Objetivos del tutorial

Los objetivos de este tutorial se pueden resumir en los siguientes puntos:

- Conocer las máquinas de estados finitos
- Experimentar con ellas para crear conjuntos de agentes.
- Conocer un poco los Goal Oriented Action Planed (GOAP)

Estos puntos los vamos a cubrir haciendo uso del motor Unity de videojuegos

¿Qué es Unity?

Existe multitud de posibilidades cuando hablamos sobre motores de videojuegos, si bien existen dos que destacan sobre el resto debido a su adopción por parte de los diferentes estudios de desarrollo.

Uno de esos motores mayoritarios es Unreal, ya es de sobras conocido por los alumnos del curso, pero a algunos puede que nos les suene el que probablemente sea su competidor más directo, Unity. Este motor está principalmente programado en C# y Javascript sobre el framework de .NET.

La facilidad en su aproximación y aprendizaje, junto con esa posibilidad que el mismo juego sea exportado a Android, iOS, Xbox, Windows, Linux … han hecho que muchos estudios, sobre todo independientes, hayan optado por Unity para sus desarrollos. A esto se le asocia el Asset Store que permite no solo vender los juegos si no desarrollos intermedios lo que hace que las posibilidades de negocio para estos estudios pequeños no se limiten a productos finales estableciendo nuevas líneas de ingreso general, que puede aliviarla su situación económico sobre todo en estados iniciales. Así mismo, si bien el motor no es gratuito para el uso comercial, la compañía solo cobra si la facturación supera los 100.000 dólares de los que ellos solicitan un pequeño porcentaje.

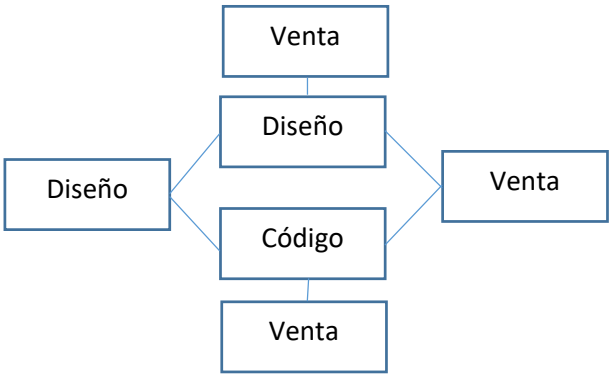


Figura 1 Esquema de negocio de Unity con la venta de desarrollos intermedios

El porqué de Unity para las prácticas

Por lo comentado de Unity, este tiene dos características principales, por un lado, es multiplataforma es decir el mismo desarrollo se puede exportar de manera sencilla sin reprogramar hasta a 48 plataformas diferentes. Por otro lado, el motor da la posibilidad de desarrollar en dos lenguajes de alto nivel que hacen provocan que la curva de aprendizaje tenga una pendiente mucho menor que cualquier otro entorno que use un lenguaje de bajo nivel como C++. Si bien esto tiene un coste asociado en el rendimiento, a la hora de una primera

aproximación el lenguaje C# hace que sea mucho más accesible en contraposición a Unreal, siempre que este no le da soporte.

No es objeto de esta asignatura el explicar todas las capacidades con que cuenta Unity, si así lo desea, el profesorado anima a los estudiantes a completar su formación profundizando en este segundo motor de videojuegos. Aun así, y debido a la facilidad de entender el código asociado respecto de otros lenguajes de bajo nivel como C++, nos vamos a apoyar en Unity para explicar 3 aspectos clave que de otra manera sería muy complicado cubrir en un entorno muy potente pero complejo como es Unreal, especialmente para los no iniciados en la programación en base a código.

Interface de Unity

A mayores de lo dicho anteriormente, Unity provee de un entorno de desarrollo como el que se puede ver en la Figura 2. En dicha figura, se puede apreciar la vista general de Unity donde se pueden diferenciar 4 partes muy claramente. En primer lugar, tendríamos a la izquierda la jerarquía de los objetos donde encontramos todos los elementos que tenemos en la escena. A la izquierda está el inspector, el cual permite ver y modificar las propiedades de los elementos, así como añadir nuevo comportamiento. Finalmente, la parte central se divide en dos partes, la superior, en donde encontramos la escena que estamos creando y donde se puede hacer una prueba del juego desarrollado al más puro estilo de los programas de modela como Blender, Maya o 3DStudio. En la parte inferior, ternemos la parte de recursos, en esta parte encontraremos los ficheros asociados al proyecto que reciben el nombre genérico de Assets, pudiendo ser esto desde scripts, sonidos, modelos de personajes, etc.

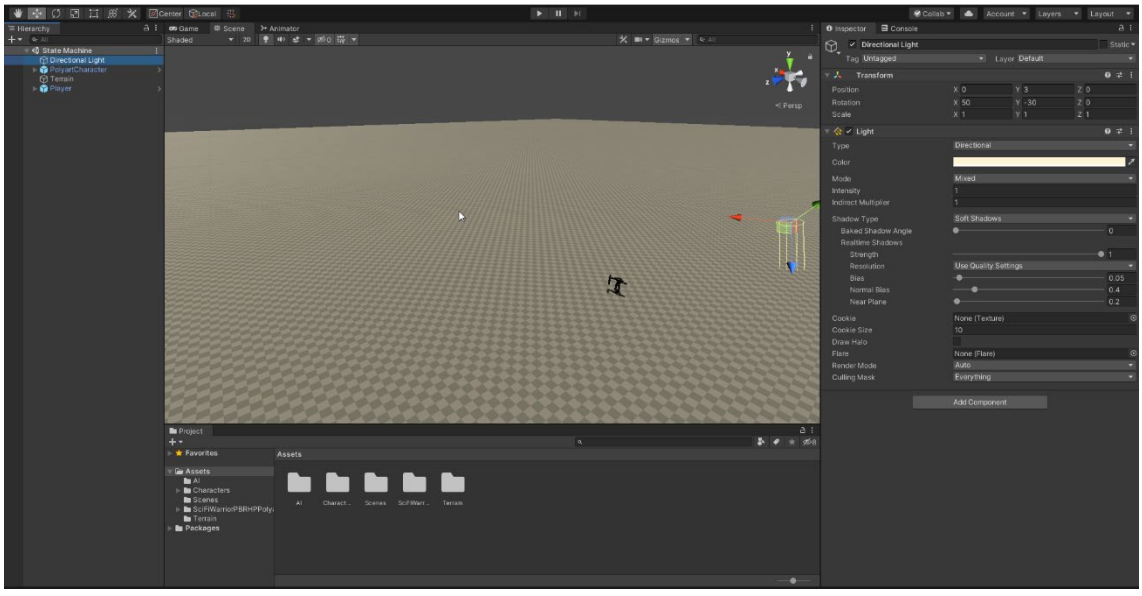


Figura 2. Interface Unity

En líneas generales lo que nos provee Unity es de una interface muy intuitiva donde tenemos integrado casi todo lo que necesitamos para el desarrollo de nuestro juego. Dicho esto, en este caso vamos a hacer un uso intensivo de una de las propiedades más salientables de Unity que es el uso de los Prefabs. Estos son plantillas de objetos, materiales, sonidos, etc. Que se puede definir previamente para después importar dentro del juego.

La Inteligencia Artificial en los Videojuegos

Si bien existen más, en este tutorial nos vamos a centrar en 2 aspectos concretos que vamos a desarrollar: máquinas de estado y Goal Oriented Action Planning.

Primer Día de Prácticas

Máquinas de Estado

Una máquina de estados no es otra cosa que una función f que dado un conjunto de estados *S* permite transicionar entre ellos en función de un estado *w* de las variables del entorno. De tal manera que, trasladando esta definición al entorno matemático tendríamos una expresión similar a:

$$f(s,w) \rightarrow s' \; \forall s, s' \in S, \exists w \in W$$

De esta manera se define un grafo de relaciones de donde los diferentes estados s de S son los nodos y donde las aristas están definidas en función de una serie de condiciones sobre el mundo W. Este grafo, tal como se vio en teoría, puede definirse de diferentes maneras en este caso lo vamos a hacer de la manera más sencilla posible que es con *hard-coding*. Para esto vamos a utilizar un proyecto de Unity y definiremos un guardia Robot que persiga al jugador y le dispare cuando este entre dentro del campo de visión.

Dentro de Unity Hub, cree un nuevo proyecto del tipo 3D. Dele un nombre y ubicación que más le convenga dentro de la máquina.

El siguiente paso, una vez arrancado el proyecto, será ir al menú Assets>Import Package>Custom Package (Figura 3). Seleccione el paquete FSM que está enlazado a esta lección en Moodle.

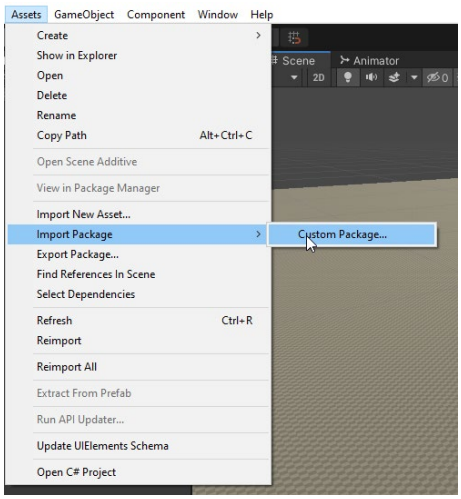


Figura 3 Importar lo Assets de la práctica

Estos son elementos que vamos a utilizar durante este primer ejercicio. Una vez importados todos los elementos, bastará con sustituir la escena por la que se encuentra en la carpeta Scenes, simplemente arrastrando a la jerarquía de la izquierda el assest y eliminando la escena de ejemplo. El resultado debiera de ser como el que se muestra en la Figura 4.

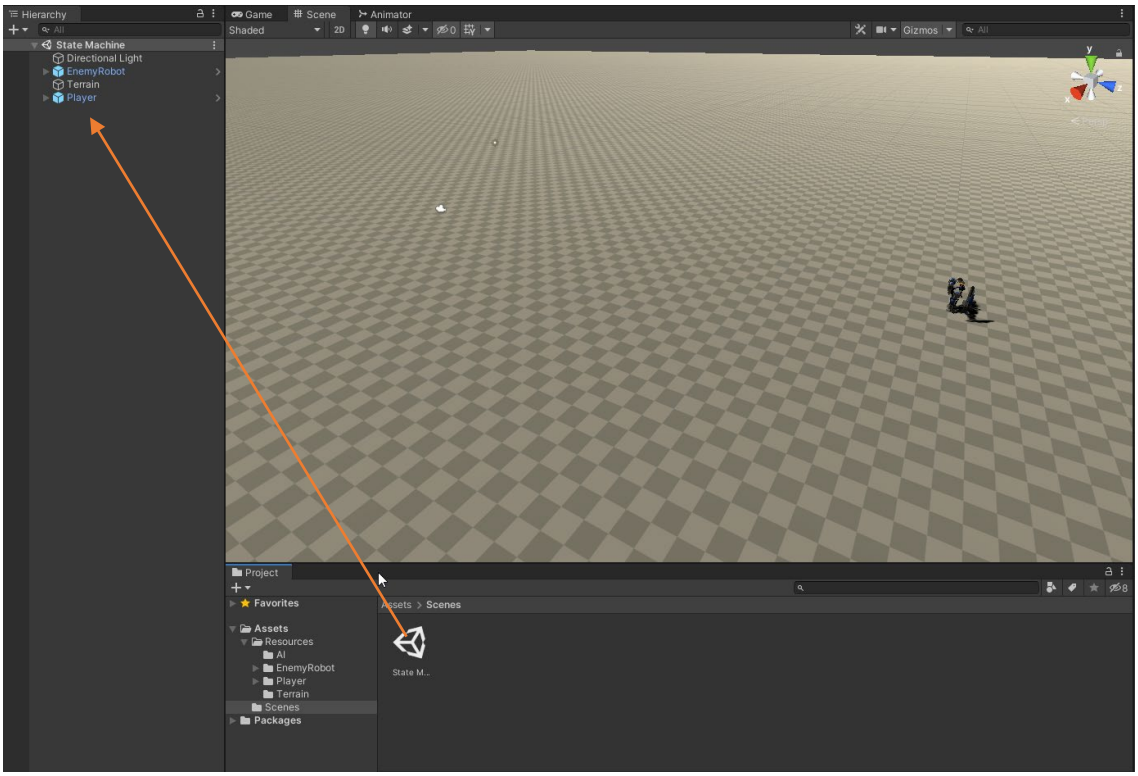


Figura 4 Importar la escena principal

Como podemos ver la escena es muy sencilla y sólo cuenta con un prefab Player que lleva asociado un First Person Controller para poder visualizar las pruebas que realicemos. Con ello si le damos a ejecutar el juego con el botón play, que se encuentra en la parte superior central podremos controlar a nuestro Player con las teclas WASD y el ratón. A mayores de eso hay un terreno y un Personaje EnemyRobot el cual será el objeto de nuestras pruebas.

Un prefab es un elemento que sirve como plantilla, es decir, cuando preparamos un personaje, un algoritmo, etc. la combinación de elementos que componen a ese personaje puede exportarse como un prefab. Ese prefab es la base para el objeto que se crea en la escena y que nos aseguramos que comparte características con el resto de sus “hermanos” o demás copias que hagamos de un prefab. Además, como ventaja si cambiamos algo del prefab, todos los objetos que han sido creados a partir de este reciben ese cambio. Este concepto viene determinado por que C# es un lenguaje que soporta el paradigma de orientación a objetos.

Este hecho del uso de los prefabs, nos permite un desarrollo muy sencillo de los elementos del juego que veremos a continuación.

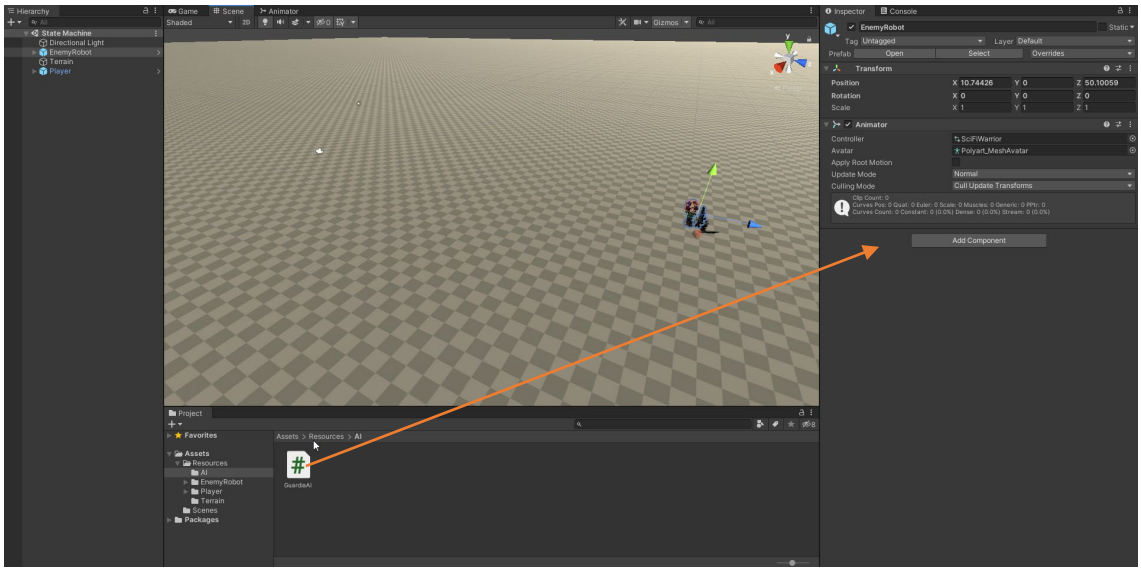


Figura 5 Añadir un Componente a un objeto

Si ha probado la escena se habrá dado cuenta de que nuestro EnemyRobot no hace nada. Esto es porque es necesario incorporarle un componente que lo dote de inteligencia. En el caso de Unity ese componente va a ser un Spript escrito en C#. Bajo la Carpeta Resources>IA entre los Assets hallará un script que se llama GuardiaIA. Llega el momento de darle una cierta inteligencia a este enemigo, para ello incluiremos el Script que figura bajo la carpeta AI y que tiene el nombre de GuardiaAI. Para añadir este script simplemente cogeremos el mismo y lo tiraremos en la encima del inspector teniendo seleccionado el EnemyRobot, como se muestra en Figura 5. También se puede añadir encima del objeto o usar el menú que pone Add Component, el resultado es indistinto y debiera de quedar como se muestra en la vista del inspector (Figura 6). Por último, para terminar la configuración de esta parte añadiremos en la variable Player del Inspector el objeto Player simplemente arrastrando desde la Jerarquía de la izquierda.

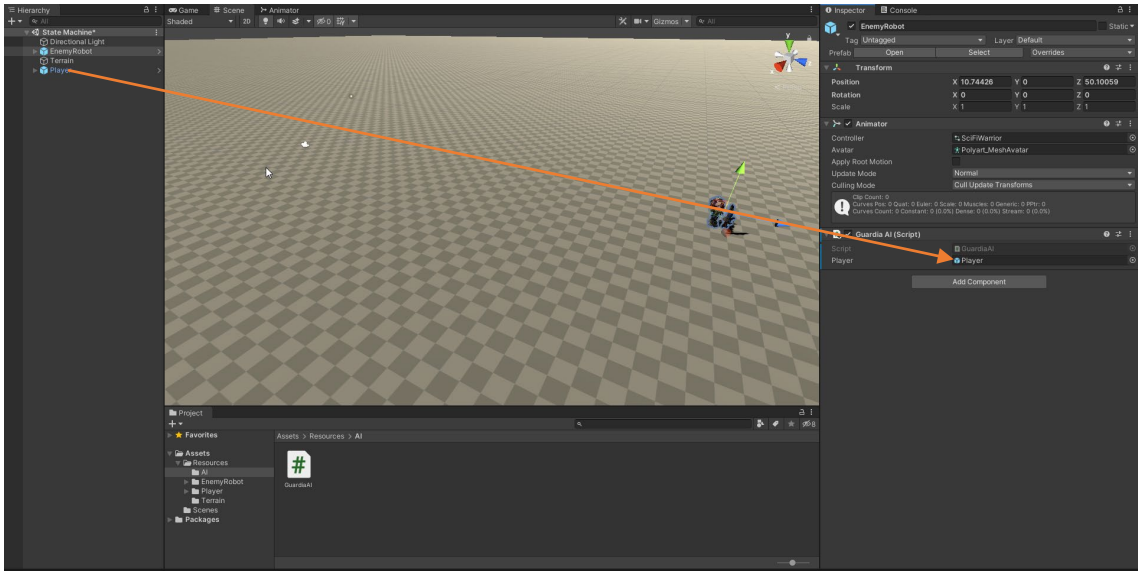


Figura 6 Añadir el player como objetivo del EnemyRobot

Vamos ahora a analizar un poco el código del Script para entender que es lo que está pasando.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GuardiaAI : MonoBehaviour
{

    public GameObject player;
    Animator anim;

    float rotationSpeed = 2.0f;
    float speed = 2.0f;
    float visDist = 40.0f;
    float visAngle = 30.0f;
    float shootDist = 20.0f;

    string state = "IDLE";

    // Use this for initialization
    void Start()
    {
        anim = this.GetComponent<Animator>();
    }

    // Update is called once per frame
    void Update()
    {
        Vector3 direction = player.transform.position - this.transform.position;
        float angle = Vector3.Angle(direction, this.transform.forward);

        if (direction.magnitude < visDist && angle < visAngle)
        {

            direction.y = 0;

            this.transform.rotation = Quaternion.Slerp(this.transform.rotation,
                                      Quaternion.LookRotation(direction),
                                      Time.deltaTime * rotationSpeed);

            if (direction.magnitude > shootDist)
            {
                state = "WALKING";
                this.transform.Translate(0, 0, Time.deltaTime * speed);
            }
            else
            {
                state = "SHOOTING";
            }
        }
        else
        {
            state = "IDLE";
        }
    }
}
```

Figura 7 Código del GuardiaAI

El código contenido en dicho Script se puede ver en Figura 7. Aquí podemos apreciar dos métodos Start y Update. El motor llamará automáticamente al primero la primera vez que se cargue un objeto, mientras que el segundo será llamado en cada ciclo de ejecución del motor adaptándolo a los FPS a los que corra el juego. El código comienza con la declaración de una serie de variables. La primera de ellas es pública y se verá en el inspector, de hecho, es donde hemos asignado e objeto Player. Las otras definen un cono de visión para nuestro robot. Si el

objeto que hayamos identificado Player entra en el cono de visión, el robot se aproximará a él hasta que este a menos de la distancia de disparo cuando procederá a atacar. Para conseguir ese comportamiento, en el método Update lo primero que hacemos es calcular la dirección en que está el jugador y el ángulo al mismo. Con estos dos elementos procederemos a codificar nuestra máquina de estados en base a una estructura condicional IF-THEN-ELSE. En este caso estamos codificando que si no está en el cono de visión el robot pasa al estado IDLE, mientras que si está calcularemos la rotación para mirar al Player y dependiendo de si está a distancia de disparo nos moveremos hasta estar en la misma con el estado WALKING o bien procederemos al estado SHOOTING.

A mayores en el estado WALKING procederemos a actualizar la posición de nuestro personaje si este está fuera del rango de disparo mediante una traslación del objeto. Podemos probar ahora a ver si el personaje se acerca a nosotros ejecutando el juego.

Todo debiera de ir bien salvo por el hecho de que nuestro EnemyRobot permanece estático. Vamos a asociar el comportamiento a cambios en la animación, esto será con otra máquina de estados que se en la pestaña Animator. Si no está abierta, se puede abrir en el menú Window>Animatio>Animator. Está nos mostrará la máquina de estados con la que Unity controla la animación.

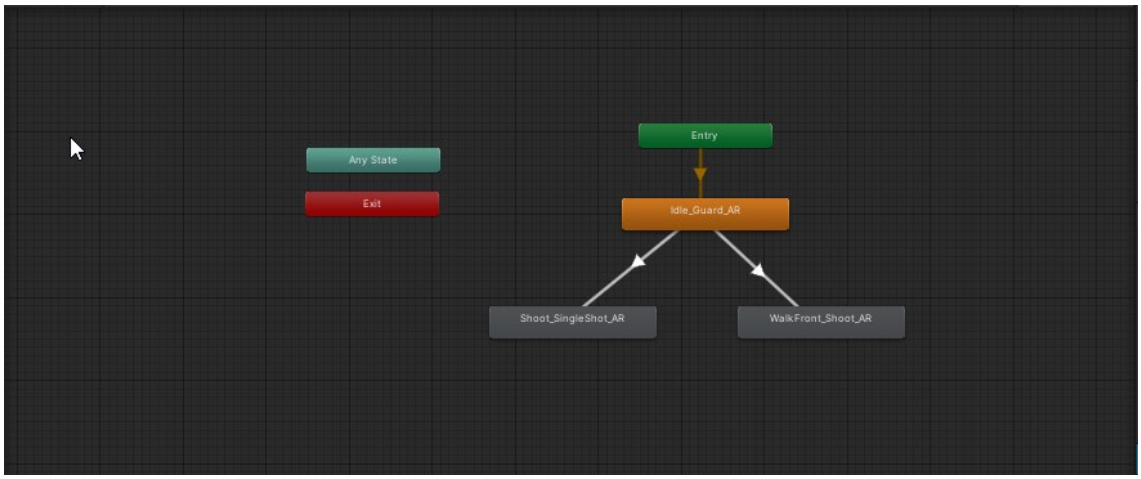


Figura 8 Máquina de estados del Animator

Como podemos ver en la Figura 8 tenemos una serie de estados y transiciones entre ellos. Para poder controlar la animación vamos, en primer lugar, a necesitar una serie de parámetros. En concreto, los parámetros que vamos a tener en cuenta son en primer lugar si el Player es visible. Para ello definimos una variable Bool a la que llamaremos esVisible y le asignaremos de salida el valor False (Figura 9, Izquierda), a continuación, el otro valor que nos interesa es la distancia al Player por lo que definiremos una variable Distancia de tipo Float (Figura 9, Derecha) y le asignaremos el valor 1000 inicialmente para asegurarnos de que ninguna transición se dispara indebidamente. Importante, tenga cuidado con cómo les llama a las variables ya que posteriormente usaremos esos nombres para alterar la animación.

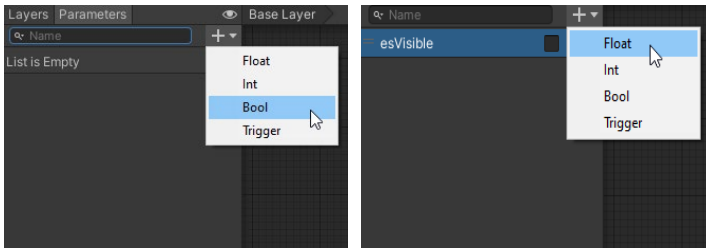


Figura 9 Insertar valores Bool (Izquierda) y Float (Derecha)

Una vez hecho este paso, vamos a volver al script GuardiaIA. En este vamos a añadir un par de líneas con el fin de que este script se comunique con el Animator para actualizar la animación. Para ello solo tendremos que añadir unas pocas líneas, en concreto, después del primer if colocar las siguientes líneas de código:

```
if (direction.magnitude < visDist && angle < visAngle)
{

    direction.y = 0;
    anim.SetBool("esVisible", true);
    anim.SetFloat("Distancia", direction.magnitude);

    this.transform.rotation = Quaternion.Slerp(this.transform.rotation,
                              Quaternion.LookRotation(direction),
                              Time.deltaTime * rotationSpeed);
```

Figura 10 Código para inyectar en la animación las transiciones

y además tras el else la siguiente línea.

```
    }
    else
    {
        state = "IDLE";
        anim.SetBool("esVisible", false);
    }
}
```

Figura 11 Código para modificar la animación

Por último, y para que todo funcione correctamente, ahora que los disparadores de los estados están configurados, solo nos falta definir las transiciones en el Animator. Empezamos seleccionando la que va del estado IDLE a Walk. Haciendo esto, aparecerá en el Inspector las propiedades del enlace, así como los disparadores que son necesarios. En este caso vamos a definir que si el jugador es visible pero su Distancia es superior a 20 (valor definido en la clase GuardiaAI como valor de la distancia de disparo) la animación debe transicionar hasta WALKING (Figura 12).

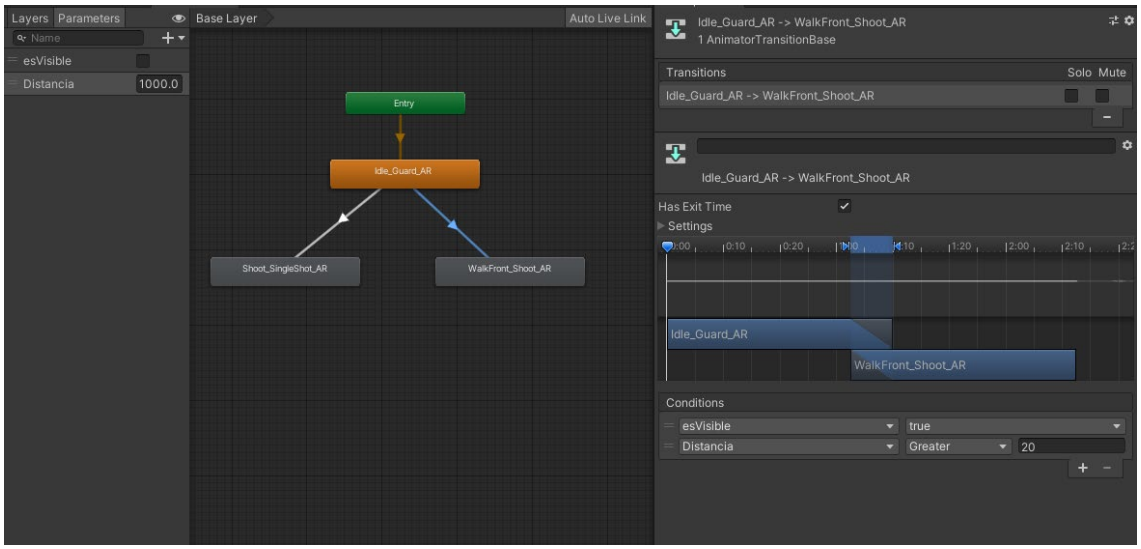


Figura 12 Restriciones en la animación

Probamos este comportamiento arrancando el juego y acercándonos al EnemyRobot desde la parte frontal. Si todo funciona correctamente ahora intentamos alejarnos lo suficiente, pero vemos que no se para la animación. Deberemos por lo tanto de completar la animación con otra transición de WALK a IDLE con la condición de que el Player ya no sea visible. Volvemos a probar el funcionamiento. Para ello bastará con presionar el botón dercho del ratón sobre el estado Walking y seleccinar Make Transition poniendo como destino el estado IDLE, Figura 13.

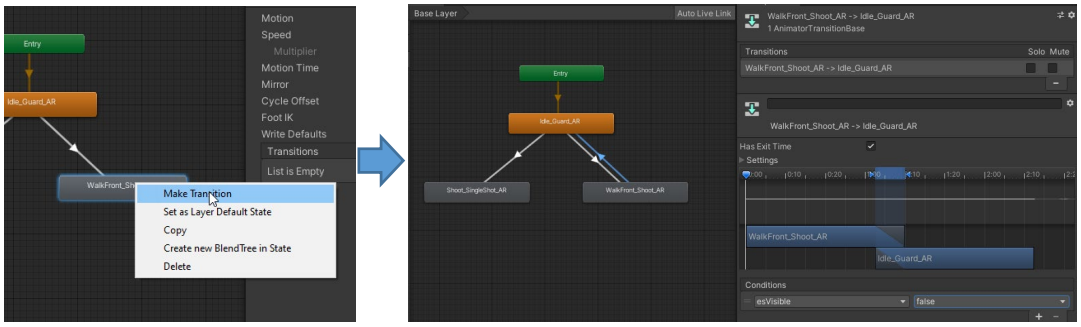


Figura 13 Insertar nueva transición

Ejercicio. El diagrama final de transiciones debiera de ser similar al que se muestra en Figura 14. Complete las transiciones y sus condiciones para que todo funcione correctamente. Este ejercicio debe de entregarse completado mediante la exportación de un paquete de Unity. Encontrará la opción en Asset>Export Package. Asegurese de tener seleccionada la escena en la jerarquía antes de exportar y la carpeta Resources en el Assset Manager.

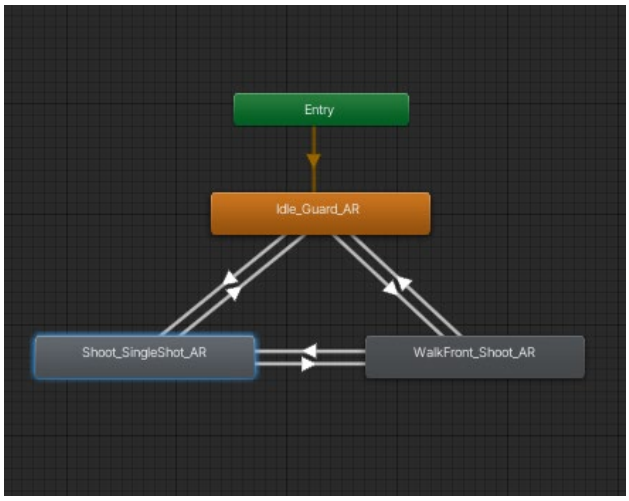


Figura 14 Esquema final de transiciones

Comentar que, como se puede ver, Unity tiene una máquina de estados implementada de por si asociada a las animaciones pero que se puede utilizar sin animaciones para diseñar e implementar las máquinas de estado de nuestros personajes. Estos estados están compuestos por clases que se implementan en código y que constan de 3 métodos (Awake, Upadate, onExit) que se ejecutan respectivamente al ser invocado, en cada actualización del bucle de juego y cuando se transiciona del estado otro. En este caso no tiene mucho interés su uso ya que el comportamiento en si es sencillo, pero de implementarse una máquina de estados más compleja sí que sería útil. Además, cabe reseñar que el coste computacional de la máquina de estados, al estar ya implementada y el bucle funcionando de continuo para las animaciones, es prácticamente nulo.

Segundo Día de Prácticas

GOAP (Goal Oriented Action-Planning)

Surgido a partir del año 2000 con el juego FEAR, los sistemas GOAP se han hecho muy populares dentro del desarrollo de los videojuegos debido a la flexibilidad que ofrecen. En este sistema ya no se establecen estados y condiciones, sino que lo que tenemos son objetivos que queremos cumplir y disponemos de un conjunto de acciones las cuales cuentan con precondiciones y poscondiciones. El sistema de planificación parte de los objetivos y va urdiendo un plan ejecutando acciones hasta que las precondiciones de todas las acciones necesarias para alcanzar el objetivo se alcanzan.

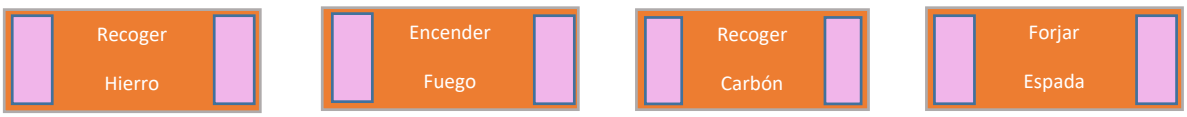


Figura 15 Ejemplos de Acciones que puede realizar un agente

Así por lo tanto lo que debiéramos es de definir un conjunto de acciones similar al que se ve en el la Figura 15. En dicha figura cada una de las cajas representa una acción que puede tomar el agente, los cuadros rosáceos, representan las precondiones para poder efectuar la acción y los efectos posteriores o poscondiciones una vez se ha completado la acción. Con estas acciones se pueden definir una sería de objetivos que pretendemos que realicen nuestros agentes en

función del estado del mundo. Por ejemplo, imaginemos que queremos codificar a un espadero dentro del juego, este necesitaría 10 unidades de Hierro para la espada y 20 unidades de carbón. En la Figura 16, se pueden ver varios ejemplos de planes en función del estado del mundo que se puede ver en la parte Izquierda representado como una caja con los recursos y las variables de estado, por su parte en la parte derecha se puede ver el objetivo que no es otro que conseguir una espada. Todos esos planes se pueden realizar con las acciones definidas y lo mejor que de todo es que van a ocurrir en función de los datos no de las acciones que tengamos predeterminadas dándole una gran flexibilidad al agente, por ejemplo, si no tiene suficiente carbón ira a por él hasta que lo tenga antes de proceder con el siguiente paso del plan, pero si hay, pues no será necesario.

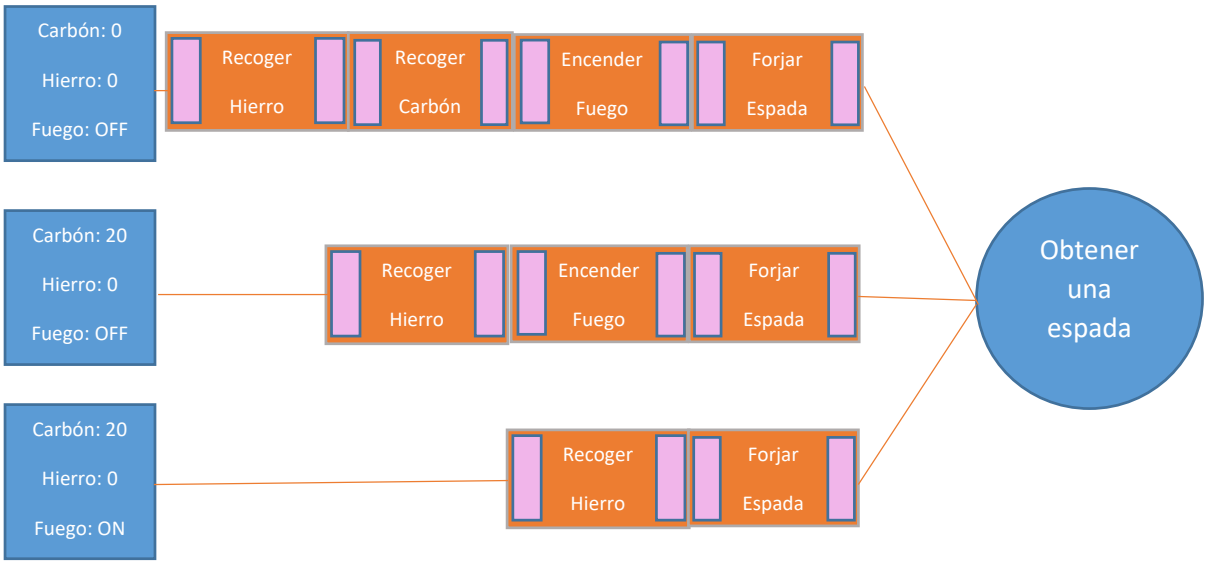


Figura 16 Ejemplos de planes de ejecución del GOAP

A continuación, vamos a crear una pequeña escena dentro de Unity que utilice este tipo de comportamiento para ello usaremos una librería externa que se incorpora como un asset. En concreto, al igual que en el caso de la máquina de estados, vamos a crear un nuevo proyecto e importaremos el paquete adjunto a esta lección y que tiene el nombre de GOAP_Inicial.unity. Importando todos los elementos asociados en la sección de Assets el primero de los pasos será el abrir la escena village simplemente haciendo doble clic sobre la misma.

Una vez hecho esto debiéramos tener una escena similar a la que se puede ver en la Figura 17, donde podemos ver a la izquierda la jerarquía en la que hemos creado una serie de puntos de interés a los que le hemos llamado windmill, Bakery y Market. A mayores con un pequeño cilindro Rosa hemos creado al protagonista de nuestra prueba que va a ser el Baker en este caso. Nuestro objetivo en esta escena será conseguir hacer pan e ir al mercado a venderlo. Para ello como podemos ver hay una carpeta que se llama IA y que contiene todo el sistema que nos va a hacer falta en esta primera aproximación. Si analizamos el contenido de dicha carpeta nos encontraremos lo que se muestra en la Figura 18.

En dicha carpeta encontraremos otras 4 carpetas las cuales identifican las partes del sistema, en concreto nos interesan las subcarpetas State, Agents y Actions principalmente.

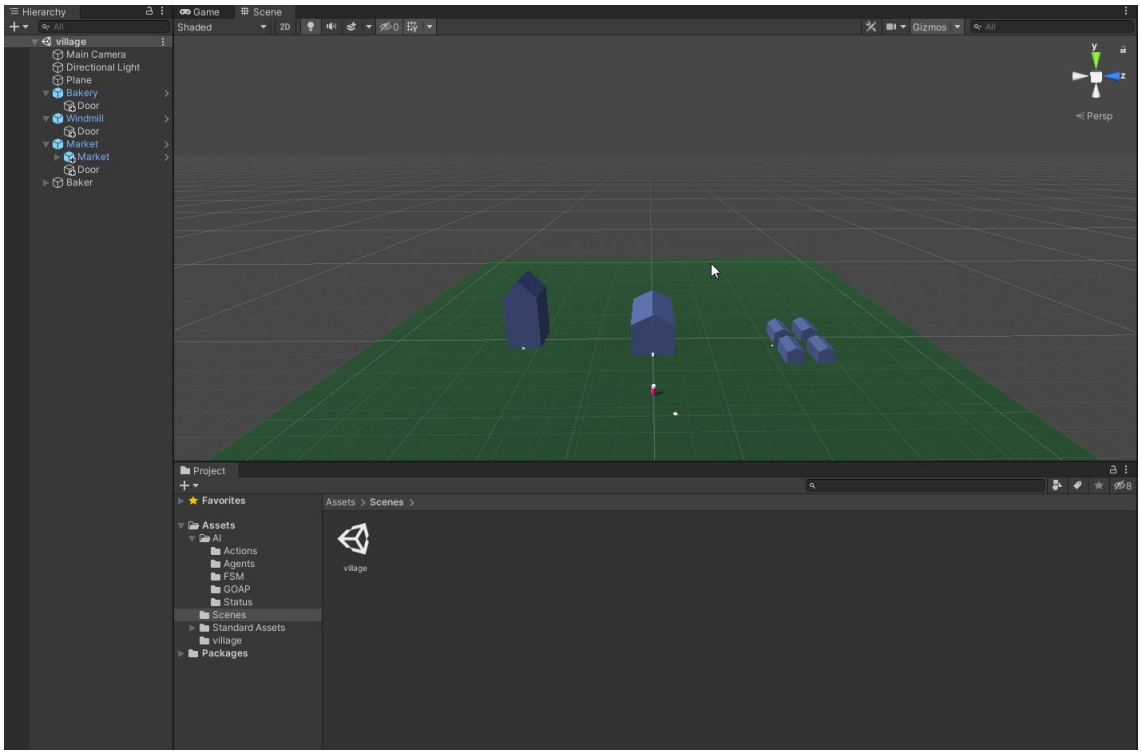


Figura 17 Escena para las pruebas del GOAP

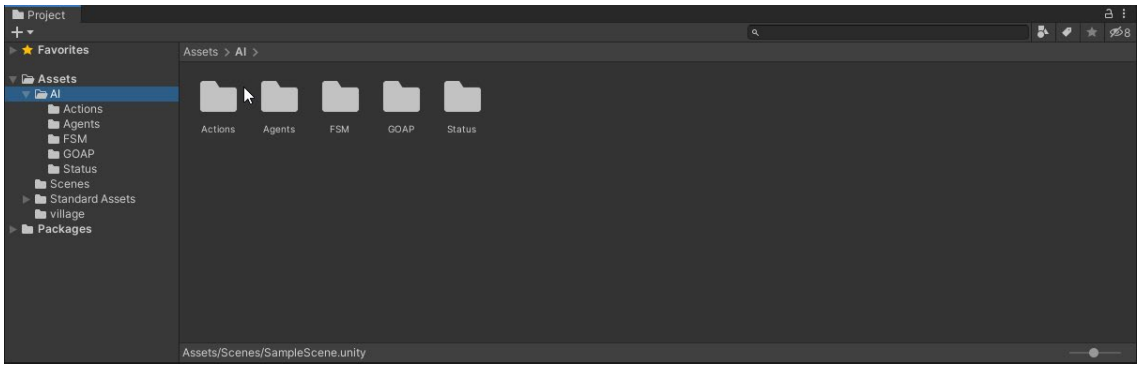


Figura 18 Contenido de la carpeta IA

Dentro de la carpeta State encontraremos un único fichero, Inventory, el cual será el encargado de guardar el estado para nuestro agente el contenido se pude ver en la Figura 19. Como se puede ver en la mencionada figura, lo único que hace es Desplegar el inventario y llevar la cuenta del nivel de harina y pan, los cuales usaremos posteriormente en la pre y post condiciones.

Este inventario está asignado a nuestro agente Baker, lo cual puede comprobarse en el Inspector sin más que seleccionar el objeto en cuestión, como se ve en Figura 20. En esta se aprecia como además de los componentes necesario para que se mueva el agente como el Nav Mesh Agent, tenemos añadidos 3 componentes adiciones, el mencionado Inventario, además de los scripts que hacen que todo funcione, Worker, responsable del al lógica del Agente y Goap Agent que será el responsable de permiter que el sistema GOAP funione.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Inventory : MonoBehaviour {

    public int flourLevel = 5;
    public int breadLevel = 0;

    void OnGUI()
    {
        GUI.Box(new Rect(0, 0, 100, 100), "Inventory");
        GUI.Label(new Rect(10, 20, 100, 20), "Flour: " + flourLevel);
        GUI.Label(new Rect(10, 35, 100, 20), "Bread: " + breadLevel);
    }
}
```

Figura 19 Código de la clase Inventory

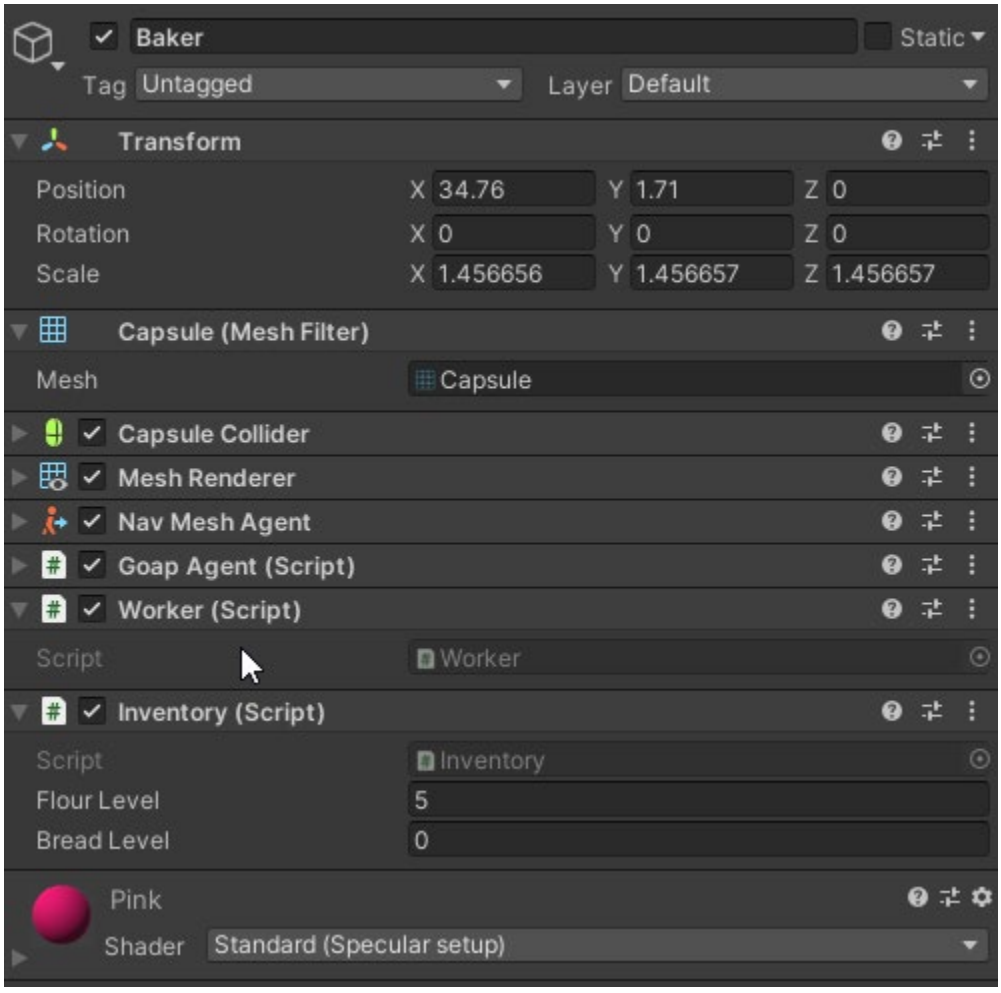


Figura 20 Inspector del Baker

El siguiente punto será en el que añadiremos las acciones y los objetivos. En primer lugar, vamos a centrarnos en que nuestro panadero haga su trabajo dándole un objetivo y registrando las variables del mundo que usará el sistema GOAP. Para ello abrá el script Worker que se encuntra en la carpeta Agents.

El código que se encontrará es un código que contempla una serie de funciones que si bien la parte que nos interesa es la primera en la que se realiza un registro de las condiciones y los objetivos. Si nos fijamos en la Figura 21, veremos que lo que destaca es que en la primera de las líneas vemos que la clase Worker es que herede no solo de MonoBehavior si no de IGoap que es lo que la identifica en el sistema como una clase de control.

```
public class Worker : MonoBehaviour, IGoap
{
    NavMeshAgent agent;
    Vector3 previousDestination;
    Inventory inv;

    void Start()
    {
        agent = this.GetComponent<NavMeshAgent>();
        inv = this.GetComponent<Inventory>();
    }

    public HashSet<KeyValuePair<string,object>> GetWorldState ()
    {
        HashSet<KeyValuePair<string,object>> worldData = new HashSet<KeyValuePair<string,object>> ();
        worldData.Add(new KeyValuePair<string, object>("hasFlour", (inv.flourLevel > 1) ));
        return worldData;
    }


    public HashSet<KeyValuePair<string,object>> CreateGoalState ()
    {
        HashSet<KeyValuePair<string,object>> goal = new HashSet<KeyValuePair<string,object>> ();
        goal.Add(new KeyValuePair<string, object>("doJob", true ));

        return goal;
    }
```

Figura 21 Código del Worker

Lo siguiente que debe de hacer en su caso es añadir el estado en el mundo, en concreto el código provisto carece de la línea que puede ver en la Figura 21 comenzando por worldData. Esta línea lo único que le dice al sistema es cuando hay suficiente harina en este caso cuando en el inventario hay más de uno. Además, debemos de darle un objetivo a nuestro Baker y eso lo registraremos en la función CreateGoalState, nuevamente, el código provisto carece de la línea que comienza por goal está lo que le dice a nuestro agente es que su objetivo es hacer su trabajo. Una cosa que se debe descartar es que se debe de ser cuidadoso con la ortografía tanto dela condición hasFlour, como con el objetivo doJob ya que después usaremos estás palabras clave en las acciones.

Una vez hecho esto, salvamos el código y pasamos a incluir la primera de las acciones que se encuentra en la carpeta Actions. En concreto, abrimos el script BakeBread cuyas partes más importantes pueden verse en Figura 22. En esta figura en la parte superior podemos ver como establecemos las precondiciones de esta acción y el efecto que tendrá al final. A mayores, en la función perform se codifica los efectos adicionales que suceden cuando se ejecuta la acción en este caso una espera y la modificación del inventario en donde restamos dos unidades de harina para crear una de pan.

```
public class BakeBread : GoapAction {

    bool completed = false;
    float startTime = 0;
    public float workDuration = 2; // seconds

    public BakeBread () {
        addPrecondition ("hasFlour", true);
        addEffect ("doJob", true);
        name = "BakeBread";
    }

    public override void reset ()
    {
        completed = false;
        startTime = 0;
    }
```

```
public override bool perform (GameObject agent)
{
    if (startTime == 0)
    {
        Debug.Log("Starting: " + name);
        startTime = Time.time;
    }

    if (Time.time - startTime > workDuration)
    {
        Debug.Log("Finished: " + name);
        this.GetComponent<Inventory>().flourLevel -= 2;
        this.GetComponent<Inventory>().breadLevel += 1;
        completed = true;
    }
    return true;
}
```

Figura 22 Código de BakeBread

Con esto no nos queda más que añadir el script a nuestro agente Baker, como se muestra en Figura 23. A mayores recuerde ponerle el Target que no será otro que la puerta de nuestra Bakery.

Si prueba ahora el sistema debiera de ver como el panadero se acerca a la panadería y, una vez allí, comienza a consumir harina y hacer pan hasta que se queda sin suficiente harina.

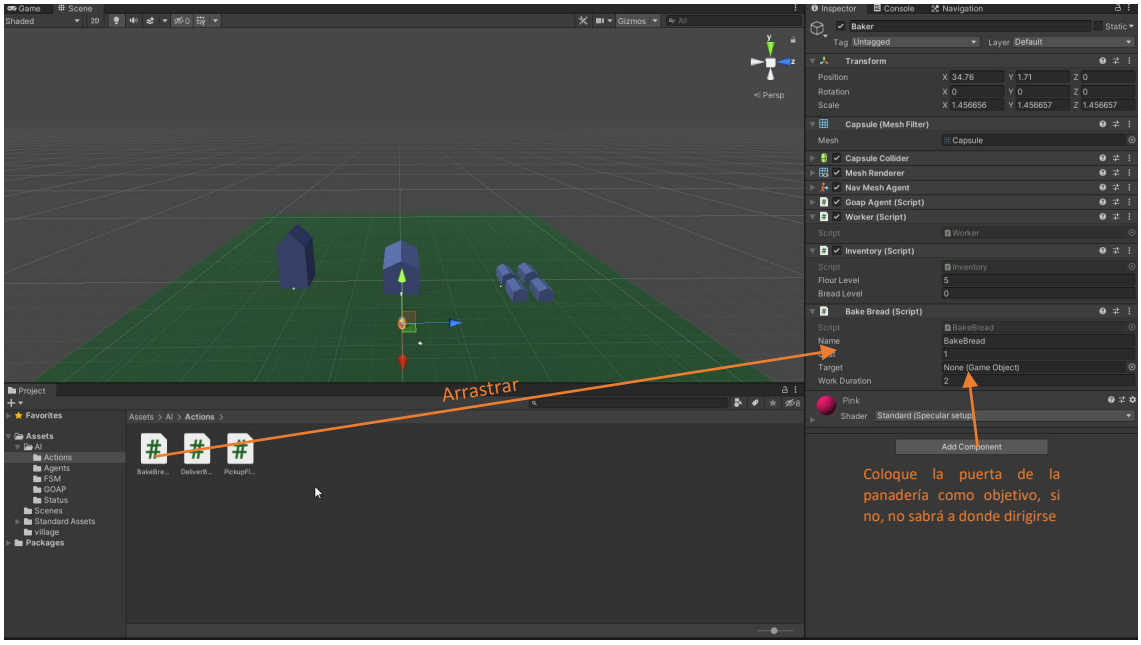


Figura 23 Añadir la acción BakeBread

Bueno, si esto funciona, el siguiente paso será añadir la acciónen PickupFlour. Está acción lo que hará será añadir 5 de harina cuando el agente este sin harina. Puede comprobar el código para corroborar como se establecen las condiciones y los efectos. El siguiente paso no es otro que añadir este script a nuestro agente como en el caso anterior, Figura 24. Con solo este pequeño cambio si probamos el comportamiento veremos que el sistema es capaz de ir a por harina cuando se le ha acabado sin necesidad de codificar absolutamente nada más solo registrando una nueva acción.

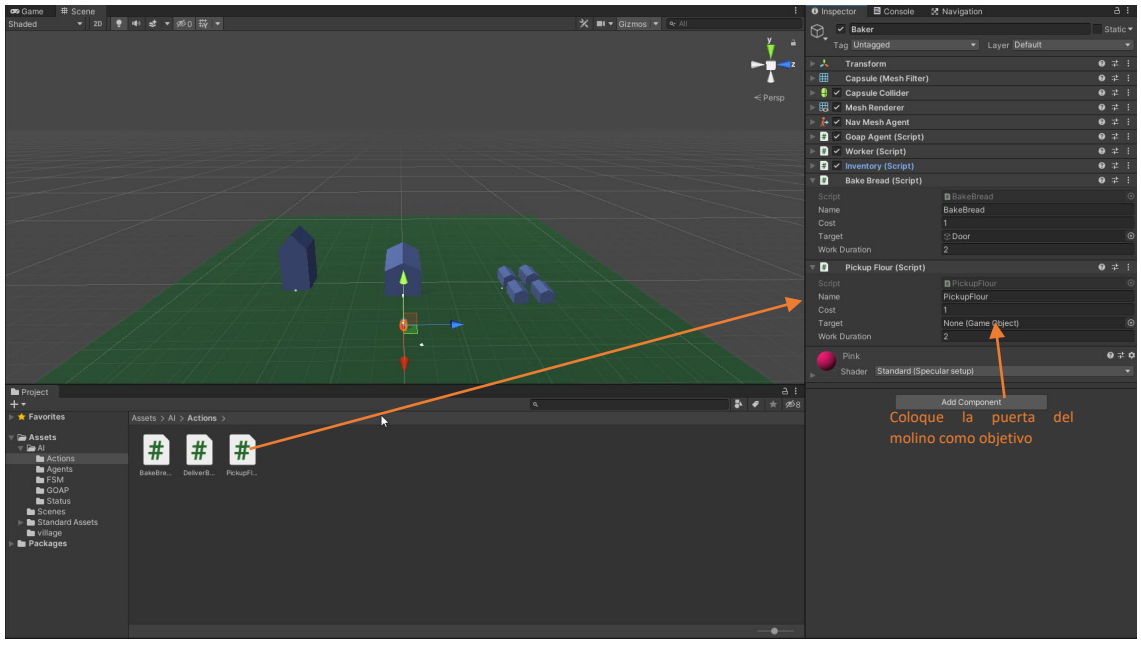
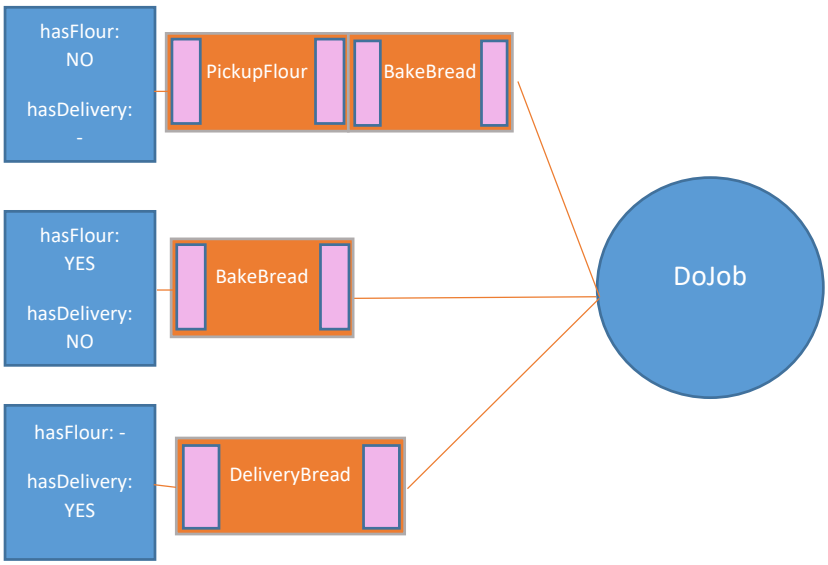


Figura 24 Añadir PickupFlour

Para añadir algo más de complejidad, en este caso vamos a darle una segunda posibilidad a nuestro panadero y es que cuando tenga cinco barras de pan, las entregue en el mercado. Para esto, tenemos otra acción, pero vamos a necesitar crear una condición adicional y eso implica

modificar el script Worker. En definitiva, lo que queremos es darle a nuestro panadero las siguientes posibles líneas de comportamiento:



Para ello es necesario definir ese atributo hasDelivery lo que haremos modificando el script Worker y añadiendo la siguiente línea que se ve en la Figura 25

```
public class Worker : MonoBehaviour, IGoap
{
    NavMeshAgent agent;
    Vector3 previousDestination;
    Inventory inv;

    void Start()
    {
        agent = this.GetComponent<NavMeshAgent>();
        inv = this.GetComponent<Inventory>();
    }

    public HashSet<KeyValuePair<string,object>> GetWorldState ()
    {
        HashSet<KeyValuePair<string,object>> worldData = new HashSet<KeyValuePair<string,object>> ();
        worldData.Add(new KeyValuePair<string, object>("hasFlour", (inv.flourLevel > 1) ));
        worldData.Add(new KeyValuePair<string, object>("hasDelivery", (inv.breadLevel > 4) ));
        return worldData;
    }
```

Figura 25 Modificar el Worker para añadir nuevas condiciones

Como se ve en la mencionada figura, hemos añadido una condición que lo que permite es comprobar si tenemos una entrega cuando hay más de cuatro panes. Una vez hecho esto, vamos al script en de la acción DeliveryBread, Figura 26.

Como vemos en está ocasión ya no se comprueba el estado del hasFlour ya que lo único que nos interesa es si hay suficiente pan para entregar. A mayores en el método perform de esta clase se restan los cinco panes mencionados anteriormente.

```
public class DeliverBread : GoapAction {

    bool completed = false;
    float startTime = 0;
    public float workDuration = 2; // seconds

    public DeliverBread () {
        addPrecondition ("hasDelivery", true);
        addEffect ("doJob", true);
        name = "DeliverBread";
    }

    public override void reset ()
    {
        completed = false;
        startTime = 0;
```

Figura 26 Nueva acción DeliveryBread

Hechas estás modificaciones, solo nos falta añadir la acción a nuestro agente exactamente igual que las anteriores y colocar en Target la puerta del mercado, .

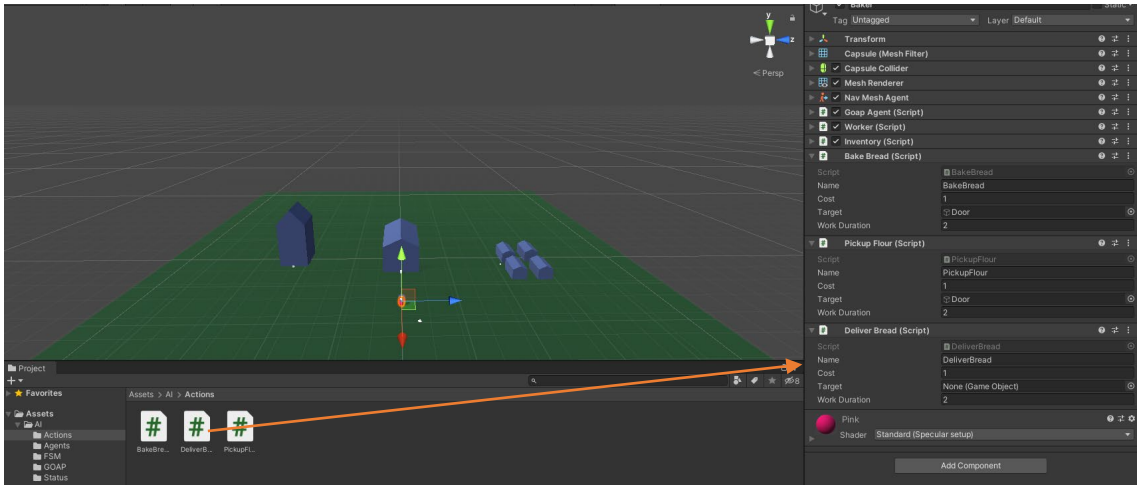


Figura 27 Añadir DeliveryBread como acción

Hecho esto podemos volver a comprobar el comportamiento y veremos que es el propio y ya se ha añadido ese proceso que comentábamos. Este procedimiento podría seguir indefinidamente y lo único que tendríamos que hacer para generar nuevos comportamientos cada vez más complejos es darle los objetivos correspondientes a nuestros agentes y las acciones pertinentes. Esto, además, hace que depurar posibles errores sea un proceso muy sencillo.

Ejercicio. Entregue en la tarea de Moodle asignada un paquete con los todo el asset con el tutorial completado.

Por último, y como ejemplo de un sistema GOAP multiagente, cierre el proyecto actual y cree uno nuevo. En este importaremos el Paquete GOAP_Multiagente.

Este ejemplo contiene un escenario similar al anterior, ábralo. El objetivo de este último ejemplo es que vea cómo se pueden crear varios comportamientos complejos que interaccionen entre si de una manera simple.

En este caso hemos añadido un segundo agente llamado Farmer. El cual tendrá dos acciones posibles Harvest que se efectuará en el campo que también hemos añadido y DeliveryWheat que llevará el trigo al molino incrementando en una unidad la cantidad de trigo disponible. En el ejemplo anterior, el Molino tenía un stock ilimitado de trigo para la harina. Ahora, hemos añadido un segundo inventario en este caso al molino para que el Baker tenga que esperar si no hay suficiente.

Puede comprobar el funcionamiento, para ello cargue la escena correspondiente y asegúrese que las puertas están colocadas y los inventarios correctamente asignados. Si todo está bien en ese caso, proceda a presionar Play y vea la simulación como funciona con sólo haber añadido las condiciones y las acciones correspondientes.