

Guía de Programación y Referencia Práctica Unity

Introducción.....	2
Definir EQUIPO.....	2
Servicios de EQUIPO : ISSR_TeamBehaviour.cs.....	2
Código ejemplo de crear equipo.....	3
Servicios de AGENTE: ISSR_Agent.cs.....	4
Bases	4
Métodos obligatorios.....	4
Código ejemplo de agente mínimo.....	5
Resumen de eventos y servicios de acción en el juego (videoreferencia)	5
Percepción Visual	5
Desplazándose	5
Cogiendo y Soltando Objetos.....	6
Llegada de Objeto a Meta.....	6
Colisiones	6
Piedras grandes.....	6
Acciones y Eventos Especiales	6
Gestores de eventos	7
Clase ISSR_Message	10
Servicios de Acción.....	10
Servicios de Información de Objetos	12
Servicios de Información.....	14
Resumen de Variables de la clase ISSR_Agent.....	15
Variables de solo lectura.....	15
Variables de lectura y escritura	16
Funciones de Ayuda: Clase ISSRHelp	16
Bibliotecas de Funciones Básicas	18
List, Listas de objetos	18
Vector3, para posiciones y direcciones 3D	18
Random, generación de números aleatorios.....	19
Mathf, matemáticas simples con números float	19
Lista de Estados propuestos	20

Introducción

Para crear el comportamiento de los agentes en el juego es necesario codificar dos scripts:

- Uno que defina el **comportamiento del equipo**, necesario para dar identificadores y crear los agentes. Este script debe contener una clase derivada de **ISSR_TeamBehaviour**.
- Uno que defina el **comportamiento de un agente**, que es el más complejo y guiaremos en las prácticas. Este script debe contener una clase derivada de **ISSR_Agent**.

Ambos scripts deben crearse en la carpeta `//Assets/StudentScripts` del proyecto Unity. Estos programas se construyen utilizando las clases auxiliares que se definen en el proyecto dado. El código que hace funcionar el juego con todas las clases auxiliares que se necesitan como base se encuentra en `y //Assets//`. **Este código base no se debe cambiar**. La definición de las clases y su interfaz de programación (API) se describen en este documento.

Definir EQUIPO

En primer lugar, se debe definir un **nombre** y un **apodo** para el equipo de agentes. El apodo debe respetar la regla para los identificadores en C#:

Todo identificador debe comenzar por letra o el símbolo `_` y usar solo símbolos alfanuméricos o `_`

Debes definir

- Nombre: frase corta que da nombre al equipo
- Apodo: identificador corto que tiene exactamente 4 caracteres, no puedes ser ni "SIRS", ni "ISSR", ni "TEST".

Supongamos que el nombre corto elegido es "XXXX". Estas letras serán las primeras de los nombres de fichero para el script de equipo y el script de agente. También serán las primeras del nombre de las clases derivadas en dichos archivos. Entonces por ejemplo dado el apodo "XXXX" hay que crear en la carpeta `//Assets/StudentScripts`:

- Script `XXXX_Equipo.cs` con definición de clase `XXXX_Equipo` derivada de clase **ISSR_TeamBehaviour**.
- Script `XXXX_Agente.cs` con definición de clase `XXXX_Agente` derivada de clase **ISSR_Agent**.

El script con el comportamiento del equipo debe asociarse como componente a un objeto vacío de la escena.

La clase que implementa ese script de comportamiento del equipo debe definir una única función:

```
public abstract void CreateTeam ();
```

El código de esta función llama a los servicios definidos en la clase base **ISSR_TeamBehaviour**.

Servicios de EQUIPO : ISSR_TeamBehaviour.cs

```
public bool InitError()// Ha habido error en la inicialización?
// Si el sistema dice que hay error no se debe seguir

// Registra un equipo de acuerdo con los marcadores de agente que haya
// en la escena: (Patio de Juego), dando sus nombres corto y largo
public bool RegisterTeam(string ShortName, string LongName="")

// Devuelve el número de agentes que habrá en la escena (número de marcadores)
public int GetNumberOfAgentsInTeam()

// Crea un agente en una de las localizaciones definidas por los marcadores
public void CreateAgent( ISSR_Agent AgentDesc, string OptionalName="")
// Basta con pasarle un objeto ISSR_Agent vacío de nuestra clase derivada XXXX_Agent:
//     new XXXX_Agente()
// Opcionalmente se le puede dar un nombre particular a cada agente
```

Código ejemplo de crear equipo

Fichero `XXXX_Equipo.cs` suponiendo que XXXX es el apodo del equipo.

```
public class XXXX_Equipo: ISSR_TeamBehaviour // Clase derivada
{
    public override void CreateTeam() // Única función a definir
    {
        if (!InitError())
        { // Si no hay error en la inicialización
            if (RegisterTeam("XXXX", "Testing Team")) // Verdadero si no hay error
            { // Registrar equipos, nombre corto APODO, nombre largo
                // el nombre largo es opcional
                Debug.Log("Equipo XXXX despierta");

                // Para cada marcador de agente en la escena
                for (int i = 0; i < this.GetNumberOfAgentsInTeam (); i++)
                {
                    CreateAgent(new XXXX_Agente ()); // Crear un agente
                    // La definición del comportamiento del agente está en la clase
                    // XXXX_Agente en el fichero del mismo nombre.
                }
            } // FIN de si no hay error
        } // FIN de CreateTeam()
    }
}
```

En ese código habría que cambiar XXXX por el nombre corto del equipo que se defina. Ese nombre corto debe preceder a cualesquiera otros nombres de identificadores públicos (**public** en C#) que se definan, para que no colisionen con los definidos por otros alumnos. La palabra reservada **override** es necesaria para redefinir el método `CreateTeam()`.

Para que el juego cree un equipo deben cumplirse las siguientes condiciones:

- Tiene que haber al menos una piedra en el escenario
- Tiene que haber al menos una bandera en el escenario
- Tiene que haber al menos un marcador de agente en la escena (del mismo color que la bandera)
- El script con el comportamiento del equipo debe asociarse a un objeto vacío de la escena.

El siguiente paso es definir el comportamiento de un agente.

Servicios de AGENTE: ISSR_Agent.cs

Bases

Como se ha visto los alumnos deben codificar una clase derivada de **ISSR_Agent** para implementar el comportamiento de sus agentes. Como se establece en el apartado de creación de un equipo esta clase se referencia como parámetro en la llamada a **CreateAgent()** dentro de la clase del equipo para crear los distintos agentes.

El código de la clase derivada codificado por los alumnos debe redefinir métodos de la clase base **ISSR_Agent** mediante la palabra clave **override**:

- Dos **métodos obligatorios** para llamar al inicio y periódicamente. **Start()** y **Update()**
- Una serie de **gestores de eventos (event handlers)** a los que el juego llamará para informar al agente de cosas importantes que han ocurrido en el juego, o han ocurrido al agente. Estas funciones comienzan por **on** y tiene la forma **onEEE()** donde EEE nombra lo que ha pasado. Por ejemplo, cuando un agente ha chocado con un objeto, **onCollision()**, cuando recibe un mensaje de otro agente **onMsgArrived()** o cuando ha completado una tarea como agarrar una piedra **onGripSuccess()**.

El código de los alumnos puede usar una serie de **servicios** que le da la clase base **ISSR_Agent** para realizar acciones u obtener información del juego, también puede consultar una serie de **variables** que se actualizan automáticamente para dar al agente diversas informaciones como los objetos que tiene a la vista o el objeto que tiene agarrado (*gripped*).

Los servicios los da la clase base como métodos a los que llamar, y son de tres tipos:

- **Información**, proporcionan información sobre la configuración del juego y su estado actual. Las funciones de este tipo comienzan por **i** (**iServices**). Por ejemplo, dónde está la meta, cuanto mide el escenario, cuantos compañeros hay en mi equipo.
- Información de **Objetos**, dan información de los objetos tal como los ve un agente. Las funciones de este tipo comienzan por **oi** (**oiServices**). Por ejemplo, de qué clase es un objeto, dónde está, si está agarrado por algún agente. Si utilizamos un servicio de información sobre un objeto que no está a la vista o ha desaparecido del juego la función dará un error.
- **Acción**, llamadas para pedir a un agente que haga cosas, sus funciones comienzan por **ac** (**acServices**). Es importante decir que si pedimos algo que viola las reglas del juego la función devolverá un error y el agente no hará lo que hemos pedido. Algunas acciones se piden y cuando se completan se informa al agente mediante un evento. Ejemplos son ir a un lugar, agarrar un objeto, enviar un mensaje.

Métodos obligatorios

Estos dos métodos es obligatorio redefinirlos con **override** en la clase derivada que implementa el agente:

```
public virtual void Start ()
```

```
// Este método lo llama la API una única vez antes de cualquier otra función.
```

```
/// Puede utilizarse para inicializaciones.
```

```
public virtual IEnumerator Update()
```

```
// Corutina, para realizar tareas periódicas, no debe retener el control, bloquearía a
```

```
// Unity. La utilizaremos para generar el evento periódico onTickElapsed
```

Las corrutinas son como hilos (*threads*) pero en forma de una función que devuelve el control a Unity mediante el comando **yield**. Además de las dos funciones obligatorias habrá que definir los **gestores de eventos**, son los sensores del agente para saber lo que ocurre en su entorno. El juego llama automáticamente a esos métodos si los redefinimos, en ellos pondremos código para responder a lo que ocurra en el juego.

Nota: no confundir los métodos **Start()** y **Update()** de un agente con los métodos nativos de Unity para la clase **MonoBehaviour**. De hecho, no debemos llamar en el código de un agente a ninguna función de Unity solo a funciones de la API del juego, es decir los servicios del agente y a bibliotecas básicas de funciones de C# que necesitamos.

Código ejemplo de agente mínimo

Un posible código ejemplo mínimo para estas funciones obligatorias para el caso del agente del equipo con apodo XXXX sería:

```
public class XXXX_Agente : ISSR_Agent // Clase derivada
{
    public override void Start ()
    { // Un mensaje de comienzo
        Debug.LogFormat ("{0} comienza", Myself.Name); // Escribe nombre del agente.
    }

    public override IEnumerator Update () // Corutina
    {
        while (true)
        { // Repetir hasta final del juego
            yield return new WaitForSeconds (1f); // Periodo en segundos
            // Importante esperar antes de la primera acción

            current_event = ISSREventType.onTickElapsed; // Generar evento síncrono
            // Insertar código para esta situación
        }
    }
}
```

Resumen de eventos y servicios de acción en el juego (videoreferencia)

A continuación, se resume la utilidad de cada uno de los eventos generados por el juego y de las posibles acciones que puede llamar un agente. Las acciones y eventos están clasificados en varias categorías y en el título de cada sección hay un enlace a un vídeo donde se explican dentro del juego. El nombre del servicio o acción tiene también un enlace al instante de un vídeo donde se muestra cómo es o cómo funciona dentro del juego.

Percepción Visual

- [onEnterSensingArea](#), cuando un objeto del juego (ISSR_Object) entra en el área perceptible por el agente, es decir, comienza a ser visible. Al comienzo del juego un agente recibe de golpe tantos de estos eventos como objetos tenga en su área visible inicialmente.
- [onExitSensingArea](#), cuando un objeto del juego sale del área perceptible por el agente o desaparece, es decir, deja de ser visible.

Desplazándose

- [acGotoLocation](#), ve a una posición en el mundo.
- [onStartsMoving](#), este agente ha comenzado a moverse. Puede usarse como confirmación de éxito cuando se le pide que se mueva.
- [onDestArrived](#), este agente trataba de llegar a una posición del patio de juegos tras [acGotoLocation\(\)](#) y lo ha conseguido.
- [onStop](#), este agente se ha parado, estaba moviéndose y se ha parado. Por ejemplo, tras chocar.
- [acGotoObject](#), ve a la posición de objeto, persiguiéndolo si se mueve.

Cogiendo y Soltando Objetos

- **acGripObject**, agarra un objeto que está dentro de la vista.
- **onGripSuccess**, cuando este agente acaba de “agarrar” con éxito un objeto.
- **acUngrip**, suelta el objeto que tienes agarrado
- **onUngrip**, este agente estaba agarrando un objeto y ha dejado de agarrarlo, ha perdido el agarre.
- **onGripFailure**, este agente estaba tratando de agarrar un objeto y ha fallado, por ejemplo, porque ha salido de su área visible u otro agente ha agarrado ese objeto antes y como las piedras pequeñas solo un agente puede agarrarlo a la vez.

Llegada de Objeto a Meta

- **onGObjectScored**, este agente tenía un objeto agarrado y ha puntuado, es decir ha habido colisión de piedra contra meta o del agente con la meta.
- **onObjectLost**, este agente estaba tratando de llegar a un objeto y ha fallado, normalmente porque el objeto ha desaparecido, después de llamar a **acGotoObject()** o a **acGripObject()**, en este último caso se recibe también **onGripFailure**.

Colisiones

- **onCollision**, este agente (su cuerpo) ha colisionado con otro objeto
- **onGObjectCollision**, el objeto que este agente tenía agarrado ha chocado con otro objeto.
- **onManyCollisions**, este agente está colisionando constantemente con otros objetos (bloqueo), este evento se recibe inmediatamente después de uno de los dos anteriores con la colisión.

Piedras grandes

- **onAnotherAgentGripped**, este agente estaba agarrando una piedra grande y otro agente ha agarrado ahora la misma piedra.
- **onAnotherAgentUngripped**, este agente estaba agarrando una piedra grande y otro agente que también estaba agarrándola ha dejado de hacerlo.
- **onPushTimeOut**, este agente estaba agarrando una piedra grande y pidió moverse (empujarla) pero ha pasado más de un segundo (definido en **ISSR_BStoneBehaviour.MaxWaitTime** como 1) sin que hubiera suficientes agentes empujando la misma piedra con lo que no se ha conseguido cooperar para moverla. La confirmación de que sí se ha podido mover llega en forma de un evento **onStartMoving**.

Acciones y Eventos Especiales

- **acSetTimer**, pon en marcha el temporizador del agente.
- **onTimerOut**, cuando acaba el plazo definido por la acción **acSetTimer(float delay)**
- **onTickElapsed**, este evento puede enviarse periódicamente desde la rutina **Update()** para señalar el paso del tiempo, es el único que genera el código de los alumnos
- **acSendMsg**, envía mensaje.
- **onMsgArrived**, cuando llega un mensaje desde otro agente
- **acSendMsgObj**, envía mensaje con objeto incluido.

Algunas situaciones ejemplo:

- Un agente se está moviendo y choca, recibe eventos: **onCollision()** + **onStop()**
- Un agente está parado con un objeto “agarrado” pero algo choca con el agente, suelta lo cogido y recibe los eventos: **onCollision()** + **onUngrip()**

- Un agente pidió la acción `acGotoLocation()` y ha llegado correctamente a su destino, recibe eventos: `onDestArrived()` + `onStop()`
- Un agente pidió coger un objeto `acGripObject()` llegó al objeto pero falló tratando de agarrarlo, por ejemplo porque otro agente lo estaba ya agarrando (solo piedras pequeñas, las grandes pueden ser sujetadas por más de un agente), recibe `onGripFailure()` + `onStop()`
- Un agente pidió la acción `acGripObject()` y el objeto desapareció de su vista: `onObjectLost()` + `onGripFailure()` + `onStop()` + `onExitSensingArea()`
- Un agente empujando una piedra la hace llegar a la meta, recibe los eventos: `onUngrip()` + `onGObjectScored()` + `onStop()` + `onExitSensingArea()`
- En cualquiera de los casos que se pide una de las acciones siguientes `acGotoLocation()`, `acGotoObject()` o `acGripObject()` justo cuando el agente comienza a moverse, se recibe el evento `onStartMoving()` (al completar la tarea se confirmará con otros eventos)
- Si un agente está agarrando una piedra grande y trata de moverse pero ningún otro agente trata de empujarla a tiempo, se recibe el evento `onPushTimeout()`.

Gestores de eventos

Cada evento tiene un nombre asociado y una función con el mismo nombre, el gestor del evento. Los nombres de los eventos están definidos por en el fichero `ISSR_Event.cs` mediante el tipo enumerado `ISSREventType`. Las funciones gestoras de los eventos tienen en su mayoría un parámetro para dar información adicional.

Como se ha comentado antes, los gestores de eventos son funciones del agente que el alumno debe redefinir en su código mediante el comando **override**, el juego los llama cuando ocurre algo importante al agente como, por ejemplo, se recibe un mensaje, un objeto entra en la vista o el agente ha chocado.

Para utilizar un gestor de eventos en tu código copia una de las cabeceras siguientes sustituyendo **virtual** por **override**.

Para ayudar a la gestión de eventos el juego asigna el tipo de evento a la siguiente variable agente justo antes de llamar a la función de evento, de forma que podemos consultarla en nuestra máquina de estados:

Variables de ayuda para construir una máquina de estados.

```
public ISSREventType current_event; // Evento actual, definido por el tipo enumerado
// ISSR_Event en el fichero ISSR_Event.cs, actualizado por el juego antes de llamar
// a la función gestora del evento que ha tenido lugar.
```

Otra variable que puede servir para mantener el **estado actual del agente**:

```
public ISSRState current_state; // Libre de usar, valores definidos en ISSRDictState.cs
```

Esta otra variable nos permite guardar un estado anterior mientras se resuelve una colisión por ejemplo:

```
public ISSRState last_state; // estado anterior
```

Vemos los gestores de los eventos posibles agrupados por tipos con sus parámetros, después del resumen del apartado anterior:

Percepción en un agente

```
// Cuando el agente comienza a “ver” un objeto, obj el objeto que comienza a ver
public virtual void onEnterSensingArea( ISSR_Object obj)
// Cuando el agente deja de “ver” un objeto, probablemente porque o el agente
// o el objeto se han movido, obj el objeto que deja de verse, sale de zona visible.
public virtual void onExitSensingArea( ISSR_Object obj)
```

Las siguientes variables del agente se pueden consultar para acceder a los objetos visibles en un momento dado por ese agente o el objeto que representa al propio agente:

```
//Todos los objetos visibles por este agente, actualizada por el juego, solo lectura
public List<ISSR_Object> SensableObjects;
// El objeto que representa a este mismo agente, solo lectura
public ISSR_Object Myself;
```

Importante: una variable de tipo `ISSR_Object` representa a cualquier objeto del juego, visible o no. Tratar de obtener información de un objeto no visible puede dar un error. Siempre se puede comprobar si un objeto `obj` es visible por un agente mediante una condición como (ver Servicios de Información de Objetos):

```
if (oiSensible(obj)) ...
```

Más adelante vemos lo básico del manejo de listas, tipo `List`.

Movimiento o locomoción del agente

```
public virtual void onStop() // el agente se ha parado, puede ser por distintas razones
public virtual void onStartMoving() // el agente ha comenzado a moverse
public virtual void onDestArrived()
// el agente ha llegado al destino pedido con la llamada a acGotoLocation()
```

Cuando está tratando de llegar a un objeto o agarrarlo

```
public virtual void onGripSuccess(ISSR_Object obj_gripped)
// cuando se ha llegado al objeto y se ha agarrado sin problemas.
public virtual void onGripFailure(ISSR_Object obj_I_wanted_to_grip)
// cuando se ha llegado al objeto, pero no se ha conseguido agarrarlo
public virtual void onObjectLost(ISSR_Object obj_i_was_looking_for)
// cuando se ha perdido el objeto que se trata de agarrar, ya no puede verlo el agente
En los tres casos el parámetro de tipo ISSR_Object nos indica el objeto que se trataba de agarrar.
```

```
// Cuando tenía agarrado un objeto y dejo de tenerlo agarrado: porque es una piedra y
// he puntuado en la meta o porque otro objeto ha chocado contra este agente.
```

```
public virtual void onUngrip(ISSR_Object ungripped_object)
Parámetro ungripped_object el objeto del que se ha perdido el agarre.
```

Otra variable del agente que es actualizada por el juego nos permite consultar (solo lectura) si tenemos un objeto agarrado y cual, si no hay ninguno agarrado vale `null`.

```
public ISSR_Object GrippedObject;
```

Colisiones y puntuación

```
// Cuando un objeto choca contra el agente, el parámetro es el objeto que ha chocado
public virtual void onCollision(ISSR_Object obj_that_collided_with_me)
// Cuando un objeto choca contra el objeto que este agente está agarrando,
// el parámetro nos dice qué objeto ha chocado contra el que llevo.
public virtual void onGObjectCollision(ISSR_Object obj_that_with_gripped_obj)
```



```
// Cuando el agente está chocando constantemente contra otros objetos, puede servir
// para detectar bloqueos.
```

```
public virtual void onManyCollisions()
```

Nota: el número de colisiones del agente se mide en un periodo de un segundo y si se supera un umbral, este evento se lanza. Aparece inmediatamente después del último `onCollision` o `onGameObjectCollision` que generó la última colisión anotada.

```
// Cuando el objeto que llevo ha puntuado: es decir colisión de piedra contra meta,
// aunque también se puntúa si el agente que lleva la piedra choca contra la meta.
```

```
public virtual void onGameObjectScored(ISSR_Object stone_that_scored)
```

Reglas relativas a las colisiones de los agentes:

- Si un agente se está moviendo con o sin objeto agarrado, **al chocar, se para**. Se generan dos eventos, el de colisión (`onCollision` o `onGameObjectCollision`) y el de parada `onStop`
- Si algo choca contra un agente que está parado mientras está sujetando un objeto, entonces por el impacto suelta lo que esté agarrando, produciendo además de `onCollision` el evento `onUngrip`.

Temporizador

```
// Cuando el agente lanzó su temporizador y el plazo ha expirado, el temporizador
// se lanza con acSetTimer(float delay),
// se consulta si está en marcha con iTimerRunning();
public virtual void onTimerOut(float delay)
```

Recepción de mensajes

- Para colaborar entre los agentes se pueden enviar y recibir mensajes, ver las explicaciones de la clase `ISSR_Message` más adelante.
- No se ha definido ninguna sintaxis de mensajes, un mensaje contiene un código entero definible por el alumno, la identificación del agente que envía el mensaje y datos opcionales.

```
// Cuando se recibe un mensaje, el parámetro es el mensaje
```

```
public virtual void onMsgArrived( ISSR_Message msg)
```

Variables de ayuda asociadas a mensaje recibido

Estas variables las rellena el juego justo cuando antes de llamar a `onMsgArrived()`

```
public int          user_msg_code; // Código de usuario del mensaje recibido
public ISSR_Object  msg_obj; // Objeto transportado en el mensaje recibido
```

Eventos específicos para la cooperación con las piedras grandes:

```
// Cuando un agente ha agarrado la misma piedra grande que estamos agarrando
```

```
public virtual void onAnotherAgentGripped(ISSR_Object agent)
```

```
// Cuando un agente que estaba agarrando la misma piedra grande que yo la ha soltado
```

```
public virtual void onAnotherAgentUngripped(ISSR_Object agent)
```

En ambos casos el parámetro es el agente que acaba de agarrar o soltar. La idea para mover una piedra grande es que el agente la agarre y espere a que al menos otro lo haga también, en ese momento deben moverse ambos hacia la meta de forma sincronizada.

```
// Cuando un agente tiene agarrada una piedra grande y trata de moverse con ella, pero
// no hay ningún otro agente que empuje (se mueva a la vez). En realidad, se espera un
// intervalo de tiempo, antes de generar este Push Time Out (empujar, plazo vencido)
```

```
public virtual void onPushTimeOut(ISSR_Object gripped_big_stone)
```

```
// El parámetro es la piedra que trataba de mover
```

```
// En caso de que sí haya un agente empujando a tiempo se recibirá como confirmación
```

```
// de que se avanza el evento onStartsMoving
```

Clase ISSR_Message

Esta clase representa un mensaje a enviar o recibido. No tenemos ningún protocolo con lo que el alumno debe definir el suyo, basado en el código de usuario de mensaje, que debe definir como un tipo enumerado. Por ejemplo, define en el fichero de comportamiento de tu agente:

```
public enum   XXX_MsgCode // sustituye XXX por el apodo de tu equipo
{
    AvailableStone,      // Aviso de piedra disponible
    NonAvailableStone,   // Aviso de piedra no disponible, etc
    // etc
}
```

Atributos de la clase:

```
ISSR_Object    Sender,      // Agente que envía el mensaje
ISSR_MsgCode   code,        // Código de mensaje, definido en ISSRDictMsg
                                     // no cambiar: Hello, Asset o Query
int            ucode,        // Código de usuario: A DEFINIR
// Datos opcionales:
Vector3        location,    // Posición en espacio
ISSR_Object    Obj,         // Objeto
float          fvalue,      // Valor de punto flotante
int            ivalue       // Valor entero
```

Servicios de Acción

Sirven para pedir al agente que realice acciones en el juego. Todas las llamadas son no bloqueantes, es decir, piden al juego hacer algo, normalmente toma un tiempo en hacerse, la llamada no hace esperar al agente. Por ello la terminación con éxito de la acción no la conoceremos normalmente hasta más adelante.

De todas formas, es importante comprobar que las llamadas no dan error porque en las circunstancias cambiantes del juego y de acuerdo con las reglas una acción puede no ser posible. Por ejemplo, se pide coger un objeto que ya no está a la vista o no existe, se pide moverse llevando una piedra cuando otro agente del mismo equipo ya se está moviendo con otra piedra...

Ir hacia un lugar en el mundo, parámetro, las coordenadas del lugar a alcanzar `location`.

```
public void acGotoLocation(Vector3 location)
```

Ir hacia un objeto, persiguiéndolo si se mueve, como consecuencia de esta acción si el agente no se para antes de llegar al objeto chocará con él. Parámetro, el objeto a alcanzar `obj_to_reach`:

```
public void acGotoObject( ISSR_Object obj_to_reach)
```

Si el objeto está fuera de la vista dará un error al pedirlo, en su lugar se puede tratar de ir al último lugar dónde se vio el objeto con `acGotoLocation(obj_to_reach.LastLocation)`

Importante: si un agente está agarrando una piedra, cuando intente moverse, se moverá con ella, empujándola. Esta llamada puede fallar por dos razones:

- Hay otra piedra moviéndose empujada por agentes del mismo equipo.
- Se trata de una piedra grande y no hay al menos otro agente que a la vez quiera moverla

Intentar llegar a un objeto para **agarrarlo**, primero persiguiéndolo si se mueve y luego cuando lo alcanza lo intenta agarrar. El parámetro es el objeto que se quiere agarrar `obj_to_grip`. Dará error si el objeto no está visible:

```
public void acGripObject(ISSR_Object obj_to_grip)
```

Parar el agente si está moviéndose:

```
public void acStop()
```

Soltar el objeto que se lleva agarrado:

```
public void acUngrip()
```

Temporizador:

Cada agente tiene un único temporizador que se puede lanzar para que nos avise pasado un tiempo.

Lanzar el temporizador del agente para que salte pasados `delay` segundos, tiempo que representa el plazo a esperar:

```
public void acSetTimer(float delay)
```

Nos avisará de que el plazo ha expirado con el evento `onTimerOut`. Si el temporizador está en marcha ya no hace caso y da un error. Se puede consultar si el temporizador está en marcha con `isTimerRunning()`.

Envío de mensajes:

Las siguientes llamadas también son no bloqueantes, es decir devuelven el control al agente inmediatamente. No hay confirmación explícita de que el mensaje llegue a ningún agente, de hecho, el envío no tiene destinatarios. Se trata de una especie de *broadcast* que se escucha en los alrededores del agente. Como la distancia de escucha de mensajes es mayor que la de vista de objetos, lo que sí es seguro es que, si un agente está a la vista y envía un mensaje, ese agente lo recibirá. Si se quiere asegurar algún patrón de comunicación habrá que construir el protocolo. El alumno definirá los códigos de sus mensajes que se pasarán en el parámetro `usercode` de las llamadas:

Envío de un mensaje con algunos de los argumentos posibles, código, código de usuario, posición, número y entero:

```
public void acSendMsg(ISSRMsgCode code, int usercode,
    Vector3 location=new Vector3 (), float fvalue =0, int ivalue = 0)
```

Otra llamada que además de los parámetros anteriores envía un objeto:

```
public void acSendMsgObj(ISSRMsgCode code, int usercode, ISSR_Object Obj,
    Vector3 location=new Vector3 (), float fvalue =0, int ivalue = 0)
```

Control de errores de los servicios de acción:

```
public bool aiError;
```

Se pone a verdadero si ha habido un error al realizar la última acción, se puede consultar.

```
public aiError_codes aiError_code;
```

En caso de error contiene un código de error para la última llamada.

```
public string acError_msg( aiError_codes code)
```

Da una cadena de caracteres describiendo el error que se corresponde con el código que se le pasa en `code`

Esta función comprueba si hay error e imprime por consola el mensaje de error en ese caso.

```
public bool acCheckError()
```

Un patrón típico de uso puede ser:

```
acXXX(); // Llamada a un servicio de acción
if (acCheckError()) /// Comprueba error e imprime mensaje en consola si lo hay
{
    // Cosas a hacer en caso de error
}
else
{
    // Cosas a hacer en caso de éxito
}
```

Servicios de Información de Objetos

Todo objeto en el juego, sea un agente, una piedra, una meta o una pared se representa por un objeto de la clase `ISSR_Object`. Los objetos de esta clase son parámetros de algunos de los eventos que recibe el agente, cuando ve alguno, choca con uno, etc. El agente necesita los métodos de esta clase para acceder a la información sobre los objetos del juego, son como los recuerdos del agente.

Se utiliza el símil de recuerdos porque la información que puedo conocer de un objeto cuando lo tengo a mano es mayor que cuando ya lo he perdido de vista.

En particular casi todos sus campos o atributos son siempre accesibles:

Nombre del objeto, el mismo que aparece en la jerarquía de la escena en Unity.

```
public string Name
```

El **tipo** de objeto, dado por un enumerado definido en `ISSRObject.cs`

```
public ISSR_Type type
```

Tipos posibles:

```
public enum ISSR_Type
{Undefined,      AgentA, AgentB,
  SmallStone,    BigStone,
  GoalA,   GoalB,
  NorthWall, SouthWall, EastWall, WestWall};
```

Último lugar donde vi al objeto

```
public Vector3 LastLocation;
```

Último momento en que vi al objeto

```
public float TimeStamp; // Rellenado por el juego al comenzar a ver un objeto
```

Entonces para acceder a esos campos en un objeto del juego llamado `Obj` puedo usar

`Obj.Name`, `Obj.type`, `Obj.LastLocation`, `Obj.TimeStamp` (Pero el objeto no debe ser `null`)

Las siguientes llamadas tienen el mismo efecto que las anteriores, pero comprueban que `Obj` no sea `null`:

```
public string oiName(ISSR_Object Obj)
public ISSR_Type oiType(ISSR_Object Obj)
public Vector3 oiLastLocation(ISSR_Object Obj)
public float oiTimeStamp(ISSR_Object Obj)
```

Muy importante:

Para comparar dos objetos de tipo `ISSR_Object` hay que usar el método `Equal()`, no usar los operadores `==` ó `!=`.

- Para ver si `Obj` y `otroObj` son **iguales** en lugar de `Obj == otroObj`, usar `Obj.Equals(otroObj)`
- Para ver si `Obj` y `otroObj` son **distintos** en lugar de `Obj != otroObj`, usar `!Obj.Equals(otroObj)`

Otras comprobaciones simples que pueden hacerse sin que un objeto sea visible son:

Si el objeto es un muro:

```
public bool oiIsAWall(ISSR_Object Obj)
```

Nos devuelve el número de orden dentro del equipo del agente que le pasamos, no permite distinguirlos, este número va de cero al número total de agentes en el equipo -1, este número total de agentes lo determina el número de marcadores de agente que había en la escena al comenzar:

```
public int oiAgentNumber(ISSR_Object Agent)
```

Para ver si un agente es de mi equipo o no

```
public bool oiIsAgentInMyTeam(ISSR_Object Obj) // Verdadero si es de mi equipo
public bool oiIsAgentInOtherTeam(ISSR_Object Obj) // Verdadero si NO es de mi equipo
```

El campo `TimeStamp` puede servir para llevar la cuenta del instante de tiempo en que hicimos una comprobación sobre el objeto. De hecho cuando el evento `onEnterSensingArea()` entrega un objeto asigna este valor al instante de tiempo actual de juego. El alumno puede escribirlo en su código para indicar cuando fue la última vez que comprobó algo, por ejemplo, que una piedra `stone` estaba libre. Podría representar el tiempo que hace de nuestro 'recuerdo'. Para ello hay que asignarle el instante de tiempo actual:

```
stone.TimeStamp = Time.time; // Asigna momento actual al objeto
```

Objetos fuera de la vista

Cuando un objeto está ya fuera de la vista del agente, pero tenemos una referencia a él se nos permite llamar a todas las anteriores funciones, pero todas las siguientes funciones dan error si se llaman usando como parámetro un objeto `Obj` que está fuera de la vista.

Las dos siguientes funciones necesitan que el objeto esté a la vista del agente, en caso contrario dan error:

Posición actual del objeto `Obj`:

```
public Vector3 oiLocation(ISSR_Object Obj)
```

Distancia desde el agente que llama hasta objeto `Obj` en este momento:

```
public float oiDistanceToMe(ISSR_Object Obj)
```

Para evitar este error podemos averiguar si el objeto `Obj` está dentro de la zona visible por el agente mediante la siguiente función:

```
public bool oiSensible(ISSR_Object Obj) // Devuelve verdadero si es objeto es visible
```

Una vez que sabemos que el agente está visible podemos llamar a `oiLocation()` ó `oiDistanceToMe()`.

De todas formas, si el objeto `Obj` no está visible podemos consultar el último lugar dónde lo vimos con:

```
public Vector3 oiLastLocation(ISSR_Object Obj)
```

También podríamos calcular la distancia desde donde está este agente ahora hasta donde estuvo el objeto `Obj` por última vez con (ver funciones de clase `Vector3` más adelante):

```
Distancia = (Obj.LastLocation - oiLocation (Myself)).magnitude;
```

Dirección hacia la que mira un agente `Agent` (su dirección de avance)

```
public Vector3 oiAgentDirection(ISSR_Object Agent)
```

Para saber si un agente u objeto se está moviendo

```
public bool oiMoving(ISSR_Object Obj) // verdadero si se está moviendo
```

Para saber si un agente `Agent` está agarrando un objeto: (devuelve verdadero si el agente está sujetando algo)

```
public bool oiIsAgentGripping(ISSR_Object Agent)
```

Informa del objeto que está agarrando un agente `Agent`, devuelve `null` si ninguno

```
public ISSR_Object oiAgentGrippedObject(ISSR_Object Agent)
```

Informa del número de agentes que está agarrando un objeto `Obj`

```
public int oiGrippingAgents(ISSR_Object Obj)
```

Nota, solo las piedras grandes pueden tener más de un agente sujetándolas.

Recordamos que estas funciones dan error si los agentes u objetos que se pasan como parámetros están fuera de la vista o no definidos (`null`).

Control de errores de los servicios de información de Objeto:

```
public bool oiError;
```

Se pone a verdadero si ha habido un error al realizar la última acción

```
public oiError_codes oiError_code;
```

En caso de error contiene un código de error para la última llamada.

```
public string oiError_msg( oiError_codes code)
```

Da una cadena de caracteres describiendo el error que se corresponde con el código que se le pasa en `code`

Esta función comprueba si hay error e imprime por consola el mensaje de error en ese caso.

```
public bool oiCheckError() // Checks and prints
```

Servicios de Información

Esta serie de funciones permiten al agente conocer otras informaciones sobre el mundo del juego, algunas de estas informaciones son estáticas, otras cambian con el tiempo.

Distancia máxima alrededor de un agente dentro de la que puede **ver** un objeto:

```
public float iSensingRange()
```

Distancia máxima de comunicación, es decir, la distancia máxima alrededor de un agente dentro de la que sus mensajes se escuchan, como es lógico también es la distancia máxima a la que puede estar otro agente para que este agente escuche los mensajes de ese otro agente:

```
public float iCommsRange()
```

Número de agentes que componen un equipo:

```
public int iAgentsPerTeam()
```

Tipo de escenario de juego, **cooperativo o competitivo**. Lo da el valor del enumerado devuelto que puede ser `ISSR_GameScenario.Cooperative` o `ISSR_GameScenario.Competitive`:

```
public ISSR_GameScenario iGameScenarioType()
```

El escenario es cooperativo cuando hay un solo equipo de agentes, se considera competitivo cuando hay dos equipos de agentes.

Lugar en el patio de juegos donde están las **metas**:

```
public Vector3 iMyGoalLocation() // la meta de mi equipo
```

```
public Vector3 iOtherGoalLocation() // la meta del otro equipo en caso de haberlo
```

Puntuación actual (contabilizada con las piedras llevadas a una meta). Las piedras pequeñas dan un punto, las grandes dan tres puntos.

```
public int iMyScore() // Puntuación de mi equipo
```

```
public int iOtherScore() // Puntuación del otro equipo en caso de haberlo
```

Dimensiones del patio de juego:

Ancho del patio de juego según se ve en la cámara: de izquierda a derecha o lo que es lo mismo de Oeste a Este:

```
public float iGameyardXDim()
```

Fondo del patio de juego según se ve en la cámara: de pared cercana a lejana o de Sur a Norte:

```
public float iGameyardZDim()
```

Piedras en movimiento

Número de piedras que puede mover un equipo simultáneamente (las reglas del juego lo definen como 1)

```
public int iMaxMovingStonesAllowed()
```

Número de piedras en movimiento por mi equipo en este momento:

```
public int iMovingStonesInMyTeam()
```

Número de piedras en movimiento por el otro equipo (si lo hay) en este momento:

```
public int iMovingStonesInOtherTeam()
```

Tiempo restante de juego en segundos de juego.

```
public float iRemainingTime()
```

Verdadero si el temporizador del agente está en marcha. Hay un único temporizador por agente que se lanza con `public void acSetTimer(float delay)`, cuando vence produce el evento siguiente:

```
public virtual void onTimerOut(float delay)
```

```
public bool iTimerRunning()
```

Servicios especiales:

Por defecto la percepción (vista) de los agentes y su capacidad de comunicación están habilitadas, estas funciones permiten habilitarlas o deshabilitarlas.

Activa o desactiva la 'vista' de los agentes según el valor del parámetro `enable`:

```
public void EnableSensing(bool enable)
```

Activa o desactiva la comunicación de los agentes según el valor del parámetro `enable`:

```
public void EnableReceiving(bool enable)
```

Resumen de Variables de la clase ISSR_Agent

Estas variables se pueden utilizar en el código:

- Algunas son de **solo lectura**, el juego las actualiza, solo tiene sentido leerlas
- Otras se pueden usar libremente para mantener valores o listas de objetos.

Tener estas variables predefinidas en la clase base nos permite verlas de forma cómoda en la ventana **Inspector**.

Variables de solo lectura

```
// Variables de SOLO LECTURA, actualizadas por el juego.
```

```
public ISSR_Object      Myself;           // Este agente
```

```
public ISSR_Object      GrippedObject;    // Objeto agarrado o null si ninguno
```

```
public List<ISSR_Object> SensableObjects; // Lista de objetos a la vista
```

```
public ISSREventType     current_event;   // Evento actual
```

El valor de esta variable lo fija el juego cuando se produce un evento `onXXX` justo antes de llamar al gestor del evento, que es una función con el mismo nombre: `onXXX()`. Estrictamente hablando no es de solo lectura, le asignamos el valor `ISSREventType.onTickElapsed` en la función `Update()`.

```
public Vector3          dest_location;    // Posición hacia la que quiere moverse el agente
```

```
public int              usr_msg_code;     // Código de último mensaje recibido
```

```
public ISSR_Object      msg_obj;         // Objeto de último mensaje recibido.
```

```
public Vector3          msg_location;     // Posición dentro de último mensaje recibido.
```

Las tres variables se rellenan justo antes de llamar al gestor de eventos `onMsgArrived()`.

Variables de lectura y escritura

Es preferible usar estas variables a nuevas variables creadas en la clase del agente. Estas son visibles en el Inspector, las de la clase derivada no son visibles.

// Variables de lectura y escritura, libres de uso por el programador del agente

```
public ISSRState      current_state; //Estado actual, para máquina de estados
public ISSRState      last_state;    //Último estado, para máquina de estados
public Vector3        focus_location; // Una posición en el patio de juegos
public ISSR_Object     focus_object;  // Un objeto
public ISSR_Object     focus_agent;   // Un agente
public ISSR_Object     colliding_object; // Objeto colisionado
public ISSR_Object     object_just_seen; // Objeto que acabo de ver.
public float          focus_time;     // Un instante de tiempo

public List<ISSR_Object> Stones;       // Una lista de objetos (piedras)
public List<ISSR_Object> Agents;       // Una lista de objetos (agentes)
public List<ISSR_Object> Objects;      // Una lista de objetos
public List<Vector3>    Locations;     // Una lista de posiciones en el espacio

public List<ISSR_Object> Valid_Small_Stones; // Listas de piedras pequeñas
public List<ISSR_Object> Invalid_Small_Stones;
public List<ISSR_Object> Valid_Big_Stones;    // Listas de piedras grandes
public List<ISSR_Object> Invalid_Big_Stones;
public List<Vector3>    Valid_Locations;      // Listas de posiciones
public List<Vector3>    Invalid_Locations;
```

Todas las listas de uso libre están inicializadas a lista vacía al comienzo del juego.

Funciones de Ayuda: Clase ISSRHelp

Este fichero contiene varias funciones agrupadas como métodos de la clase ISSHelp que no son servicios del agente. Pero tienen una utilidad general en el juego, manejo de listas, sobre todo. Algunas de estas funciones tienen como parámetro un ISSR_Agent, en este parámetro hay que poner **this**, para referirse al agente que llama a la función.

Ejemplo de llamada a la primera función:

```
ISSRHelp.CalculateSafeLocation(this, objeto_colisionado)
```

Calcula una posición segura para desplazar el agente dado después de colisionar con el objeto dado. Está en una dirección perpendicular a la de choque tratando de evitar volver a chocar con él. La distancia a recorrer depende del objeto con el que se ha chocado y un factor aleatorio.

```
public static Vector3 CalculateSafeLocation(ISSR_Agent a, ISSR_Object
colliding_object)
```

Devuelve el número de objeto de un tipo dado que hay en una lista de objetos dada.

```
public static int NumberOfObjectsOfTypeInList(List<ISSR_Object> ObjList, ISSR_Type
obj_type)
```

Devuelve la distancia desde el agente dado hasta el objeto dado.

```
public static float Distance_from_object_to_me(ISSR_Agent a, ISSR_Object obj)
```

Devuelve la distancia desde el objeto dado hasta la meta del mismo color que el agente dado.

```
public static float Distance_from_object_to_goal(ISSR_Agent a, ISSR_Object obj)
```

Devuelve el objeto del tipo dado más próximo al agente dado dentro de la lista de objetos dada.

```
public static ISSR_Object GetCloserToMeObjectInList(ISSR_Agent a, List<ISSR_Object>
ObjList, ISSR_Type obj_type)
```

Devuelve el objeto del tipo dado más próximo a la meta del mismo color que el agente dado dentro de la lista de objetos dada.

```
public static ISSR_Object GetCloserToGoalObjectInList(ISSR_Agent a, List<ISSR_Object>
ObjList, ISSR_Type obj_type)
```

Devuelve la localización más próxima al agente dentro de la lista de posiciones dada, si la lista está vacía en la salida `remaining_elements` aparece cero y se ha de ignorar la posición entregada.

```
public static Vector3 GetCloserToMeLocationInList(ISSR_Agent a, List<Vector3> LocList,
out int remaining_elements)
```

Devuelve la localización más próxima la meta del mismo color que el agente dado dentro de la lista de posiciones dada, si la lista está vacía en la salida `remaining_elements` aparece cero y se debe ignorar esa posición.

```
public static Vector3 GetCloserToGoalLocationInList(ISSR_Agent a, List<Vector3>
LocList, out int remaining_elements)
```

Consulta las listas `Valid_Small_Stones` y `Valid_Big_Stones` del agente y devuelve el objeto más próximo al agente de entre los presentes en las dos listas. Si no hay ninguna piedra en esas listas devuelve `null`.

```
public static ISSR_Object Get_next_available_stone_closer_to_me(ISSR_Agent a)
```

Consulta las listas `Valid_Small_Stones` y `Valid_Big_Stones` del agente y devuelve el objeto más próximo a la meta del mismo color que el agente de entre los presentes en las dos listas.

Si no hay ninguna piedra en esas listas devuelve `null`.

```
public static ISSR_Object Get_next_available_stone_closer_to_goal(ISSR_Agent a)
```

Actualiza una lista de objetos con la versión más reciente de los objetos que se le pasan. Si el objeto no está en la lista lo añade. Si está en la lista compara el campo `TimeStamp` del objeto de la lista con el del objeto que se pasa como parámetro, si el parámetro es más reciente (`TimeStamp` mayor) lo actualiza, en caso contrario lo ignora.

```
public static void UpdateObjectList(ISSR_Object obj, List<ISSR_Object>obj_list)
```

Se le pasa una piedra y un booleano `available` que indica si debe pertenecer a la lista de piedras válidas o a la de piedras no válidas. Si la piedra no está en ninguna de las dos listas se añade a la que corresponda según el valor de `available`. Si la piedra está en alguna de las dos listas se consulta el `TimeStamp` de la piedra de la lista y el de la piedra que pasamos como parámetro. Si la información en las listas es más reciente se ignora. Si la información de la piedra parámetro es más reciente se actualiza: copiando el objeto `stone` a la lista que corresponda, puede implicar quitarlo de una lista y meterlo en otra según el valor de `available`.

```
public static void UpdateStoneLists(ISSR_Object stone, bool available,
List<ISSR_Object> ValidList, List<ISSR_Object> InvalidList)
```

Se le pasa una línea o camino determinado por dos puntos: `From` y `To`. Se le pasa también un punto de referencia `Point`. Determina un vector de dirección de movimiento perpendicular al camino y que si se sigue desde el punto de referencia lleve a alejarse del mismo. Las coordenadas y de los tres vectores de entrada se ignoran, el vector de salida tiene longitud unidad.

```
public static Vector3 AwayFromPathDirection(Vector3 From, Vector3 To, Vector3 Point)
```

Añade una matriz de posiciones a la lista `Valid_Locations` de forma que si un agente visita todas habrá cubierto todo el patio de juegos con su vista (basta con acercarse a menos de 1.6 unidades de un punto para cubrir su zona). Vacía también la lista `Invalid_Locations`.

```
public static void SetupScoutingLocations(ISSR_Agent a)
```

Busca en la lista `Valid_Locations` la posición más cercana al agente llamante, si la posición está a menos de 1.6 unidades la quita de la lista `Valid_Locations` y la añade a la lista `Invalid_Locations`. En ese caso devuelve `true`, en caso contrario devuelve `false`.

```
public static bool UpdateVisitedScoutingLocation(ISSR_Agent a)
```

Esta función se llama por el agente durante todo el juego en cualquier estado para ir actualizando su registro de lugares visitados (lista `Invalid_Locations`) y lugares no visitados (lista `Valid_Locations`). Al entrar en estado `Scouting` el agente debe pedir ir a una posición de las no visitadas, para buscar piedras, en ese estado al recibir verdadero como retorno debe pedir ir a otra posición no visitada. Todo ello hasta que se encuentre alguna piedra.

Bibliotecas de Funciones Básicas

Estas son algunas de las bibliotecas de funciones básicas de Unity y C# que puede ser útiles en la construcción de los comportamientos, recordamos que solo se pueden usar estas bibliotecas básicas y los servicios del agente para programar los comportamientos.

List, Listas de objetos

Es una colección de elementos de longitud variable, necesita incluir la siguiente línea para encontrar sus símbolos:

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

Declaración e inicialización de una lista de objetos de un tipo `Type`:

```
List<Type> listName= new List<Type>();
```

Declaración de una lista de objetos de juego, `ISSR_Object`:

```
public List<ISSR_Object> myList; // object list example
```

Ejemplos de métodos de acceso.

```
myList.Add(obj);           // Añadir un elemento obj al final de la lista myList
if (myList.Contains(obj)); // Comprueba si el elemento obj está en la lista myList
myList.Remove(obj);        // Eliminar el elemento dado obj de la lista myList
myList.Count;              // Devuelve el número de elementos de la list myList
i = myList.IndexOf(obj);   // Devuelve el índice del objeto obj en list myList
myList[i] = newObj;        // Cambiar elemento de la posición i de la lista myList por newObj
obj = myList[i];           // Obtener el elemento de la posición i de la lista myList en obj
myList.RemoveAt(i);        // Eliminar el elemento de la posición i de la lista myList
```

Para una discusión general sobre los tipos de colecciones en Unity mira:

http://wiki.unity3d.com/index.php/Choosing_the_right_collection_type

Vector3, para posiciones y direcciones 3D

Para hacer cálculos geométricos puede ser necesario usar el tipo `Vector3`.

Componentes de un `Vector3 vect`

- X: `vect.x` or `vect[0]`
- Y: `vect.y` or `vect[1]`
- Z: `vect.z` or `vect[2]`

Vector unidad constante en las seis direcciones del espacio

- Dirección X: `Vector3.right`, `Vector3.left`
- Dirección Y: `Vector3.up`, `Vector3.down`
- Dirección Z: `Vector3.forward`, `Vector3.back`

Vector nulo: `Vector3.zero`

- **Longitud** de un vector: `vect.magnitude`
- **Normalizar** vector, es decir, hacerlo de longitud unidad: `vect.Normalize()`
- Devuelve el **vector unitario** de uno dado sin cambiarlo: `vect.normalized`

Operaciones con vectores (métodos estáticos de la clase):

Dados dos vectores `Vector3 v1, v2;`

- **Ángulo entre dos vectores**, desde el vector `v1` al vector `v2`: `Vector3.Angle(v1, v2)`
- **Producto vectorial**: vector que es perpendicular a ambos `v1` y `v2`, y cuya longitud es proporcional a las longitudes de `v1` y `v2` y al seno (`sine()`) del ángulo entre ellos: `Vector3.Cross(v1,v2)`. El orden de la operación importa.
- **Producto escalar**: es un escalar proporcional a las longitudes de `v1` y `v2` al coseno (`cosine()`) del ángulo que forman. `Vector3.Dot(v1, v2)` el orden de la operación no importa y representa la longitud de la proyección de un vector sobre el otro

El tipo `Vector3` es un tipo básico, no es necesario `new` para crear un elemento. La interfaz completa puede encontrarse en:

<https://docs.unity3d.com/ScriptReference/Vector3.html>

`Random`, generación de números aleatorios

La introducción de comportamientos parcialmente aleatorios puede ser buena para evitar que aparezcan ciclos repetitivos en el comportamiento de los agentes.

```
Random.value // public static float value;
```

Devuelve un valor aleatorio entre cero y uno con distribución uniforme, tipo `float`

```
Random.Range(from, to) // public static float Range(float min, float max);
```

Devuelve un valor aleatorio entre `from` y `to` de tipo `float`

La interfaz completa está en:


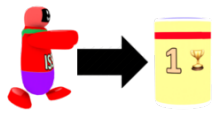
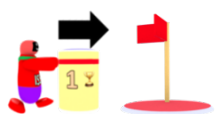



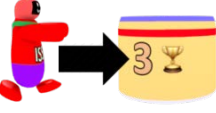

<https://docs.unity3d.com/ScriptReference/Random.html>



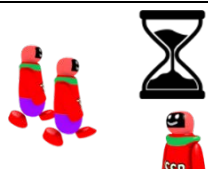
`Mathf`, matemáticas simples con números `float`

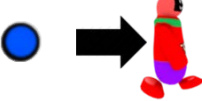







Referencia en:

<https://docs.unity3d.com/ScriptReference/Mathf.html>

Lista de Estados propuestos

Icono	Nombre ISSRState.XXX	Corresponde a:
 IDLE	Idle	Inactivo
 GTgSS	GointToGripSmallStone	Yendo a agarrar/coger una piedra pequeña
 GTGSS	GoingToGoalWithSmallStone	Yendo a la meta con piedra pequeña (agarrada)
 AO	AvoidingObstacle	Evitando obstáculo: Moviéndose a una posición segura para rodear obstáculo tras una colisión
 W4NSM	WaitforNoStonesMoving	Esperando con piedra (pequeña) agarrada a que no haya ningún agente de mi equipo moviendo una piedra para empezar a mover ésta
 SAC	SleepingAfterCollisions	Durmiendo después de colisiones: tras de una serie continua de colisiones el agente se pone a la espera un tiempo aleatorio
 GTgBS	GoingToGripBigStone	Yendo a agarrar/coger una piedra grande
 W4H2MBS	WaitingForHelpToMoveBigStone	Esperando ayuda para mover piedra grande, un agente está agarrado a una piedra grande a la espera de que otro agente colabore con él para moverla

Icono	Nombre ISSRState.XXX	Corresponde a:
 W4NSMBS	WaitforNoStonesMovingBigStone	Esperando con piedra grande agarrada junto con al menos otro agente a que no haya ningún agente de mi equipo moviendo una piedra para empezar a mover ésta.
 GTGBS	GoingToGoalWithBigStone	Yendo a la meta con piedra grande agarrada, moviéndose en colaboración con otros agentes.
 SCOUT	Scouting	Explorando el Patio de Juego buscando piedras no conocidas por ejemplo.
 END	End	Fin , el agente no tiene nada más que hacer
 GTMP	GoingToMeetingPoint	Yendo a punto de encuentro . Por ejemplo para que los agentes puedan intercambiar información u organizarse.
 S4P	SearchingForPartners	Buscando compañeros , para: comunicación, ... ¿?
 W4P	WaitingForPartners	Esperando a compañeros , por ejemplo, en un punto de encuentro hasta que estén todos para comunicarse.
 GOOTT	GettingOutOfTheWay	Quitándose de en medio , puede servir para que un agente se aparte del camino que llevan otros hacia la meta por ejemplo con una piedra grande, para no estorbarles.

Icono	Nombre ISSRState.XXX	Corresponde a:
 GA	GoingAway	Alejándose. Yendo en dirección contraria a un determinado lugar por ejemplo para evitar estorbar o colisionar.
 BLACK	Black	Negro. Estado definible por el programador, se identifica por su color.
 BLUE	Blue	Azul. Estado definible por el programador, se identifica por su color.
 RED	Red	Rojo. Estado definible por el programador, se identifica por su color.
 GREEN	Green	Verde. Estado definible por el programador, se identifica por su color.
 YELLOW	Yellow	Amarillo. Estado definible por el programador, se identifica por su color.
 WHITE	White	Blanco. Estado definible por el programador, se identifica por su color.
 ERROR	Error	Error. Puede utilizarse para indicar cuando el agente ha tenido un error, también se activa cuando el estado del agente está fuera de rango