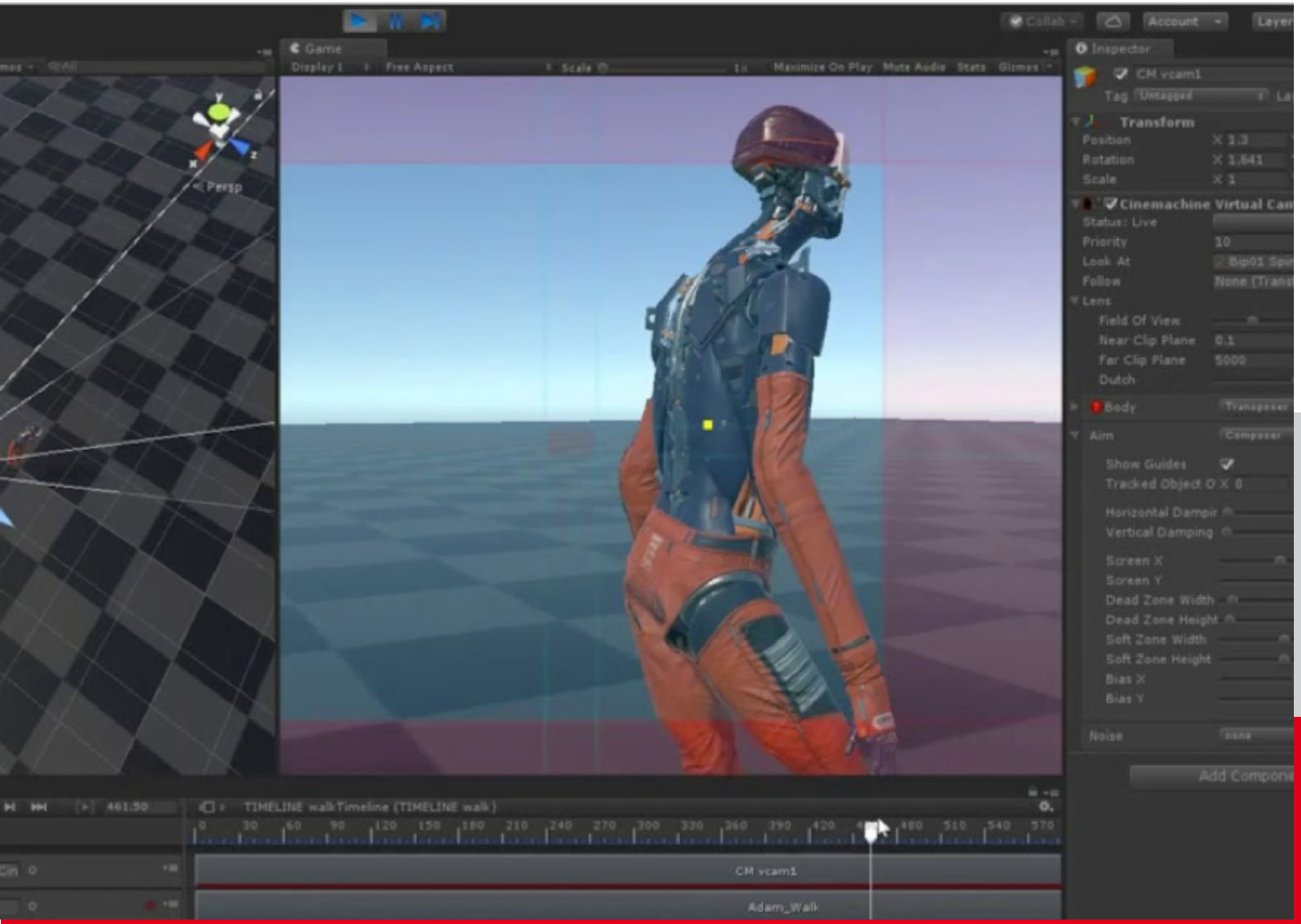
Álvaro Holguera
CIFP José Luis Garci
(Madrid)



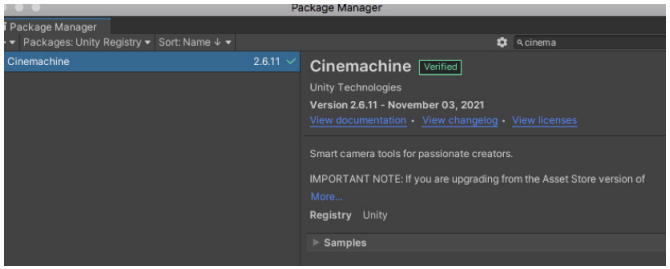

# CINEMACHINE

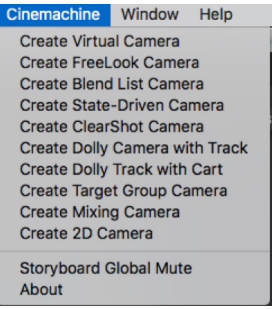
Disponemos de un paquete específico para controlar la cámara y creación de cinemáticas: el Cinemachine.

# PAQUETE CINEMACHINE

## Instalación

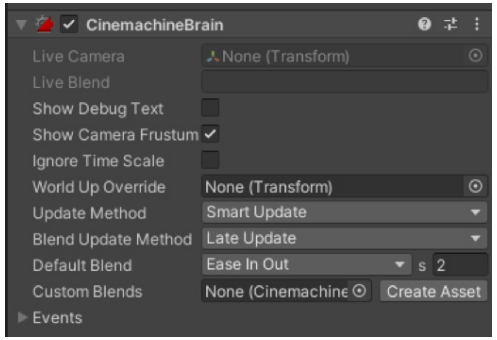
Debemos instalar el paquete a través del Window > Package Manager:




Una vez instalado, tendremos un menú específico para Cinemachine, así como componentes asociados:

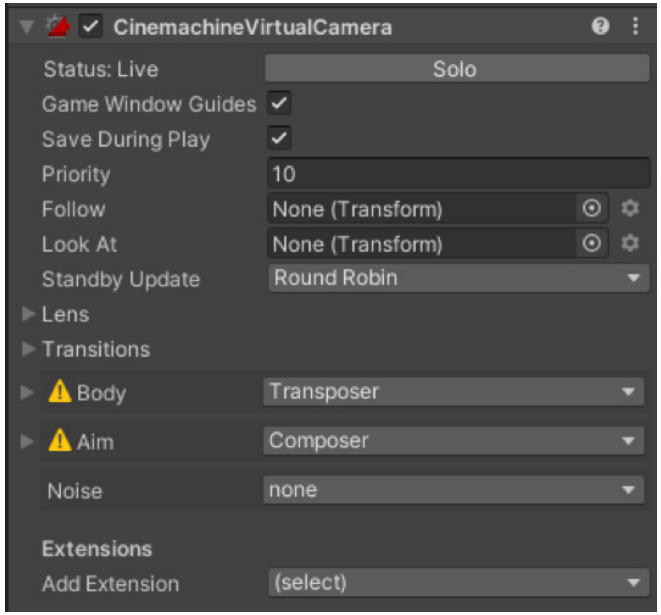
Lo primero que debemos hacer, es añadir a nuestra cámara un componente "CinemachineBrain", que tomará el control de ella a través de CineMachine.




## Cámaras virtuales

Veremos que el apartado Live Camera está sin asignar. Para ello, Crearemos una Cámara Virtual que se asignará automáticamente al apartado Live Camera de nuestra cámara principal, y ésta aparecerá con el icono de CineMachine:


Si seleccionamos la cámara virtual recién añadida, podemos configurar sus opciones en el Inspector:




Entre las opciones, podemos asignar a qué objeto mirar y/o seguir, a través de sus componentes Transform.

Si arrastramos a nuestro personaje a los campos de "Follow" y/o Look At, automáticamente la cámara que contenta esta cámara virtual como "Live Camera" en su componente "CinemachineBrain" comenzará a seguir a nuestro personaje.

Podemos crear tantas cámaras virtuales como queramos, y luego ir alternando entre unas y otras a través de su parámetro "priority" (la que tenga mayor será la elegida), o directamente desactivando unas y activando otras.

### Librería Cinemachine

Si queremos acceder a los atributos de las cámaras virtuales, deberemos incluir la librería.

Podremos crear instancias de todos los elementos del paquete, incluyendo las cámaras, y modificar mediante código sus parámetros.

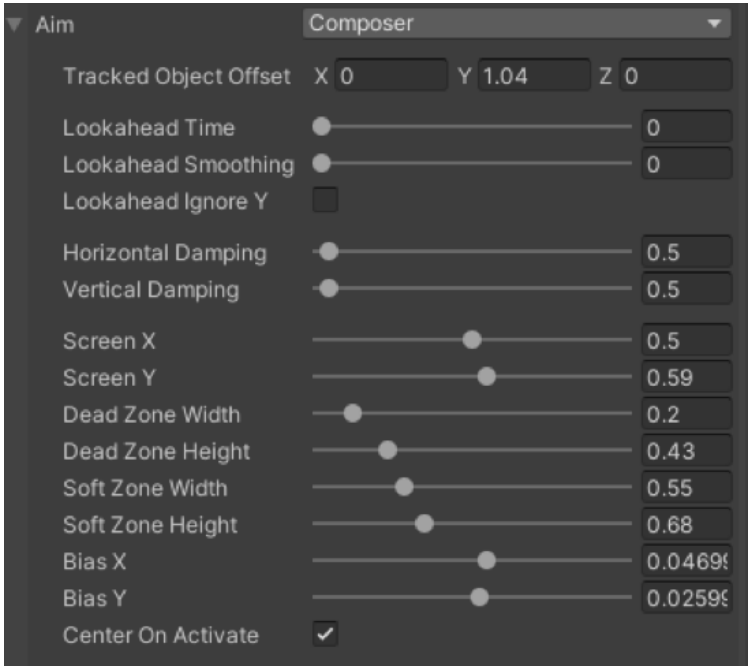
En el siguiente ejemplo, creamos una variable que contendrá una VirtualCamera, y tras asignarla en Unity le asignamos una prioridad distinta a la que tiene:

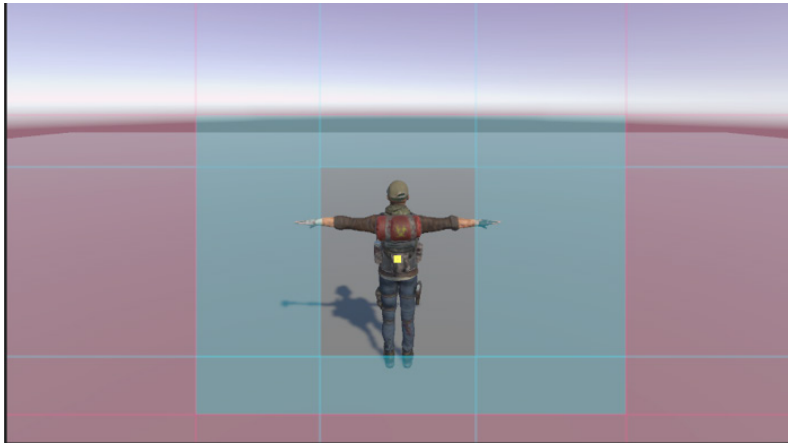
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Cinemachine;

public class CameraController : MonoBehaviour
{
    [SerializeField] CinemachineVirtualCamera virtualCamera;
    // Start is called before the first frame update
    void Start()
    {
        virtualCamera.Priority = 22;
    }
}
```

# SEGUIMIENTO CON LA CÁMARA

En el apartado "Aim" de nuestra cámara virtual podemos ajustar un gran número de parámetros:

- **Tracked Object Offset**: para que la cámara se sitúe por encima o de forma lateral
- **Lookahead Time / Smoothing**: que la cámara se adelante al movimiento del personaje
- **Damping**: tiempo que tarda la cámara en seguir al personaje
- **Screen X / Y**: rotación en X ó en Y, para que la cámara "desvíe" su mirada respecto al personaje
- Ajustar las **zonas de seguridad**: si seleccionamos nuestra cámara virtual, aparecerán las zonas de seguimiento:
    1. Dead zone: si el personaje se mueve dentro de esta zona, la cámara no rota. En la imagen del juego lo identificaremos porque aparece sin colorear.
    2. Soft Zone: la cámara rotará gradualmente a medida que el personaje entra dentro de esta zona. Corresponde a la zona coloreada de azul.
- **Bias / Center:** permite descentrar las zonas anteriores
- **Center On Activate:** cuando la cámara se activa (puede haber varias cámaras en el escenario) el personaje se centra automáticamente.

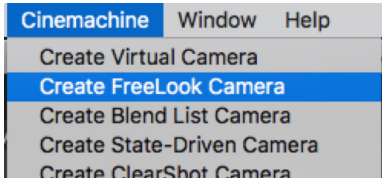


# FREE LOOK CAMERA

## Freelook camera

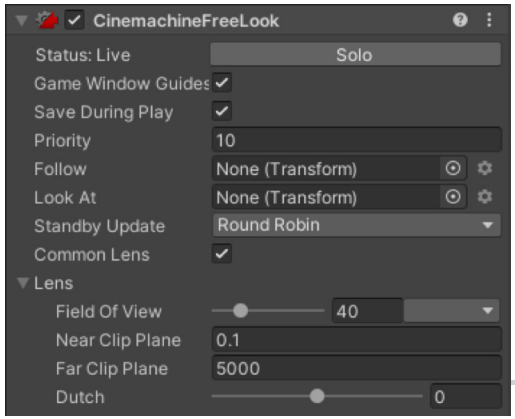
Para poder controlar una cámara en toorno al personaje debemos crear una FreeLook Camera.



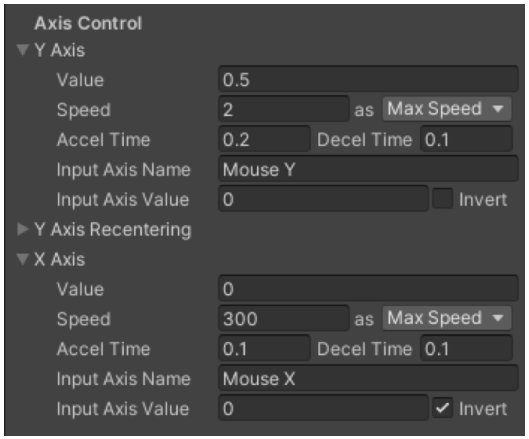

Si tenemos otra cámara virtual, deberemos desactivarla, o por lo menos indicar a nuestra cámara real que use esta, para lo cual es suficiente con aumentar su prioridad.

Igual que en el anterior ejemplo, debemos indicar a la cámara virutal a qué GameObject debe mirar y/o seguir, así como el angular de la lente.

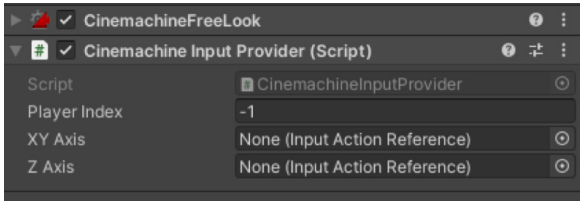
También podemos ajustar parámetros de distancia al objetivo, velocidad de los ejes, aceleración, etc.



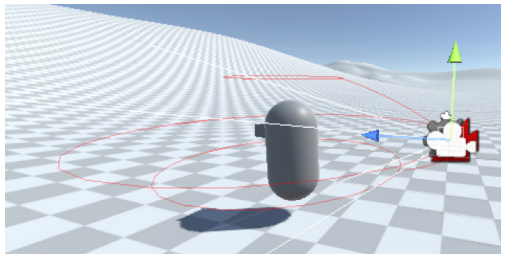

En el apartado de "Input Axis Name" escribiremos cómo se llama el eje que controla el movimiento de pivote (por defecto, "Mouse Y" y "Mouse X"):



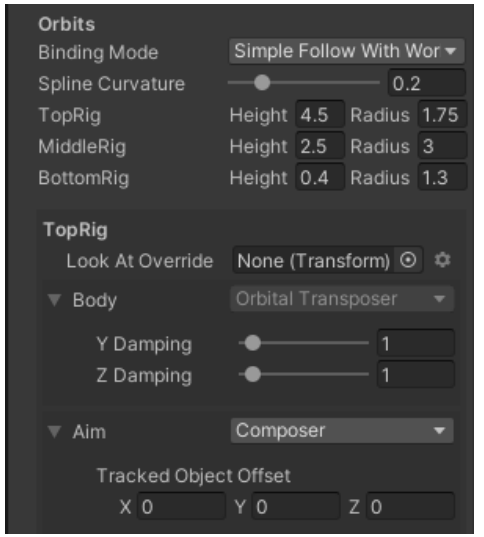

IMPORTANTE: Para que funcione con el nuevo paquete de Input System, debemos reemplazar ("override") los ejes, añadiendo un componente a la Cámara Virtual de tipo Free Look, llamado Cinemachine Input Provider, que es un script en el que podremos ajustar los ejes con entradas de nuestro InputSystem:




La órbita en torno al personaje lo hace basándose en 3 "rigs". Si seleccionamos la cámara y activamos los gizmos los veremos:



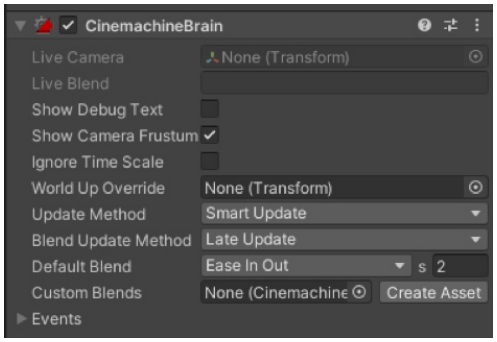
Podemos ajustar la anchura y la altura de esos 3 rigs, así como modificar a lo que miran o su desviación, para controlar el pivote en cada uno de ellos:



# ALTERNAR ENTRE CÁMARAS

Es habitual cambiar entre dos posiciones de cámara, por ejemplo, para apuntar en un Shooter en tercera persona o para crear una cinemática.

El componente CinemachineBrain que tiene nuestra cámara principal, tiene la opción de configurar la forma




de pasar entre dos cámaras ("Default Blend"), y su duración:

Tendremos que crear dos cámaras (o más) entre las que alternar. Lo normal es pasar de una FreeLook Camera a una Virtual Camera en posición de apuntando.

Para pasar de una cámara a otra podemos hacerlo aumentando su prioridad, como hemos visto antes, aunque es suficiente con desactivar una y activar la otra, ya que CinemachineBrain lo detectará automáticamente. Previamente las habremos obtenido mediante variables de tipo "GameObject":

```
freeCamera.SetActive(false);
aimCamera.SetActive(true);
```

Podemos comprobar si una cámara está activa preguntando si ese GameObject está activo en la escena, es decir, en el panel de Jerarquía (con un "!" antes si queremos comprobar si NO está activa):

```
if(!aimCamera.
activeInHierarchy) { }
```

Podemos llamar a una función que activa/desactiva las cámaras al pulsar un botón. Al desactivar una cámara, el componente CineMachineBrain pasa automáticamente a la otra activa

Vamos a hacer algo que nos vendrá bien luego para cualquier shooter en terera persona. Ahora que tienes al personaje 3D añadido a tu escena, crea una cámara que pivote alrededor de él, usando para ello el joystick derecho del Gamepad -o bien el ratón-.

A continuación, crea una cámara virtual sobre su hombro. Ajusta la posición, el offset y la lente para que quede a tu gusto.

Añade un script que al comenzar el juego desactive la cámara virtual, quedando solo la FreeLook, pero que al pulsar un botón, el que quieras, active la virtual. Si tiene una prioridad mayor, automáticamente verás como la cámara de pivote se desplaza hacia la virtual a la velocidad que le hayas configurado.

Acabas de crear una función de "apuntar".

Álvaro Holguera
CIFP José Luis Garci
(Madrid)


# CHARACTER CONTROLLER

Veamos esta herramienta de Unity específica para controlar personajes 3D

# CHARACTER CONTROLLER

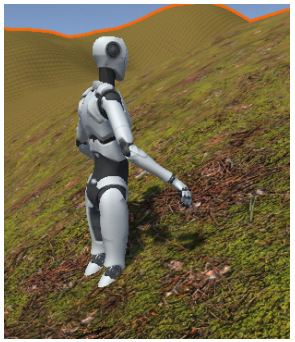
**Cómo se comporta nuestro personaje 3D en nuestro juego depende de muchas variables: movimientos, gravedad, saltos, movimiento en el aire, pendientes, bordillos, interacciones...**

El componente CharacterController, habitual en juegos en tercera persona, permite generar de forma fácil movimientos y colisiones en nuestro personaje, sin necesidad de recurrir a un RigidBody, lo que permite un mayor control sobre su comportamiento, ya que a menudo no nos interesa recurrir a las físicas por sus limitaciones.

Este componente, tiene varias características propias que veremos en detalle:

- No le afectan las fuerzas, por lo que el personaje solo se moverá cuando llamemos a los métodos SimpleMove y Move.
- Si queremos que actúe sobre otros RigidBodies, debemos usar el método OnControllerColliderHit() mediante código.
- En posición de reposo no tiene gravedad, se la tendremos que añadir nosotros.
- Añade un atributo realmente útil, que nos dice en todo momento si el personaje está tocando suelo a través de una booleana: isGrounded.

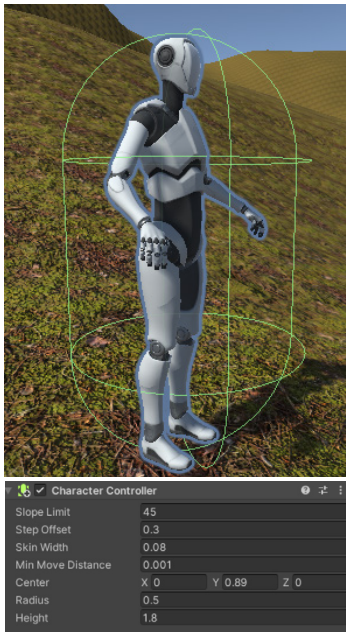
IMPORTANTE: hasta ahora el personaje de Ethan se movía porque sus animaciones incluían desplazamiento, pero esto no es aconsejable ya que son desplazamientos en coordenadas absolutas, sin tener en cuenta el entorno, como el suelo. Si lo hacemos moverse por un terreno, veremos que se hunde en la tierra



## Añadiendo el componente

Al aplicar el componente, automáticamente genera una capsula en torno a nuestro personaje.

Mediante los parámetros de "Center", "Radius" y "Height" podemos ajustarlo al tamaño y a la forma de nuestro personaje.




Los demás parámetros que podemos controlar son:

- **Slope Limit**: ángulo máximo de pendiente que podrá subir nuestro personaje.
- **Step Offset**: tamaño máximo de los escalones que podremos subir (depende de las escalas que usemos, si una unidad en nuestro proyecto equivale a 1 metro, un escalón de 0,3 será de 30 cmts.
- **Skin width**: cuando dos colliders contactan, pueden penetrar uno en el otro su ancho de piel. Un valor alto evita temblores (jitter) pero un valor pequeño puede producir que se quede atascado. Lo recomendable es un 10% del radio
- **Min Move Distance**: si el personaje se mueve por debajo de ese margen (al darle un impulso pequeño), no se moverá

Una vez añadido a nuestro personaje, podemos acceder a él mediante código y controlarlo mediante sus propiedades y métodos públicos, algunos de los cuales veremos a continuación.

# USANDO EL CHARACTER CONTROLLER

## Mover al personaje

Utilizamos el método SimpleMove, al que le pasaremos un Vector3. La longitud de ese vector determinará la velocidad.

El vector que indica la dirección también indica la velocidad, por eso lo multiplicamos tanto por el valor del eje (si es 0 no se mueve, y si es negativo se mueve hacia detrás) como por una variable de velocidad fija.

Para generar la dirección del vector, usamos el método "TransformDirection", submétodo del método Transform, que convierte la dirección local del personaje en una dirección global.

El método SimpleMove es independiente del frame rate del juego, por eso NO es necesario multiplicarlo por Time.deltaTime (a diferencia del método Move)

Adicionalmente, añadimos un opción de rotación usando el submétodo Rotate.

Declararemos dos variables que determinarán la velocidad de desplazamiento y de giro:

```
float speed = 3.0f;
```

```
float rotateSpeed = 3.0f;
```

A continuación, declaramos la variable

```
CharacterController cc;
```

Y oobtenemos el componente Character Controller en el método Start;

```
cc = GetComponent<CharacterController>();
```

Finalmente, creamos un método al que llamaremos en el Update para aplicar los desplazamientos y las rotaciones ("move" es el valor del joystick)

```
void CharacterMove()
{
    //Creamos un Vector3 que mira hacia delante de nuestro personaje
    Vector3 forward = transform.TransformDirection(Vector3.forward);

    //Movemos al personaje hacia adelante según el joystick y la velocidad
    cc.SimpleMove(forward * move.y * moveSpeed);

    //Añadimos rotación según el joystick
    transform.Rotate(0, move.x * rotateSpeed, 0);
}
```

## Gravedad

El método empleado anteriormente, SimpleMove, añade gravedad de forma automática al movernos, pero si queremos un mayor control sobre cómo se mueve en el eje Y, por ejemplo para saltos, debemos hacerlo mediante código.

Para ello, acudirémos al método Move, multiplicado por Time.deltaTime, ya que este método es independiente del frame rate.

Para lograrlo, vamos a aplicar una fuerza constante hacia abajo (en el eje Y en negativo), que lógicamente será contrarrestada por el suelo cuando esté tocándolo (usaremos uno de los atributos más útiles del Character Controlles, isGrounded, que devuelve true si nuestro personaje está tocando el suelo, y por tanto si puede saltar o no)

Por otro lado, debemos añadir una variable que determina la velocidad (fuerza) del salto, y otra de gravedad, porque el Character Controller no aplica gravedad por defecto (como el Rigid Body). Para todo eso, crearemos las siguientes variables:

```
float gravity = 9.8f;
```

```
Vector3 fallDirection = new Vector3();
```

La gravedad la aplicaremos en todo momento hacia abajo (eje Y en negativo). Podemos usar el método "fixedUpdate" para recibir mejor las pulsaciones del botón que ejecutará el método Saltar, el cual aplicará un desplazamiento hacia arriba contrarrestando la gravedad:

```
private void FixedUpdate()
{
    Gravedad();
}

void Gravedad()
{
    //El Vector que me empuja hacia abajo
    fallDirection.y -= gravity * Time.deltaTime;
    cc.Move(fallDirection * Time.deltaTime);

    //Si estoy tocando suelo, pero esigo cayendo, lo poongo a 0
    if (cc.isGrounded && fallDirection.y < 0)
        fallDirection.y = 0;

}
```

# USANDO EL CHARACTER CONTROLLER

## Saltar

Ahora que controlamos la gravedad, vamos a crear un salto en el Character Controller (recuerda que no estamos afectados por las físicas).

Lo primero, crear la variable que determinará la altura del salto:

```
float jumpSpeed = 28.0F;
```

**Curiosidad**: podemos calcular exactamente la altura del salto usando la fórmula: v = √h x -2 x g , donde V es la velocidad ascendente que tenemos que aplicar, h la altura y g la gravedad (tendremos que declarar la variable jumpHeight con los metros que salta, y si la gravedad es negativa, el -2 puede ser positivo):

fallDirection.y = Mathf.Sqrt(jumpHeight * – 2f * gravity);

```
private void FixedUpdate()
{
    Gravedad();
}

void Gravedad()
{

    //Hacemos que en todo momento la gravedad se aplique restándose al vector
    fallDirection.y -= gravity * Time.deltaTime;
    cc.Move(fallDirection * Time.deltaTime);

    //Si estamos tocando el suelo y la gravedad tira hacia abajo, la ponemos a cero
    if (cc.isGrounded && fallDirection.y < 0)
        fallDirection.y = 0;

}
//Método al que llamaremos cuando pulsemos el botón de salto (1 vez)
void Saltar()
{
    //Comprobamos que está tocando el suelo para evitar doble salto
    if (cc.isGrounded)
    {
        print("saltando");
        fallDirection.y = jumpSpeed;
    }
}
```

PARA NOTA: Lo ideal sería combinar este código con el de desplazamiento para que solo exista un Vector3 de desplazamiento, y así evitar conflictos (si es necesario, evitando que cuando no estemos tocando el suelo podamos girar o desplazarnos).

## Colisionar con otros objetos

Algunos inconvenientes del Character Controller es que, si bien añade movimiento sobre el terreno a nuestro personaje, no le permite interactuar físicamente con objetos que tienen RigidBody.

Para ello usaremos el método OnControllerColliderHit, que se activa cada vez que nuestro personaje golpea contra un collider al moverse.

Al igual que otros colisionadores, nos permite obtener los datos del objeto colisionado mediante una variable de tipo ControllerColliderHit:

void OnControllerColliderHit (ControllerColliderHit hit) { …}

A partir de ese momento, podemos actuar sobre el objeto colisionado, como se ve en el siguiente ejemplo:

```
//Método que se activa al chocar con otro objeto
private void OnControllerColliderHit(ControllerColliderHit hit)
{
    //Variable de tipo Rigidbody asociada al objeto colisionado
    Rigidbody body = hit.collider.attachedRigidbody;

    // Si no tiene Rigidbody, o es de tipo kinematic
    if (body == null || body.isKinematic)
        return;
    // Si el objeto está por debajo de nosotros
    if (hit.moveDirection.y < -0.3f)
        return;

    // Calcular la drección de empuje a partir de nuestro movimiento
    // Siempre empujamos hacia los lados, no hacia arriba o abajo
    Vector3 pushDir = new Vector3(hit.moveDirection.x, 0, hit.moveDirection.z);
    // Le aplicamos al rigidbody del objeto el empuje
    body.velocity = pushDir;

    //Idealmente, debemos multiplicarlo por nuestra velocidad:
    //body.velocity = pushDir * speed;

}
```

En el anterior ejemplo, en caso de que el objeto con el que choquemos no tenga RigidBody, o este sea kinemático, paramos la ejecución del método mediante el comando “return”.

## Deslizarse por pendientes

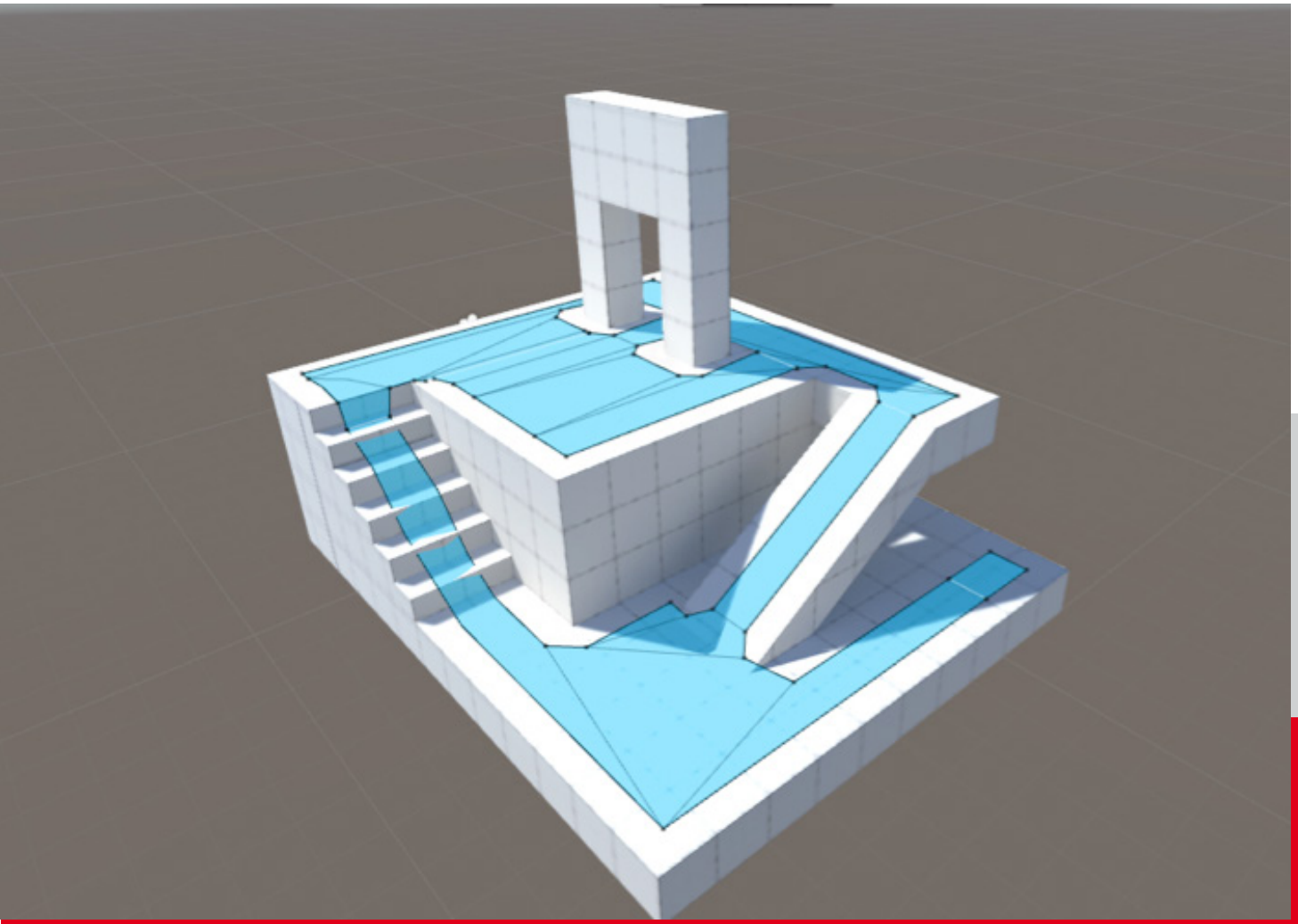
Otro inconveniente es que el Character Controller no hace que nos deslicemos por una pendiente empinada

Para hacerlo, podemos detectar la perpendicular al plano en el que estamos apoyados (su normal)

Esa normal la comparamos con la pendiente máxima que puede subir el personaje

https://www.youtube.com/watch?v=peskzx_5x7A

Álvaro Holguera
CIFP José Luis Garci
(Madrid)



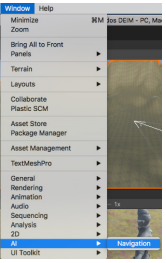
# NAV MESH

Aprendamos ahora cómo añadir enemigos a nuestro juego y que se desplacen solos por el terreno.

# CREANDO UNA NAV MESH

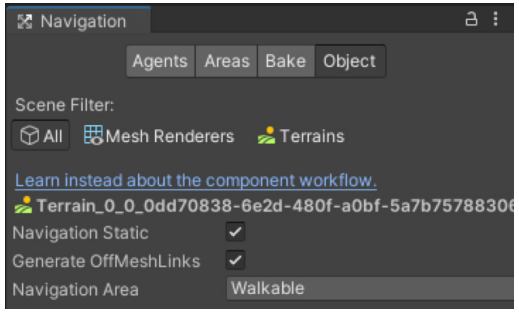
**Esta herramienta nos permite delimitar por qué zona de nuestro terreno se moverá tanto nuestro personaje, como una IA del juego (ó NPC) mediante las herramientas de Navigation y Pathfinding**

Una Nav Mesh es una malla sencilla que se extiende entre objetos más complejos, para poder indicar por dónde pueden moverse determinados elementos del juego.



Para crearla, debemos abrir el panel de Navegación: Window > AI > Navigation

En la ventana que se abre nos mostrará varias pestañas:

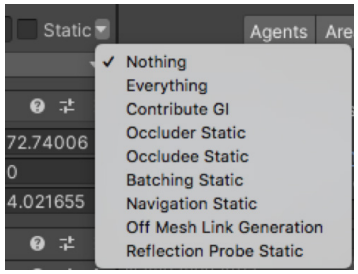



## Object

Debemos indicar qué objetos debe tener en consideración para crear la NavMesh. Lo ideal es seleccionar objetos estáticos como paredes, puertas, etc. Y activar "Navigation Static"

Filtrando los objetos en la ventana de jerarquía, seleccionaremos aquellos que queremos que tome de referencia (se puede hacer directamente en el Inspector de los objetos, en el desplegable de la parte superior derecha, llamado "Static").

Si un objeto que no es estático queremos que se comporte como un obstáculo (por ejemplo una puerta), deberemos añadirle el componente "Nav Mesh Obstacle"




Indicaremos también la capa (Área) a la que pertenece, por defecto hay 3 (Default, Walkeable, Jump).

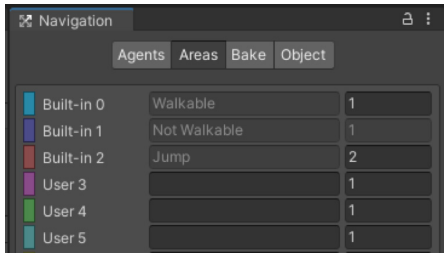
Los OffMeshLinks hace referencia a la posibilidad de saltar entre islas de la zona una vez "bakeada".

## Bake

Define los parámetros de nuestro NavMesh, los cuales veremos en detalle al hablar del "Path Finding". Cuando los tengamos configurados, pulsaremos en "Bake" y se generará la Nav Mesh, la cual podremos ver en el escenario.

## Areas

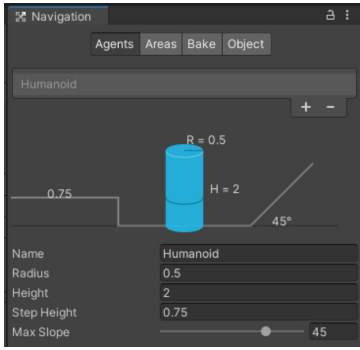
Podemos añadir más "capas", y otorgarle un coste a cada una (cuanto mayor es el coste, menos intentará el NPC ir por esa zona, por ejemplo un camino tendrá menos coste que el barro)




Veremos las tres que vienen por defecto, y casillas para añadir más.

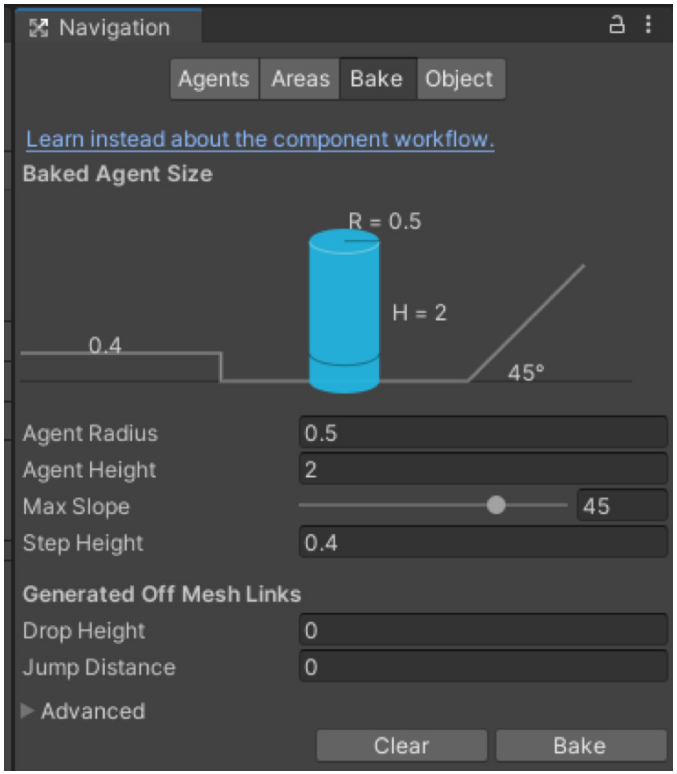
## Agents

Podemos crear diferentes "agentes", con características independientes de cómo se moverán por ese terreno, con parámetros individuales. Posteriormente podremos asignarlos a los personajes.



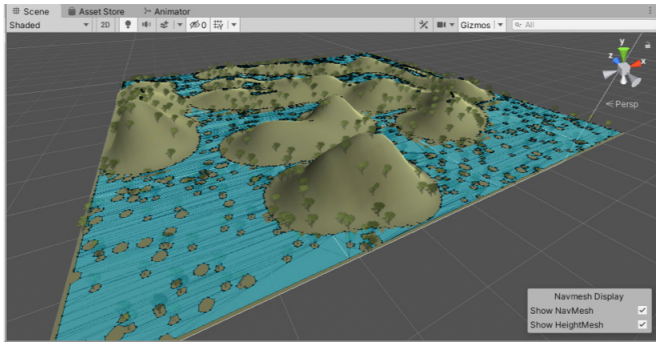
# PATH FINDING

Para poder indicar por dónde se mueve un personaje, debemos realizar un "bakeado" de la NavMesh. Pero antes, debemos configurar los parámetros en la pestaña de "Bake":




- **Agent Radious**: cercanía a las paredes. Evita que vaya chocando con los elementos del entorno, y define el mínimo de ancho que debe tener un camino para que el personaje pase por él
- **Agent Height**: altura que tiene el elemento que se va a mover. Cualquier obstáculo por debajo de eso, un techo, no podrá pasarlo
- **Max Slope**: pendiente en grados que podrá subir nuestro personaje. Por encima de esa inclinación, el terreno será considerado una pared
- **Step Height**: escalones que puede encontrar el agente y que podrá salvar. Tiene que ser menor que el vaor de "Agent Height"o habrá discordancia entre el bakeado y lo que el agente podrá navegar)
- **Generated Off Mesh Links**: permite que el personaje "salte" de una zona a otra del Nav Mesh, evitando huecos. Para ello debemos configurar los objetos. Más info.
- **Opciones avanzadas**, como el área mínima que debe tener una zona, la precisión para crear los bordes, o si creamos una Height Mesh para que sea más preciso

Una vez indicados, le damos a "Bake" para que los aplique y los muestre en el mapa como una capa azul que indica las zonas por las que se puede caminar (debe estar activada la casilla de Show Nav Mesh en el panel de la escena):




Cualquier objeto de nuestra escana puede usar este Nav Mesh que hemos creado mediante el Componente "Nav Mesh Agent"

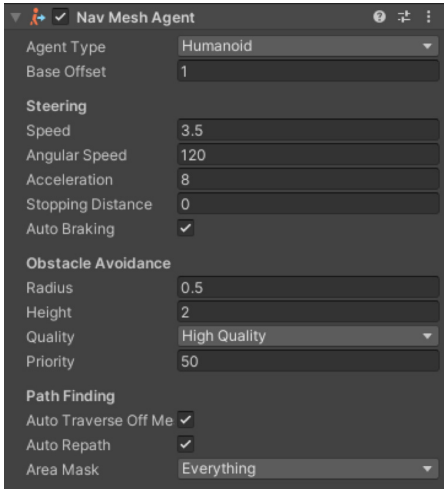
**IMPORTANTE**: actualmente solo se puede "bakear" la superficie para el agente por defecto ("Humaoid"). Si creamos un nuevo agente y lo asignamos a nuestro personaje no funcionará. Para crear varias debemos instalar este paquete

# NAV MESH AGENT

## Este componente nos permitirá crear NPCs que sigan a nuestro personaje a través del Nave Mesh creado

Para que un elemento se mueva a través de un Nav Mesh, debemos añadirle el componente "Nav Mesh Agent".

Aparecerá en nuestro inspector del personaje, donde podremos ajustar cómo queremos que se mueva por ese Nav Mesh.




Estos son algunos de los parámetros

- Agent type: en la pestaña de "Agents" del panel de navigation hemos podido crear tantos como queramos, de forma que aquí podamos asignar esas propiedades. De nuevo, si no instalamos paquetes adicionales solo podemos configurar una agente, el humanoide.
- Base offset: similar al mesh collider, permite subir o bajar la parte de nuestro personaje que se toma de referencia para moverse por el Nav Mesh
- • Steering: los parámetros que determinarán la velocidad, aceleración, velocidad de giro, etc. del personaje. Los podremos ajustar por código, por supuesto.
- •Stoppin Distance permite para el personaje a una distancia determinada del objetivo (útil para que no "penetre" en él), y Auto Braking hace que se detenga cuando alcanza su objetivo.
- •Obstacle Avoidance: parámetros que determina cómo se comporta con los obstáculos, así como su precisión o prioridad (agentes con menor número tienen mayor prioridad)
- Path finding: podemos indicar que auto genere nuevos caminos si se ve atascado.

Estos parámetros se pueden asignar a los agentes (Agent Type), los cuales se pueden crear varios y más tarde asignar a nuestros personajes

Si queremos bakear varias superficies para diferentes "agents", tendremos que instalar herramientas adicionales para IA.

## Ejemplo de NPC

Una vez creado la zona por la que se desplazará, y después de haberle asignado el componente de Nav Mesh Agent a nuestro NPC, podemos indicarle un punto de destino al iniciarse el código mediante el método SetDestination().

IMPORTANTE: Para usar las clases del NavMesh deberemos usar la librería "UnityEngine.AI"

Ese punto de destino debe ser un Vector3, que podemos sacarlo directamente del componente Transform.position de un GameObject de nuestro juego, al que llamaremos "goal":

```
using UnityEngine;
using UnityEngine.AI;

public class EnemyController : MonoBehaviour
{
    //Variable que contendrá la posición del Empty Object, asignada en Unity
    public Transform goal;

    //Creamos una variable de tipo NavMeshAgent que contendrá el componente
    NavMeshAgent agent;

    void Start()
    {
        //Obtenemos el componente en la variable "agent"
        agent = GetComponent<NavMeshAgent>();

        //Mediante "SetDestination", hacemos que se dirija a la posición del "goal"
        agent.SetDestination(goal.position);
    }

}
```

Al lanzar el juego, el personaje se moverá por el terreno indicado, con los parámetros definidos, hacia el punto donde se encuentre.

Si en lugar de ejecutar esta línea en el Start lo hacemos en el método Update, el NPC seguirá en todo momento a nuestro personaje.

# EJERCICIO

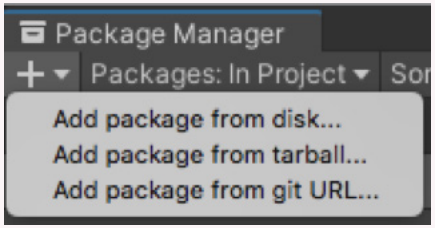
¿Serías capaz de colocar un enemigo en tu terreno y que se dirija hacia ti?

Recuerda que si quieres que cambie de animación, tendrá que ser mediante código, porque ese enemigo no responde a las entradas del mando.

PARA NOTA

Hay una serie de herramientas adicionales para el Nave Mesh, que no están instaladas por defecto, ni siquiera se encuentran como paquete descargable

Para instalarlas, debemos seleccionar en la ventana de Package Manager la opción de Añadir usando nombre o URL de Git, y usar la siguiente dirección: com.unity.ai.navigation





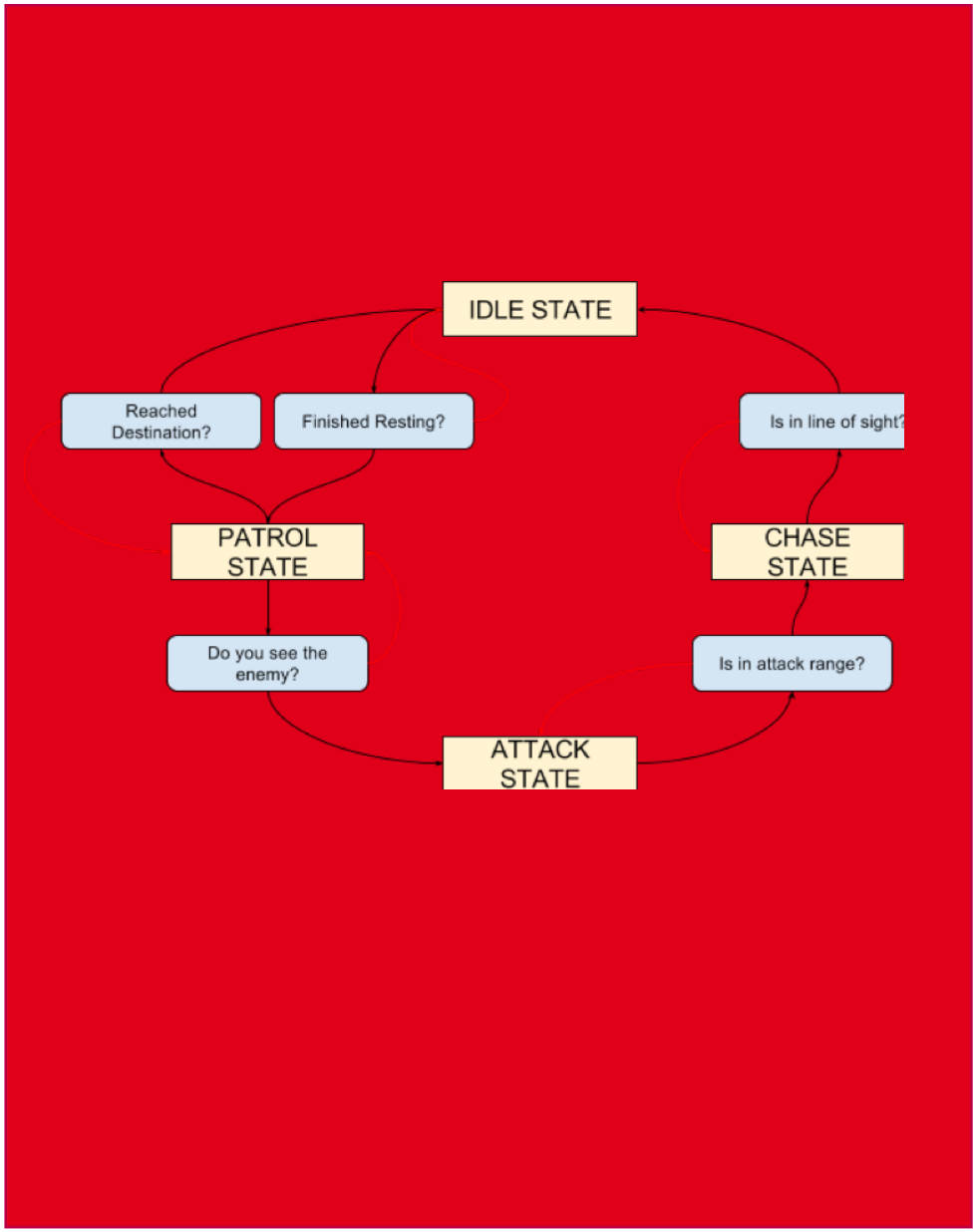
Una vez instalada, contaremos con herramientas adicionales como:

- NavMeshSurface - Use for building and enabling a NavMesh surface for one type of Agent.
- NavMeshModifier - Use for affecting the NavMesh generation of NavMesh area types based on the transform hierarchy.
- NavMeshModifierVolume - Use for affecting the NavMesh generation of NavMesh area types based on volume.
- NavMeshLink - Use for connecting the same or different NavMesh surfaces for one type of Agent.

Más información:

https://docs.unity3d.com/2020.1/Documentation/Manual/NavMesh-BuildingComponents.html

# FINITE STATE MACHINE




El "Autómata Infinito" ó "Máquina de Estado Finito" (FSM por sus siglas en inglés) es un modelo computacional preparado realiza cálculos de forma automática a partir de una entrada para producir una salida.

En videojuegos se utiliza habitualmente para programas IA aplicadas a NPC's.

Ejemplo: un NPC camina entre dos puntos de forma cíclica hasta que ve al jugador, y en ese momento ataca. Si deja de verlo, vuelve a caminar entre los dos puntos:


Para crear un FSM en Unity debemos crear los estados de la animación, que se irán disparando a medida que se producen sucesos, como pueden ser:

- Alcanzar un destino: hacer que se dirija a un punto marcado por un Empty Object con colisionador, que al tocarlo cambia de estado y le hace dirigirse a otro
- Ver al jugador: es útil usar el RayCast para comprobar si el usuario está en su línea de visión, o mucho más efectivo usar un cono de visión mediante el método Angle (podemos saber qué ángulo hay entre el jugador y nosotros y su distancia, y comprobar si son menores que el máximo permitido)
- Escuchar un sonido, y comprobar si es de un volumen concreto

# EJEMPLOS PARA CREAR UN FSM

## Aquí te proponemos algunos ejemplos que te pueden ayudar para crear una IA básica para tus NPCs

En muchos casos, crearemos una variable boolena llamada "detected", y si es true, moveremos al enemigo hacia la posición del personaje ("goal"):

```
If(detected)

        agent.SetDestination(goal.position);
```

### Activar o desactivar su Nav Mesh Agent

Podemos desactivar algunas de las funciones del Game Object, como el componente Nav Mesh Agent o incluso el script que hace que se mueva (en este caso llamado "moveTo"), localizando el GameObject mediante "GameObject.Find". De esta forma el agente no se moverá hasta que, por ejemplo, termine una cuenta atrás o se active una trampa:

```
GameObject.Find("Enemy").
GetComponent<NavMeshAgent>().enabled =
false;

GameObject.Find("Enemy").
GetComponent<moveTo>().enabled = false;
```

### Activarse por proximidad

Una de las formas más básicas es comenzar a moverse si estamos a menos de una distancia determinada del enemigo. Recuerda que el método Vector3.Distance nos puede devolver la distancia entre dos Vectores:

```
float distance = Vector3.Distance(transform.
position, goal.position);

if(distance <= 10f) { detected = true;}
```

### Comprobar si existe línea de visión

Algo más elaborado consiste en detectar cuando el enemigo tiene línea visual con el objetivo. Para ello usamos el método Physics.Raycast, que crea un "rayo virtual" desde un punto hasta otro, con una distancia determinada, y devuelve "true" si ha colisionado con algo (así como los datos de aquello con lo que ha colisionado, por ejemplo su distancia).

```
void VisualDetect()
{

    //Dirección hacia la que mandamos el rayo de visión, hacia adelante
    Vector3 fwd = transform.TransformDirection(Vector3.forward);

    //Variable de tipo Raycast que contendrá los datos del "rayo"
    RaycastHit hit;

    //Si el rayo desde nuestra posición hasta la del objetivo colisiona
    if (Physics.Raycast(transform.position, fwd, out hit))
    {
        //Si el nombre del objeto colisionado es "Payer"
        if (hit.collider.name == "Player" && hit.distance <= 10f)
        {
            //Si no está detectado, lo hacemos mediante un interruptor
            if(!detected)
                detected = true;

        }
        //Podemos dibujar el rayo para ver si está bien orientado
        Debug.DrawRay(transform.position, goal.position, Color.yellow);
    }

}
```

### Crear un cono de visión

Este script permite detectar a un jugador si está dentro de su rango de visión (medido en metros) y en su ángulo de visión (medido en grados)

Si lo está, sea activa la alarma que lo pondrá de color rojo y comenzará a seguirlo con la mirada, hasta que se haya alejado.

Estableceremos los parámetros del cono, tanto la distancia como el ángulo en grados:

```
[SerializeField] float visionRange = 10f;

[SerializeField] float visionConeAngle = 30f;
```

Creamos el método al que llamaremos desde el Udate:

```
void ConoDeVision()
{
    //Creamos un Vector3 con la posición del jugador, y otro entre nosotros y él
    Vector3 playerPosition = goal.position;
    Vector3 vectorToPlayer = playerPosition - transform.position;

    //Distancia hasta el jugador y ángulo que forma nuestra visión frontal con él
    //Si es una IA, podemmos con navMeshAgent, podemos usar remainingDistance
    float distanceToPlayer = Vector3.Distance(transform.position, playerPosition);
    float angleToPlayer = Vector3.Angle(transform.forward, vectorToPlayer);
    //Si está en mi rango y en mi ángulo de visión
    if (distanceToPlayer <= visionRange && angleToPlayer <= visionConeAngle)
    {
        detected = true;
    }
    else
    {
        detected = false;
    }

    if (detected)
    {
        //Miramos en todo momento al jugador
        transform.LookAt(goal);
    }
    else
    {
        //Miramos de nuevo al frente
        transform.LookAt(transform.forward);
    }

}
```

# EJERCICIO

Ya estamos en disposición de crear nuestro primer juego complejo de supervivencia, ambientado en un terreno creado por nosotros, con total control sobre nuestro personaje y con NPC's.

Además de tu personaje, crea unos enemigos y diseña un FSM mediante un Nav Mesh Agent que cambiará su estado de varias formas:

1. Que inicien una "ronda" aleatoria, cambiando su destino cada ciertos segundos.
2. Si detectan al jugador que se dirijan hacia él más rápido
3. Cuando estén a corta distancia, que ataquen

Puedes iniciar diseñando un "dummy" y más adelante cambiarlo por un modelo más complejo con animaciones. En la Asset Store encontrarás algunos realmente interesantes.

Puedes crear un objetivo real en el juego, por ejemplo que nuestro personaje vaya hasta cierta zona segura del escenario, evitando los enemigos.

PARA NOTA

Puedes usar un personaje distinto para tu jugador, también buscándolo en la Asset Store. Incluso, puedes añadir animaciones o elementos adicionales mediante Mixamo.

¿Te atreves a añadir una animación de apuntar con un arma? Puedes hacer que al apuntar, le aparezca el arma en la mano, y, si te animas, hasta instanciar proyectiles.

# ENLACES A VÍDEOS y MATERIALES

Aquí podrás acceder a los vídeos que explican y/o profundizan sobre los conceptos vistos, por si te son de utilidad.

## Terrenos

Creación y modelado:

https://www.youtube.com/watch?v=fE0BYhTNTiU

Añadir texturas y detalles

https://www.youtube.com/watch?v=Ju5v-x8TKmk

## Personajes 3D

Animar un personaje 3D

https://www.youtube.com/watch?v=l6yYw0lP4bk

Cinemachine

https://www.youtube.com/watch?v=Z7xz5zrcdaY

## Character controller y Nav Mesh

https://youtu.be/v4uXH-upw0s

Vídeo que explica cómo desplazarse por pendientes mediante CharacterController

https://www.youtube.com/watch?v=peskzx_5x7A

Paquete para poder incorporar varios agentes en un mismo Nav Mesh

https://github.com/Unity-Technologies/NavMeshComponents

Más información sobre cómo crear un NPC

https://learn.unity.com/tutorial/navigation-basics?language=en#5c7f8528edbc2a002053b49e