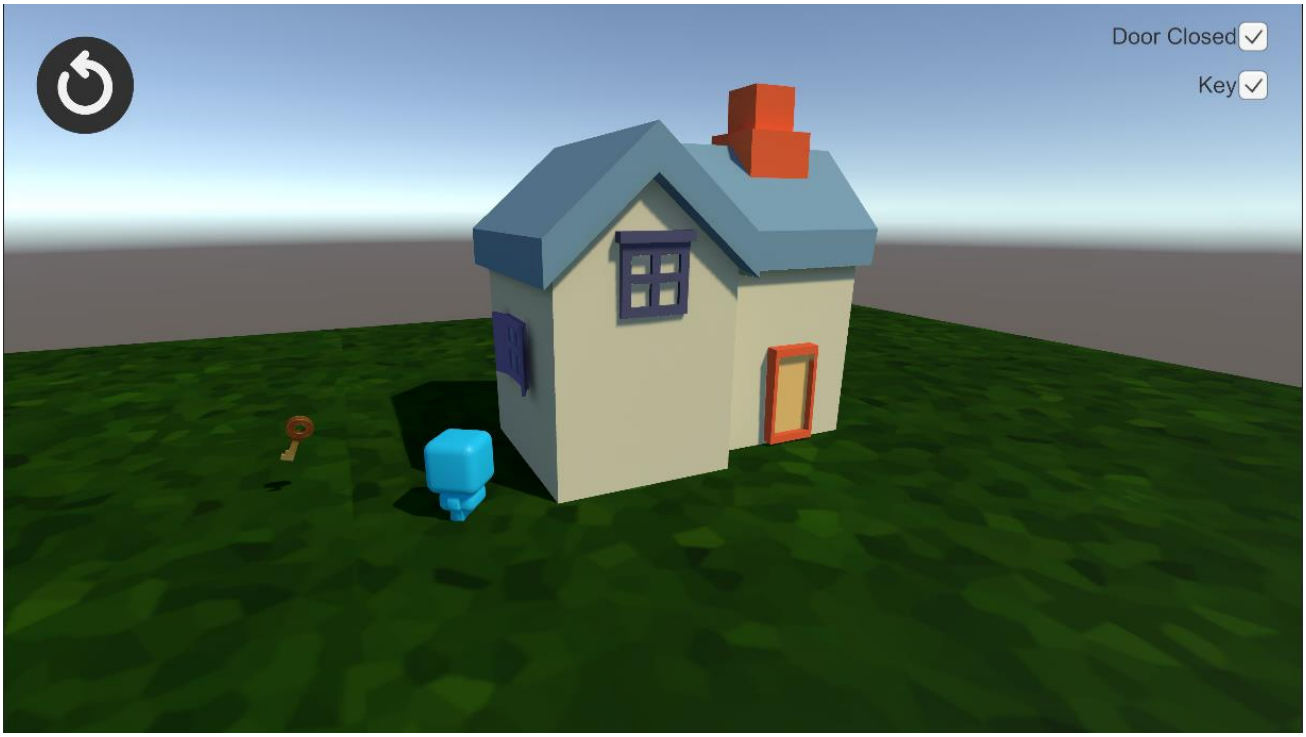
En esta escena de árboles de comportamientos el muñeco azul intentará entrar en su casa. La escena puede configurarse con las opciones de arriba a la derecha para hacer que la puerta este cerrada con llave o no y para activar y desactivar la llave.

Primero, el personaje se acercará a la puerta. Si no está cerrada con llave entrará directamente.

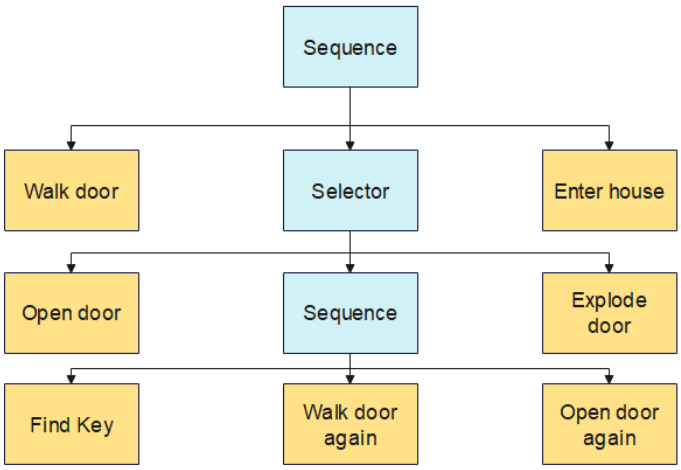
Si la puerta está cerrada y la llave está activa, caminará hasta ella para cogerla y volverá a la puerta para abrirla y entrar en la casa.

Si está cerrada pero no hay llave el personaje explotará la puerta y entrará igualmente.

Para reiniciar la escena se pulsa el botón de la esquina superior izquierda.



3.1 Árbol de comportamiento



3.2 Explicación de la implementación

El nodo raíz del árbol es un nodo secuencia con tres ramas.

```
var root = bt.CreateComposite<SequencerNode>("root", false, walkToDoor, sel, enter);
```

La primera rama es un único nodo hoja que ejecuta una acción de tipo WalkAction, que usa el componente NavMeshAgent del personaje para moverlo hasta la puerta. Este tipo de acciones permite personalizar la posición destino y la velocidad de movimiento.

```
var doorPos = new Vector3(_door.transform.position.x, transform.position.y,
_door.transform.position.z);
var walkToDoorAction = new WalkAction(doorPos, 5f);
var walkToDoor = bt.CreateLeafNode("walkToDoor", walkToDoorAction);
```

La segunda rama es un selector con 3 subramas. La ejecución de estas ramas dependerá de como se haya configurado la escena.

```
var sel = bt.CreateComposite<SelectorNode>("sel", false, openDoor, seq, explode);
```

La primera subrama es un nodo hoja que ejecuta el método OpenDoor y después devuelve directamente Success o Failure en función de si la puerta está abierta o no. Se puede usar el método de extensión `ToStatus` para convertir un booleano en Status, de forma que true equivale a *Success* y false a *Failure*.

```
var openDoorAction = new FunctionalAction(OpenDoor, () => (!_door.IsClosed).ToStatus());
var openDoor = bt.CreateLeafNode("open door", openDoorAction);
```

La siguiente subrama es otra secuencia de tres nodos. El primero es un nodo hoja que ejecuta una acción personalizada, que comprueba si la llave está en el mapa y si es el caso mueve el personaje hasta ella. El segundo y tercer nodo reutilizan las acciones de caminar a la puerta y abrir la puerta.

```
var findKeyAction = new FunctionalAction(FindKey, IsKeyObtained);
var findKey = bt.CreateLeafNode("find key", findKeyAction);
var returnToDoor = bt.CreateLeafNode("use key", walkToDoorAction);
var useKey = bt.CreateLeafNode("try unlock", openDoorAction);
var seq = bt.CreateComposite<SequencerNode>("key seq", false, findKey, returnToDoor, useKey);
```

La ultima subrama es un nodo hoja que ejecuta una acción para explotar la puerta.

```
var explodeAction = new FunctionalAction(SmashDoor, () => Status.Success);
var explore = bt.CreateLeafNode("explode", explodeAction);
```

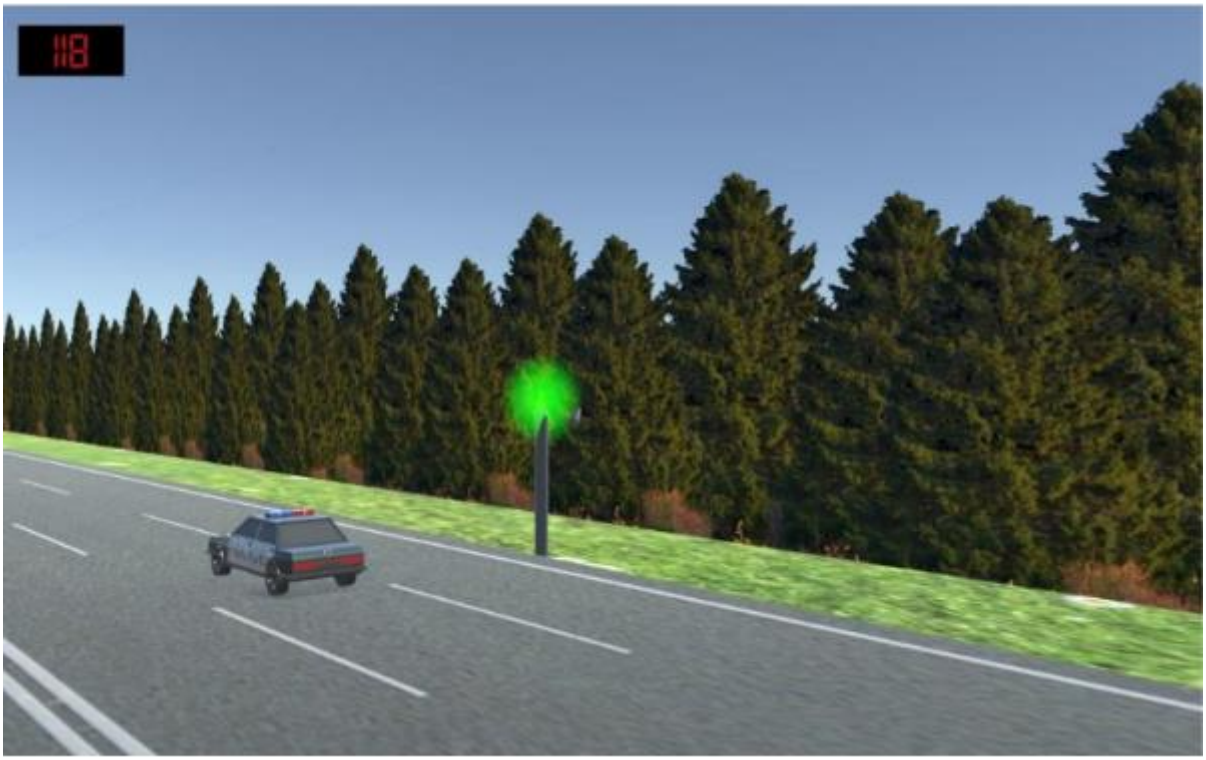
Por último, la última rama del nodo secuencia raíz es un nodo hoja que hace al personaje entrar en la casa y destruirse.

```
var enterAction = new FunctionalAction(EnterTheHouse, () => Status.Success);
var enter = bt.CreateLeafNode("enter", enterAction);
```

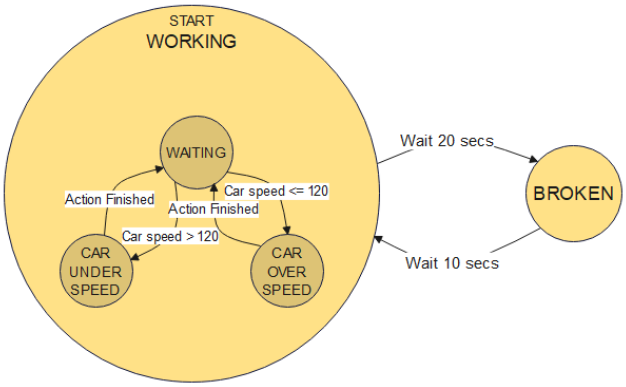
4. Demo FSM jerárquica: Radar

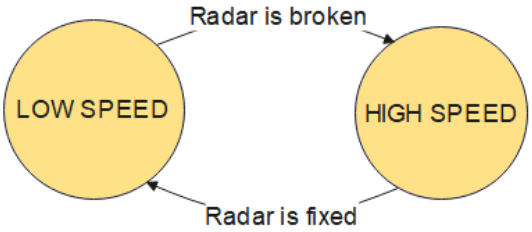
En esta escena vemos un radar en una autopista por la que pasan coches a distintas velocidades. El radar detectará la velocidad de los coches cuando pasen delante de él. Si la velocidad es menor a 120 la luz del radar se pondrá de color verde. Si la supera será de color rojo. En ambos casos se mostrará la velocidad en el panel de la esquina superior derecha.

Cada cierto tiempo el radar se estropeará, poniendo su luz en amarillo intermitente. Cuando el radar esté roto los coches irán más rápido.



4.1 Diagramas de máquinas de estados





4.2 Explicación de la implementación

En el caso del sistema del radar tenemos una máquina de estados que contiene otra submáquina en uno de sus estados.

La FSM principal funciona alternando sus estados periódicamente, usando percepciones de tipo UnityTimerPerception.

```
var fix = new UnityTimePerception(10f);
var @break = new UnityTimePerception(20f);
var @fixed = radarFSM.CreateTransition("fixed", brokenState, workingState, fix);
var broken = radarFSM.CreateTransition("broken", workingState, brokenState, @break);
```

La acción del estado “Working” ejecuta la submáquina mientras que la del estado “Broken” es una acción de tipo BlinkAction, que hace parpadear una luz en un color concreto.

```
var workingState = radarFSM.CreateState("working state", new SubsystemAction(subFSM));
var brokenState = radarFSM.CreateState("broken state", new BlinkAction(radarLight, speedText,
Color.yellow));
```

La submáquina de estados tiene tres estados. Comenzando en el estado “Waiting”, cuando un coche pasa frente al radar lanzará una de las dos transiciones a los otros dos estados, dependiendo de la velocidad del coche.

```
var overSpeedPerception = new ConditionPerception(() => CheckRadar((speed) => speed > 20));
var underSpeedPerception = new ConditionPerception(() => CheckRadar((speed) => speed <= 20));
lightSubFSM.CreateTransition("car over speed", waitingState, overSpeedState,
overSpeedPerception);
lightSubFSM.CreateTransition("car under speed", waitingState, underSpeedState,
underSpeedPerception);
```

Las acciones de estos dos estados son ambas de tipo `LightAction`, que cambian el color de una luz por un tiempo determinado.

Cuando estas acciones terminan, se vuelve al estado “*Waiting*”, usando las flags de las transiciones:

```
lightSubFSM.CreateTransition("over speed to waiting", overSpeedState, waitingState, statusFlags:
StatusFlags.Finished);
lightSubFSM.CreateTransition("under speed to waiting", underSpeedState, waitingState,
statusFlags: StatusFlags.Finished);
```

Por su lado, los coches que se van generando tienen una máquina de estados con dos estados.

Esta máquina usa percepciones de tipo ExecutionStatusPerception para saber el estado del radar. En la clase del radar se guardan los dos estados de la máquina principal en variables y cuando los coches son creados, acceden a estas variables para crear las percepciones.

```
public class RadarFSMRunner : CodeBehaviourRunner, IRadar
{
    State _brokenState, _workingState;
    …
}
```

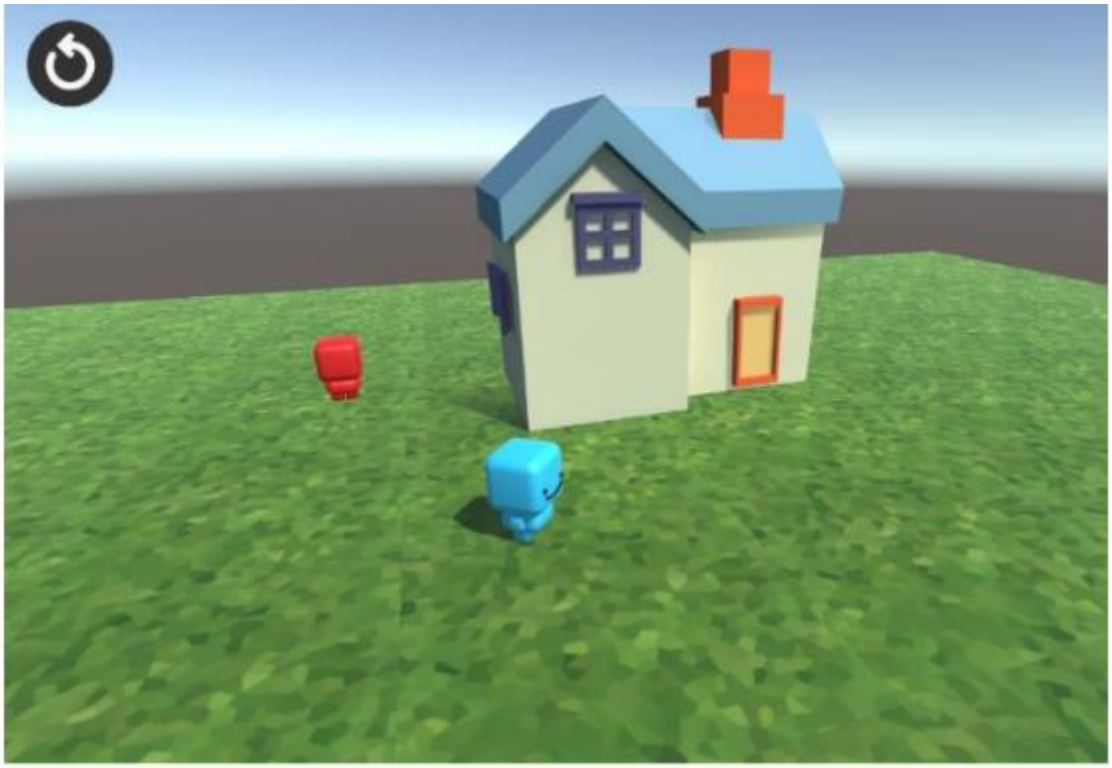
Si no se especifica el parámetro StatusFlags, las percepciones comprobarán si el elemento está en estado “Running”.

```
var radarIsBroken = new ExecutionStatusPerception(_radar.GetBrokenState());
var radarIsWorking = new ExecutionStatusPerception(_radar.GetWorkingState());
```

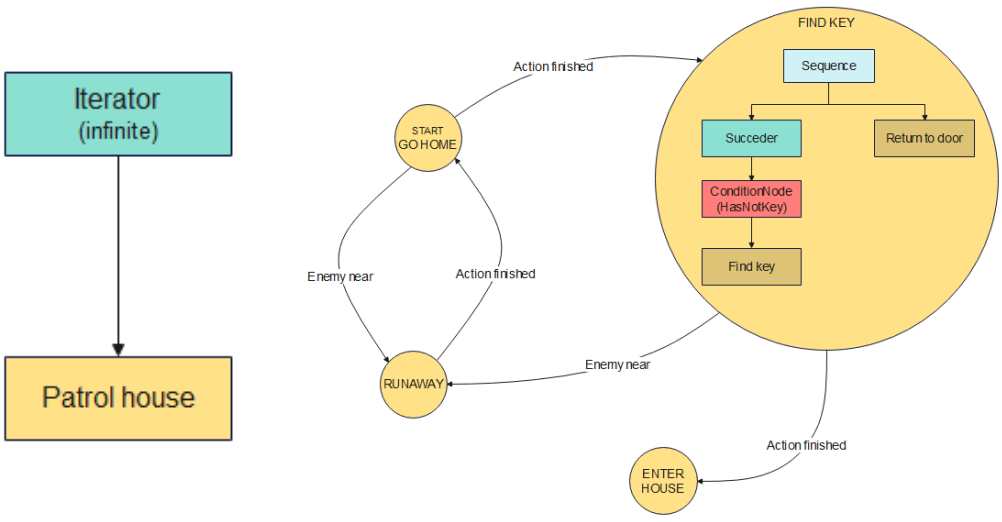
5. Demo árbol de comportamiento en FSM: Casa con enemigo

En esta escena hay un personaje azul y uno rojo. El personaje rojo da vueltas alrededor de la casa de forma infinita, mientras que el personaje azul intentará entrar. Para ello se dirigirá hacia la puerta y cuando descubra que está cerrada irá a por la llave. Cuando la tenga volverá a la puerta y usará la llave para entrar.

Si en algún momento el personaje rojo se acerca demasiado al azul, este empezará a correr sin control hasta que se aleje. Cuando lo haya hecho reanudará su rutina.



5.1 Diagramas de sistemas de comportamiento



5.2 Explicación de la implementación

El sistema de comportamiento del enemigo es simplemente un nodo iterador que ejecuta infinitamente un nodo hoja con una acción de tipo `PatrolAction`. Esta acción usa el transform del objeto para moverlo por una serie de puntos a una velocidad determinada.

```
var patrol = new PathingAction(routePoints.Select(tf => tf.position).ToList(), 3f, .1f);
var leaf = bt.CreateLeafNode(patrol);
var root = bt.CreateDecorator<IteratorNode>(leaf);
```

El sistema del personaje consiste en una FSM con cuatro estados en la que uno de estos estados contiene un árbol de comportamiento.

La FSM comienza en el estado “Enter House”. La acción de este estado mueve al personaje hasta la puerta.

```
_doorPos = new Vector3(_door.position.x, transform.position.y, _door.position.z);
var doorState = fsm.CreateState("Go to home", new WalkAction(_doorPos, 5f));
```

Al terminar la acción anterior se pasa al estado “FindKey” que contiene el subárbol. Este árbol de comportamiento está formado por un nodo secuencia con dos ramas. La estructura de Succeder + Condicional + Nodo hoja de la primera rama sirve para que solo se ejecute la acción de buscar la llave si todavía no se ha cogido, pero independientemente del caso la rama siempre devuelva Success para que la secuencia pase a la siguiente acción.

```
var l1 = bt.CreateLeafNode(new WalkAction(keyPos, 5f));
var hasKey = bt.CreateDecorator<ConditionNode>(l1).SetPerception(new ConditionPerception(() =>
!_hasKey));
var succeder = bt.CreateDecorator<SuccederNode>(hasKey);
var l2 = bt.CreateLeafNode("Return to door", new WalkAction(_doorPos, 5f));
var seqRoot = bt.CreateComposite<SequencerNode>(false, succeder, l2);
```

Cuando termina la ejecución del subarbol se pasa al estado “Enter house”. La acción de este estado hace al personaje entrar en la casa y destruirse.

```
var houseState = fsm.CreateState("Enter the house", new FunctionalAction(EnterTheHouse));
```

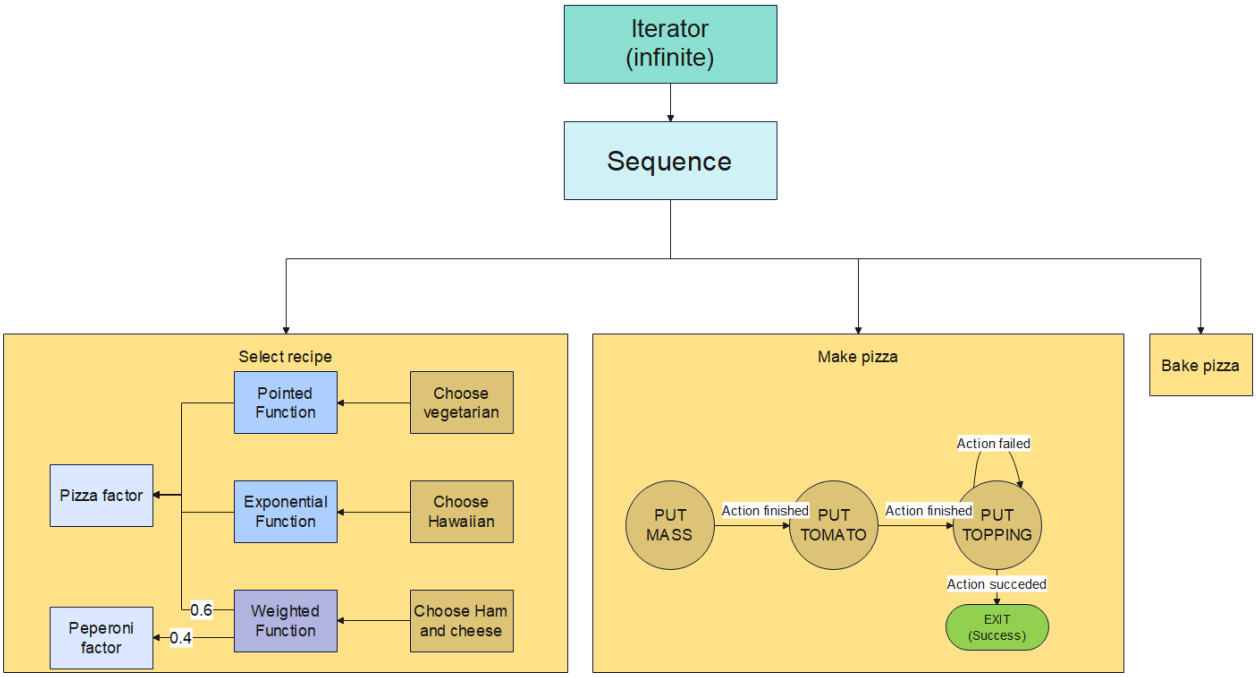
En cualquiera de los dos primeros estados, si el enemigo está suficientemente cerca se pasa al estado “Runaway”. La acción de este estado es de tipo FleeAction, y hace que el personaje corra en direcciones aleatorias. Cuando la acción termine, se volverá al estado “Enter house”.

```
var runState = fsm.CreateState("Runaway", new FleeAction(8f, 10f, 3f));
fsm.CreateTransition("Interrupt key", key State, runState, enemyNearPerception);
fsm.CreateTransition("Interrupt going to door", doorState, runState, enemyNearPerception);
```

6. Demo árbol de comportamiento + FSM + Sistema de utilidad: Pizzero

En esta escena se encuentra un pizzero que va atendiendo pedidos de forma periódica. Primero selecciona una de las recetas de las tres disponibles. Después va colocando los ingredientes uno a uno y cuando ha completado la receta lleva la pizza al horno.

6.1 Diagramas de sistemas de comportamiento



6.2 Explicación de la implementación

El sistema de comportamiento del pizzero esta formado por un árbol de comportamiento que consiste en un nodo iterador que ejecuta infinitamente una secuencia de tres nodos hoja.

El primer nodo hoja contiene una acción con un subsistema de utilidad que sirve para decidir cual de las tres recetas de pizza se va a hacer.

Este sistema de utilidad tiene dos factores variables. El primero cuenta la cantidad de pizzas que se han hecho y calcula la utilidad como $(n\_pizzas \ \% \ 10) \ / \ 10$. El segundo factor cuenta la cantidad de pepperoni que se ha usado (en las pizzas de jamón y queso), calculado la utilidad con $(n\_peperoni \ \% \ 4) \ / \ 4$.

```
var pizzafactor = us.CreateVariableFactor("pizzas", () => _pizzasCreated % 10, 10, 0);
var pepperoniFactor = us.CreateVariableFactor("peperoni_used", () => _peperoniUsed % 4, 4, 0);
```

A continuación, se usan dos factores función y un factor fusión. El primer factor función calcula su utilidad con una función a puntos y el segundo con una función exponencial. El tercero hace la media ponderada de los dos factores variables.

```
var peperoniSumFactor = us.CreateFusionFactor<WeightedFusionFactor>("peperoni", pizzafactor,
pepperoniFactor).SetWeights(0.6f, 0.4f);

var pointList = new List<BehaviourAPI.Core.Vector2>();
pointList.Add(new BehaviourAPI.Core.Vector2(0.0f, 1f));
pointList.Add(new BehaviourAPI.Core.Vector2(0.2f, 0.5f));
pointList.Add(new BehaviourAPI.Core.Vector2(0.4f, 0.1f));
pointList.Add(new BehaviourAPI.Core.Vector2(0.6f, 0.4f));
pointList.Add(new BehaviourAPI.Core.Vector2(0.8f, 0.2f));
pointList.Add(new BehaviourAPI.Core.Vector2(1.0f, 0.0f));
var vegetarianFactor = us.CreateFunctionFactor<PointedFunction>("vegetarian",
pizzafactor).SetPoints(pointList);
var hawaiianFactor = us.CreateFunctionFactor<ExponentialFunction>("hawaiian",
pizzafactor).SetExponent(.7f);
```

Cada uno de estos tres factores sirve para calcular la utilidad de una de las acciones de utilidad. Cada una de las acciones sirve para escribir la receta seleccionada en la libreta.

Al crear las acciones se pone el parámetro `finishOnComplete` a *true* para que la ejecución del sistema de utilidad termine cuando la acción termine y continuar la secuencia del árbol de comportamiento principal.

```
var peperoniAction = us.CreateAction("choose ham and cheese", peperoniSumFactor, new
FunctionalAction(() => CreateRecipe(0), RecipeCreated, CreateRecipeCompleted), finishOnComplete:
true);
var vegetarianAction = us.CreateAction("choose vegetarian", vegetarianFactor, new
FunctionalAction(() => CreateRecipe(1), RecipeCreated, CreateRecipeCompleted), finishOnComplete:
true);
var hawaiianAction = us.CreateAction("choose hawaiian", hawaiianFactor, new FunctionalAction(()
=> CreateRecipe(2), RecipeCreated, CreateRecipeCompleted), finishOnComplete: true);
```

El segundo nodo contiene una submáquina de estados con tres estados. El primer estado sirve para crear la masa de la pizza y el segundo para crear la base de tomate.

```
var massState = fsm.CreateState("mass", new FunctionalAction(() => PutIngredient(_pizzaMass),
WaitToPutIngredient));
var tomatoState = fsm.CreateState("tomato", new FunctionalAction(() => PutIngredient(_tomato),
WaitToPutIngredient));
fsm.CreateTransition("mass putted", massState, tomatoState, statusFlags: StatusFlags.Finished);
fsm.CreateTransition("tomato putted", tomatoState, toppingState, statusFlags:
StatusFlags.Finished);
```

La acción del tercer estado lee de la lista de ingredientes de la receta seleccionada y añade el siguiente ingrediente a la pizza. Si no es el último ingrediente la acción termina con Failure y se lanza la transición cíclica que reinicia el estado. Si el ingrediente era el último se lanza la transición de salida para continuar la ejecución de la secuencia.

```
var toppingState = fsm.CreateState("topping", new FunctionalAction(PutNextTopping, CheckToppings));
fsm.CreateTransition("next topping", toppingState, toppingState, statusFlags:
StatusFlags.Failure);
fsm.CreateExitTransition("pizza completed", toppingState, Status.Success, statusFlags:
StatusFlags.Success);
```

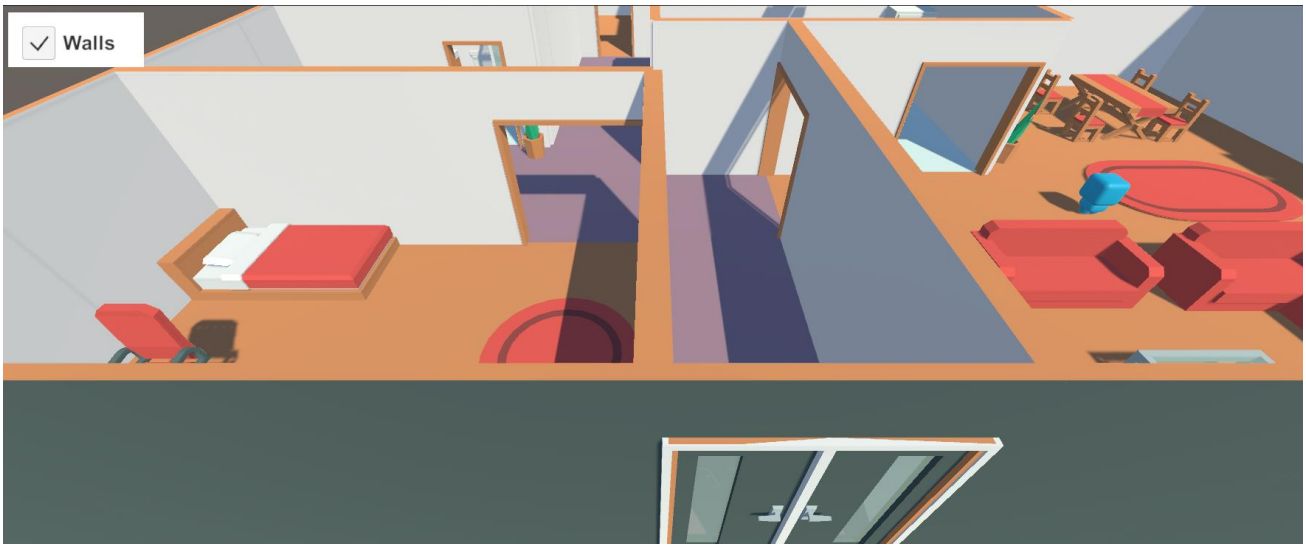
Por ultimo se ejecuta el ultimo nodo hoja de la secuencia.

```
var bakeAction = bt.CreateLeafNode("bake pizza", new FunctionalAction(BakePizza, PizzaBaked,
BakedActionCompleted));
```

7. Demo Smart objects: Sims

El escenario consiste en una casa con varias habitaciones entre las que se mueven varios personajes. Hay dos tipos de personajes, el primero selecciona objetos aleatoriamente de forma infinita y el segundo usa un sistema de utilidad para usar el *Smart Object* que más le convenga según sus necesidades en cada momento.

Los agentes se basan en seis necesidades: hambre, sed, vejiga, higiene, descanso y ocio. Cada uno de los objetos puede cubrir una de estas necesidades que disminuyen periódicamente.

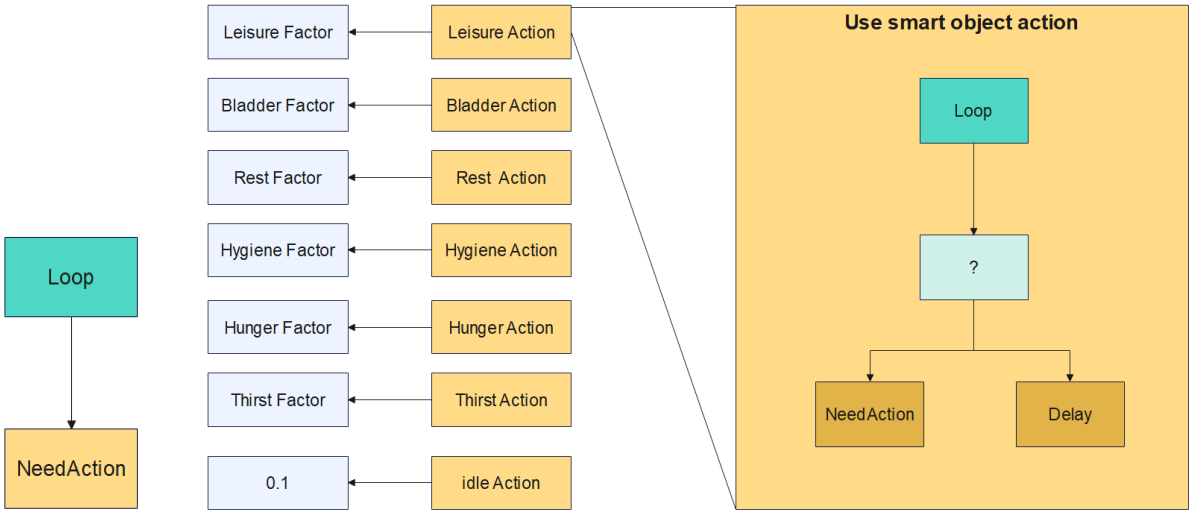


7.1 Diagramas de sistemas de comportamiento

El sistema de comportamiento del primer tipo de personajes es un árbol de comportamiento con un bucle infinito que ejecuta una acción de tipo *RandomRequestAction*. Esta acción selecciona un *SmartObject* aleatorio de la escena y lo usa.

El segundo sistema se basa en un sistema de utilidad en el que se relaciona cada necesidad del agente a una acción que consiste en usar un objeto que cubre dicha necesidad. Si ningun valor de necesidad llega al valor mínimo (0.1) el personaje simplemente caminará aleatoriamente por el escenario.

Cada una de las acciones consiste en un subarbol de comportamiento ejecutado en bucle, con un selector con dos nodos hojas. Mientras la acción ligada a la necesidad “x” se buscará un objeto que cubra esa necesidad y si se encuentra se ejecutará la interacción que proporcione. Si despues de terminar la necesidad sigue siendo prioritaria se repetirá el proceso, y si no se encuentra ningun objeto para cubrirla se saltará al otro nodo del selector que consiste en un delay de 5 segundos.



7.2 Explicación de la implementación de los agentes

El sistema de comportamiento del primer tipo de personajes se ha implementado es un árbol de comportamiento con un bucle infinito que ejecuta una acción de tipo *RandomRequestAction*. Esta acción selecciona un *SmartObject* aleatorio de la escena y lo usa.

```
BehaviourTree bt = new BehaviourTree();
RandomRequestAction randomRequestAction = new RandomRequestAction(_agent);
LeafNode leaf = bt.CreateLeafNode(randomRequestAction);
LoopNode root = bt.CreateDecorator<LoopNode>(leaf);
bt.SetRootNode(root);
```

El segundo sistema se basa en un sistema de utilidad en el que se relaciona cada necesidad del agente a una acción que consiste en usar un objeto que cubre dicha necesidad. Si ningun valor de necesidad llega al valor mínimo (0.1) el personaje simplemente caminará aleatoriamente por el escenario.

```
UtilitySystem us = new UtilitySystem(1.2f);

Factor leisureFactor = us.CreateVariable(() => m_Agent.GetNeed("leisure"), 1, 0);
Factor restFactor = us.CreateVariable(() => m_Agent.GetNeed("rest"), 1, 0);
Factor hygieneFactor = us.CreateVariable(() => m_Agent.GetNeed("hygiene"), 1, 0);
Factor bladderFactor = us.CreateVariable(() => m_Agent.GetNeed("bladder"), 1, 0);
Factor hungerFactor = us.CreateVariable(() => m_Agent.GetNeed("hunger"), 1, 0);
Factor thirstFactor = us.CreateVariable(() => m_Agent.GetNeed("thirst"), 1, 0);

Factor defaultFactor = us.CreateConstant(0.1f);

us.CreateAction("leisure action", leisureFactor, CreateSmartObjectAction("leisure"));
us.CreateAction("rest action", restFactor, CreateSmartObjectAction("rest"));
us.CreateAction("hygiene action", hygieneFactor, CreateSmartObjectAction("hygiene"));
us.CreateAction("bladder action", bladderFactor, CreateSmartObjectAction("bladder"));
us.CreateAction("hunger action", hungerFactor, CreateSmartObjectAction("hunger"));
us.CreateAction("thirst action", thirstFactor, CreateSmartObjectAction("thirst"));

us.CreateAction("default action", defaultFactor, CreateDefaultAction());
```

Cada una de las acciones consiste en un subarbol de comportamiento ejecutado en bucle, con un selector con dos nodos hojas. Mientras la acción ligada a la necesidad “x” se buscará un objeto que cubra esa necesidad y si se encuentra se ejecutará la interacción que proporcione. Si despues de terminar la necesidad sigue siendo prioritaria se repetirá el proceso, y si no se encuentra ningun objeto para cubrirla se saltará al otro nodo del selector que consiste en un delay de 5 segundos.

```
var bt = new BehaviourTree();
bt.SetRootNode(
       bt.CreateDecorator<LoopNode>("loop",
           bt.CreateComposite<SelectorNode>("sel",
                false,
                bt.CreateLeafNode("request", new NeedRequestAction(needName)),
                bt.CreateLeafNode("delay", new DelayAction(5f))
            )
        )
    );
```

7.3 Explicación de la implementación de los Smart objects

Dentro de la escena hay varios tipos de *SmartObjects*. La mayoría de ellos se usan moviendo el agente hasta el objeto y ejecutando una acción durante un tiempo definido y al acabar se aplica la capacidad del objeto. Estos son la cama, la ducha, la pila del baño, el retrete, las sillas y la nevera. Para cada uno de ellos hay una acción especifica cuando el agente se coloca frente al objeto, como la ducha que activa un sistema de partículas, la nevera que abre la puerta o la cama que coloca el personaje tumbado en ella.

El resto de los objetos usan otros objetos para su interacción. Al usar el ordenador, el agente se sienta en la silla que se encuentra frente a él, y al usar una librería o el televisor se buscará una silla cercana para sentarse.

Por otro lado, objetos como la nevera no se usan directamente, sino que sirven para crear la interacción de otros objetos. La nevera contiene varios objetos dentro (manzana, pollo, refresco…) y son estos objetos los que se usan.

Para implementar el primer grupo de acciones se ha creado una clase común de *SmartObject* llamada *DirectSmartObject* de la que heredan el resto. La interacción que crea el objeto consiste en una secuencia de dos acciones, siendo la primera una acción de movimiento del agente hasta el objeto y la segunda se define en cada subclase mediante el método *GetUseAction*.

```
// Forma simple de hacer una secuencia de acciones sin usar un BehaviourTree
SequenceAction sequence = new SequenceAction();

// Primer paso: mover al agente hasta el objeto
sequence.SubActions.Add(new WalkAction(_placeTarget.position));

// Segundo paso: usar el objeto
sequence.SubActions.Add(GetUseAction(agent, requestData));
```

Para la interacción del *SmartObject* del ordenador se usa una acción de tipo *TargetRequestAction* pasando como parámetro el objeto de la silla.

```
Action action = new TargetRequestAction(agent, seat, requestData);
```

Para el objeto de la estantería de libros se ha creado un tipo de *RequestAction* llamado *SeatRequestAction* que busca el asiento libre más cercano. Para gestionar todos los asientos de la escena se incluye en script *SeatManager*. El objeto del televisor funciona de forma muy parecida, pero utiliza *TVSeatRequestAction*, que limita la distancia a la que puede estar el asiento.

```
Action seatAction = new SeatRequestAction(agent);
Action action = new TVSeatRequestAction(agent, transform, maxDistance, useTime);
```

Por último, el objeto *ChickenSmartObject* encadena el uso de varios objetos dentro de su interacción: primero mueve el personaje hasta la nevera para coger el pollo, después lo mueve hasta el horno para cocinarlo y después lo sienta para que se lo coma. Para implementar esto se ha usado una secuencia de *RequestActions* a cada uno de los objetos implicados.

```
SequenceAction sequence = new SequenceAction();
sequence.SubActions.Add(new TargetRequestAction(agent, _fridge, requestData));
sequence.SubActions.Add(new TargetRequestAction(agent, _oven, requestData));
sequence.SubActions.Add(new SeatRequestAction(agent));
```