First Person Shooter

PID_00244518

Rafael González Fernández
Pierre Bourdin Kreitz

Tiempo mínimo de dedicación recomendado: 4 horas

Universitat Oberta de Catalunya

Índice

Introducción..... 5

Objetivos..... 6

1. «Character Controller»..... 7
1.1. Propiedades 7
1.2. Detalles 8

2. «First Person Controller»..... 9
2.1. Propiedades 9
2.2. Código 12

3. «NavMesh»..... 14
3.1. Creando el escenario 14
3.2. Generando el «NavMesh» 15
3.3. El «NavMesh Agent» 17
3.4. Propiedades del «Bake» 18
3.5. Código 18

4. *IA State Machine*..... 20
4.1. *State Machine*..... 20
4.2. Interfaces 20
4.3. Código 21

5. Proyecto: Un juego *First Person Shooter*..... 26
5.1. El escenario 26
5.2. El personaje principal 27
5.3. El arma 27
5.4. Los enemigos 31
5.5. Soluciones a los retos propuestos 40

Resumen..... 46

Introducción

En este módulo aprenderemos a crear juegos 3D más complejos y empezaremos con un *First Person Shooter* (FPS). De modo que todos los temas que trabajaremos en este módulo estarán centrados en que conozcáis los puntos básicos para crear un juego de este estilo.

Primeramente, aprenderemos las bases para crear un personaje y entender cómo moverlo por el entorno centrándonos en el uso del «Character Controller». Para ello, veremos cómo funciona el «First Person Controller» que nos ofrece Unity y que nos permitirá crear muy fácilmente las bases de un FPS.

Después nos centraremos en cómo crear enemigos y en su comportamiento. Veremos cómo funciona la herramienta «NavMesh», que nos permite moverlos por el escenario, y aprenderemos a crear un sistema de estados para controlar su comportamiento y dotarlos de una mínima inteligencia artificial (IA).

Para dotar al juego de «vida», crearemos una máquina de estados para simular su inteligencia artificial y veremos cómo implementar la funcionalidad deseada en cada uno de sus estados.

Con todo esto, y ya en la parte práctica del módulo, crearemos nuestro primer *First Person Shooter* utilizando todo lo aprendido previamente.

Objetivos

En este módulo didáctico, presentamos al alumno los conocimientos que necesita para alcanzar los siguientes objetivos:

1. Aprender a utilizar las herramientas básicas para crear personajes jugables.
2. Utilizar el «Character Controller» para crear la base de sus movimientos.
3. Entender el *prefab* «FirstPersonController» para juegos tipo FPS.
4. Conocer cómo es la estructura básica de una máquina de estados en una IA sencilla.
5. Utilizar la herramienta «NavMesh» y el «NavMeshAgent» para que los enemigos se muevan de manera autónoma.
6. Crear un juego tipo FPS aplicando los conocimientos previos ya adquiridos en anteriores módulos.

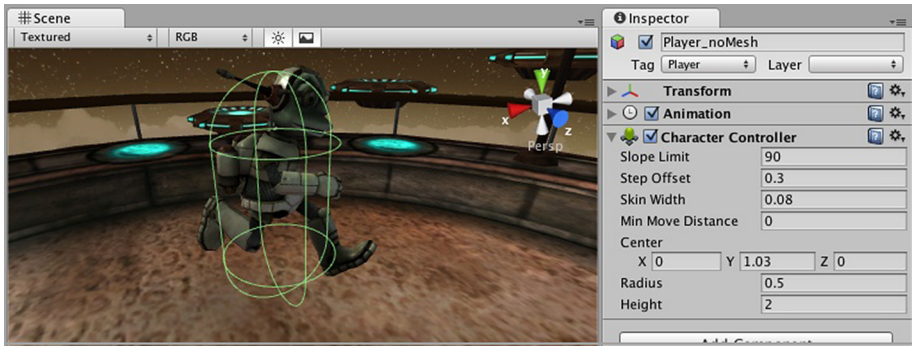
1. «Character Controller»

La base de los controladores de personaje que nos ofrece Unity se fundamenta en el componente «Character Controller». Este es un tipo de «Collider» especial que interactúa con el sistema físico, pero que no hace uso de un «Rigybody».

Esto es así porque normalmente el movimiento de los personajes en los videojuegos no es realista, tiene velocidades mucho más rápidas de lo normal o saltos increíbles. Si utilizásemos un comportamiento físico realista, no tendríamos la jugabilidad a la que estamos acostumbrados.

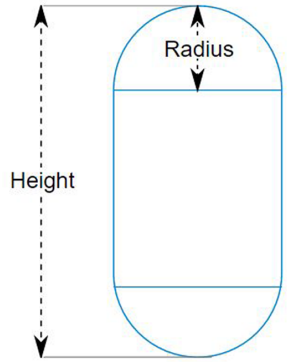
1.1. Propiedades

Este componente lo podríamos resumir como un «Capsulle Collider» que contiene una lógica interna pensada especialmente para ser utilizada en el desplazamiento básico por el escenario del *GameObject* que lo contiene, normalmente el *player* o los enemigos.



«Capsulle Collider»

El «Capsulle Collider» es un tipo específico de «Collider» que está formado por dos semiesferas unidas por un cilindro.



Sus propiedades más importantes son estas:

- «Slope Limit»: Es el ángulo máximo que puede tener una rampa para que el *player* pueda subirla.

- «Step Offset»: Es la altura máxima que puede tener un escalón para que el *player* pueda subirlo.

- «Min Move Distance»: Si intentamos mover el carácter «Controler» con un desplazamiento menor a este valor, el *player* no se moverá.

- «Center/Radius/Height»: Son los valores de configuración del «Capsulle Collider».

Con estas propiedades definimos cómo va a ser este controlador, pero la función clave que hace que podamos mover al *player* es la función «Move».

Veamos un ejemplo de su uso en el que movemos el «Controller» hacia delante y a velocidad constante:

```
1.  public float speed = 6.0f;
2.  public float gravity = 9.8f;
3.  private Vector3 moveDirection = Vector3.zero;
4.  private CharacterController controller;
5.
6.  void Start()
7.  {
8.      controller = GetComponent<CharacterController>();
9.  }
10.
11. void Update()
12. {
13.     if(controller.isGrounded)
14.         moveDirection = Vector3.forward * speed;
15.
16.     moveDirection.y -= gravity * Time.deltaTime;
17.
18.     controller.Move(moveDirection * Time.deltaTime);
19. }
```

1.2. Detalles

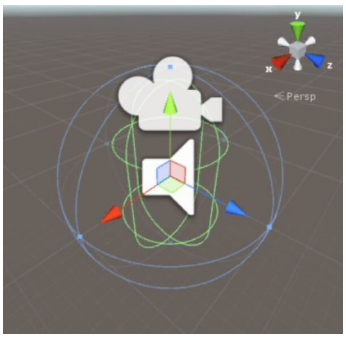
Una de las peculiaridades que tiene el uso de este controlador es que, como no tiene «Rigybody», el «Collider» nunca empujará a otros objetos mientras se mueve. Por ello, si quisiéramos tener esa funcionalidad en nuestro juego deberíamos crearla nosotros mismos utilizando la función «OnControllerColliderHit», tal como se ve en este ejemplo:

```
1. private void OnControllerColliderHit(ControllerColliderHit hit)
2. {
3.     Rigidbody body = hit.collider.attachedRigidbody;
4.
5.     //don't move the rigidbody if the character is on top of it
6.     if (myCharacterController.collisionFlags == CollisionFlags.Below) {
7.         return;
8.     }
9.     if (body == null || body.isKinematic) {
10.            return;
11.        }
12.        body.AddForceAtPosition(m_CharacterController.velocity * 0.1f,
13.                                hit.point,
14.                                ForceMode.Impulse);
15.    }
```

2. «First Person Controller»

El «Character Controller» es la base de la estructura del controlador de personaje. Pero la pieza que hace que todo funcione es el *script* que gestiona ese «Character Controller» y que es capaz de leer los *inputs* del teclado/ratón para transformarlos en movimientos del personaje y en rotaciones de cámara.

En nuestro caso, ese *script* nos lo proporciona Unity y lo podemos encontrar en el *asset package Characters* en «Assets > Import Package > Characters».



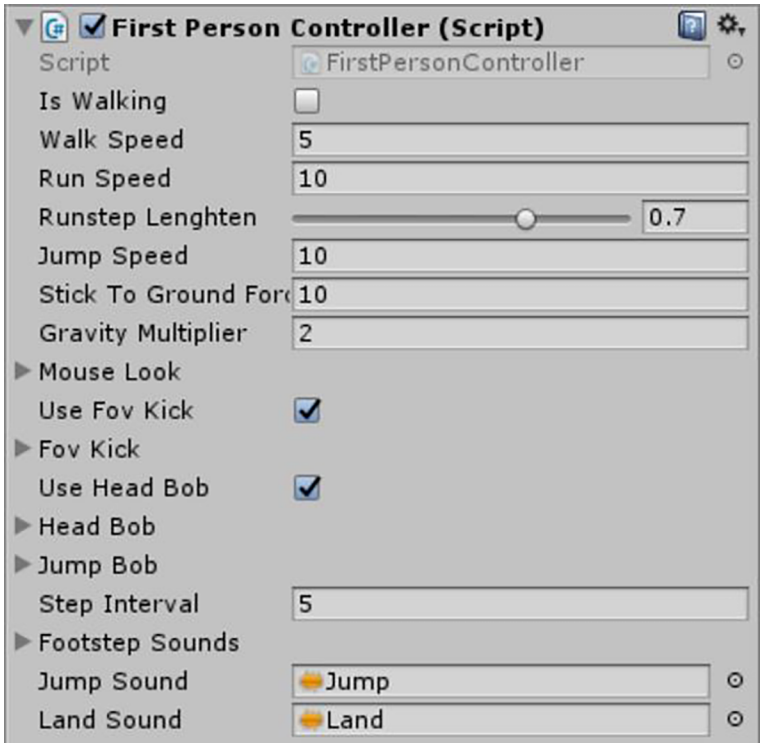
Detalle del «First Person Controller» en la escena

Antes de diseccionar ese *script*, insertemos un *prefab* de «First Person Controller» en la escena. Podéis encontrarlo dentro de «Assets > Standard Assets > Characters > First Person Character > Prefabs > FPSController.prefab».

2.1. Propiedades

Este *prefab* contiene un «Character Controller», el *script* «First Person Controller» que será el que gestione todo su movimiento, un *sound emiter* para crear el sonido de los pasos del *player*, y un hijo con la cámara de juego que acompañará siempre al personaje como si fueran sus ojos.

Veamos pues las propiedades más importantes del «First Person Controller»:



«IsWalking»: Es una variable de consulta que nos indica si el personaje está listo para caminar o para correr. Si apretamos «Shift ingame», se pone a «false» para indicar que el personaje correrá.

Nota

Si quisiéramos ocultar en el Inspector una variable «pública» como «isWalking», ya que no la vamos a modificar en tiempo de edición sino a consultarla en tiempo de juego, tan solo tendríamos que colocar encima o delante de la declaración de esa variable:

[HideInInspector] public bool isWalking;

«WalkSpeed / RunSpeed / JumpSpeed»: Son las velocidades en unidades por segundo a la que se moverá nuestro *player* al caminar, correr o saltar.

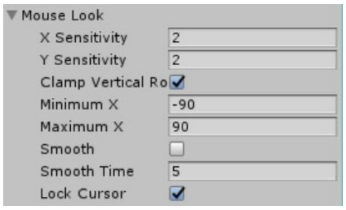
«Stick To Ground Force»: Es la gravedad que se le aplicará a nuestro personaje. En los juegos no se suele utilizar 9,81 sino 10 para facilitar los cálculos, ya que el resultado será muy similar.

«Gravity Multiplier»: Cuando el *player* no está tocando el suelo y para crear una sensación jugable más gratificante, normalmente se suele aumentar la gravedad para que el jugador caiga más rápido. Este valor es el factor de gravedad que le aplicaremos cuando esté en el aire.

«Mouse Look»: Es un grupo de propiedades que en el Inspector se muestra como un grupo, ya que el *script* contiene una variable del tipo «MouseLook»:

```
1.  public MouseLook m_MouseLook;
```

Y esta clase está definida como «Serializable» (con esta etiqueta también podemos mostrar las variables públicas de esta clase en el Inspector) con las siguientes variables:



Detalle de cómo se ve en el Inspector una variable pública de un tipo de clase propio

```
1.  [Serializable]
2.  public class MouseLook
3.  {
4.          public float XSensitivity = 2 f;
5.          public float YSensitivity = 2 f;
6.          public bool clampVerticalRotation = true;
7.          public float MinimumX = -90 F;
8.          public float MaximumX = 90 F;
9.          public bool smooth;
10.         public float smoothTime = 5 f;
11.         public bool lockCursor = true;
12. }
```

«X/Y Sensitivity»: A mayor sensibilidad, mayor será el movimiento de la cámara al mover el ratón.

«clampVerticalRotation»: Lo normal es tener este *bool* en «true», ya que hará que, si miramos hacia arriba o hacia abajo, la cámara no de la vuelta y nos quedemos mirando boca abajo en nuestra nuca. Los ángulos máximos y mínimos de estos ángulos de *clampeo* serían «MinimumX» y «MaximumX».

«Smooth»: Si está en «true», la cámara se moverá con un *lerp* que nos dará un desplazamiento suave que tendrá un pequeño retraso respecto a cualquier movimiento que hagamos de la cámara. La cantidad de *lerp* y el tiempo de

blend dependerán de la variable «smoothTime». En los juegos es muy habitual tener los desplazamientos o las transiciones con un pequeño *lerp* para suavizar los movimientos y hacerlos más realistas y agradables.

«lockCursor»: Si esta variable está en «true», el cursor será bloqueado y ocultado, por lo que no se saldrá de la pantalla, aunque estemos jugando. Como curiosidad, este es el código que se ejecuta si esta variable está en «true»:

```
1.  private void InternalLockUpdate()
2.  {
3.      if (Input.GetKeyUp(KeyCode.Escape))
4.      {
5.          m_cursorIsLocked = false;
6.      }
7.      else if (Input.GetMouseButtonUp(0))
8.      {
9.          m_cursorIsLocked = true;
10.     }
11.
12.     if (m_cursorIsLocked)
13.     {
14.         Cursor.lockState = CursorLockMode.Locked;
15.         Cursor.visible = false;
16.     }
17.     else if (!m_cursorIsLocked)
18.     {
19.         Cursor.lockState = CursorLockMode.None;
20.         Cursor.visible = true;
21.     }
22. }
```

Nota

El *enum* «CursorLockMode» puede ser del tipo:

- «None»: No cambia el comportamiento por defecto.
- «Locked»: Bloquea el cursor en el centro de la pantalla de juego.
- «Confined»: El cursor es libre, pero nunca puede salir de la ventana de juego.

Nota

Para conseguir que a una variable como esta, que simplemente es un *float*, podamos decirle que esté comprendida entre un valor máximo y mínimo y poder setearla con una barra de desplazamiento, tendremos que colocar antes de la declaración de la variable:

[Range(0f, 1f)] public float runstepLenghten;

Como veis, solo se volverá a mostrar y se liberará cuando el *player* toque «escape», y se volverá a ocultar en cuanto cliquemos de nuevo en la zona de juego. Esto es ideal para trabajar y jugar en el editor de Unity.

También tenemos propiedades más decorativas, como son, por ejemplo:

- «Runstep Lenghten»: Es la distancia en unidades en que tardará en sonar un paso nuevo o en mover la cabeza mientras nos desplacemos.

- «Use Fov Kick»: Si está en «true», cada vez que empecemos a caminar o a correr el «Field of view» de la cámara se ampliará dinámicamente para dar más sensación de velocidad.

- «Head Bob»: Esta parte simula el movimiento de la cabeza mientras caminamos y hace que la cámara vaya basculando de izquierda a derecha, simulando los pasos que demos.

- «Jump Bob»: Lo mismo que lo anterior, pero en vertical y para, cuando caemos al suelo, simular que encogemos un poco las rodillas.

- «Footsteps / Jump / Land Sounds»: Los sonidos que se lanzarán en cada uno de estos casos.

2.2. Código

Una vez vistas las características principales del *script* y sabiendo cómo desplazar un «Character Controller», veamos un poco cómo funciona internamente su código para rotar la cámara.

```
1. public void Init(Transform character, Transform camera)
2. {
3.     m_CharacterTargetRot = character.localRotation;
4.     m_CameraTargetRot = camera.localRotation;
5. }
```

Este código se encuentra dentro de la clase «MouseLook», que es llamado por el *script* principal «First Person Controller».

Lo primero que hace en el «Start» es llamar a la función «Init» para guardar las rotaciones iniciales tanto del personaje como de la cámara. Posteriormente, a cada *frame*, llamará a la función «LookRotation» para recalcular estas rotaciones.

```
1.  public void LookRotation(Transform character, Transform camera)
2.  {
3.      float yRot = CrossPlatformInputManager.GetAxis("Mouse X")
4.                   * XSensitivity;
5.      float xRot = CrossPlatformInputManager.GetAxis("Mouse Y")
6.                   * YSensitivity;
7.
8.      m_CharacterTargetRot *= Quaternion.Euler(0f,    yRot, 0f);
9.      m_CameraTargetRot    *= Quaternion.Euler(-xRot, 0f,   0f);
10.
11.     if (clampVerticalRotation)
12.       m_CameraTargetRot = ClampRotationAroundXAxis(m_CameraTargetRot);
13.
14.     if (smooth)
15.     {
16.       character.localRotation = Quaternion.Slerp(
17.                                         character.localRotation,
18.                                         m_CharacterTargetRot,
19.                                         smoothTime * Time.deltaTime);
20.       camera.localRotation = Quaternion.Slerp(
21.                                         camera.localRotation,
22.                                         m_CameraTargetRot,
23.                                         smoothTime * Time.deltaTime);
24.     }
25.     else
26.     {
27.         character.localRotation = m_CharacterTargetRot;
28.         camera.localRotation = m_CameraTargetRot;
29.     }
30.
31.     UpdateCursorLock();
32. }
```

Nota

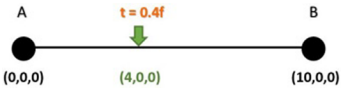
Cuando queremos sumar un desplazamiento a una posición, sumamos un vector a otro. Pero en el caso de las rotaciones, cuando queremos sumar una rotación a otra hemos de multiplicar los «Quaternions».

Aquí lo que hace antes de nada es guardar el *input* del movimiento del ratón con la sensibilidad ya aplicada. Después, a las rotaciones de la cámara y del personaje les aplica la rotación correspondiente según el movimiento del ratón.

La línea 12 se encarga de limitar el ángulo de rotación vertical (el del eje *X*) para que no sobrepase los ángulos que antes hemos definido como máximos y mínimos («MinimumX» y «MaximumX»).

«Lerp» y «SLerp»

La función «Lerp» hace una interpolación lineal entre dos puntos. Por cada valor *t* que le pasemos entre 0 y 1 generará un punto intermedio entre ambos de manera proporcional.



«SLerp» hace lo mismo, pero aplicándole una transición esférica. Eso se traduce en que tanto el arranque como la llegada al destino tendrán una pequeña aceleración y desaceleración.

Por último, aplica estas rotaciones al *transform* del *player* y de la cámara. En el caso de que hayamos elegido una transición «Smooth», hará una transición suave entre las rotaciones actuales y las objetivo.

Y estas son las bases del movimiento de un personaje por el escenario y las de su cámara en el caso de un *First Person Shooter*. ¡Con esto aprendido ya podremos construir bastantes juegos!

3. «NavMesh»

Cambiando un poco de tercio, vamos a ver ahora algo un poco más abstracto que prácticamente no requiere trabajo de código alguno, pero que nos permitirá ahorrarnos mucho trabajo en nuestras inteligencias artificiales a la hora de calcular los desplazamientos de, por ejemplo, enemigos o grupos de animales.

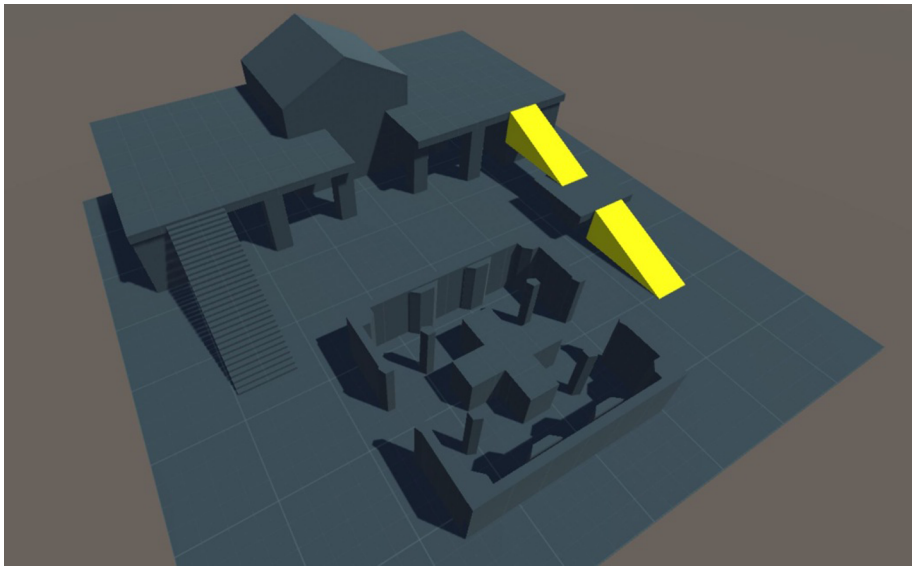
Unity viene con un sistema para calcular cuál es el camino más corto entre dos puntos evitando los obstáculos que se encuentran por el camino, que se llama «NavMesh».

Para empezar a ver cómo funciona, tendremos que activar la ventana con la que trabajaremos y configurarla a través del menú «Window > Navigation». Ahora, al lado de la pestaña del Inspector, tendremos una nueva llamada «Navigation».

3.1. Creando el escenario

Antes de meternos en faena, lo primero que haremos será crear un escenario por el que desplazar a nuestras inteligencias artificiales. Para ello, podéis utilizar cubos básicos de Unity y montar vuestra escena, o podéis utilizar los elementos que Unity nos ofrece para prototipar este tipo de cosas añadiendo el *asset package Prototyping* en «Assets > Import Package > Prototyping».

Este escenario tan molón es el que utilizaremos como ejemplo:

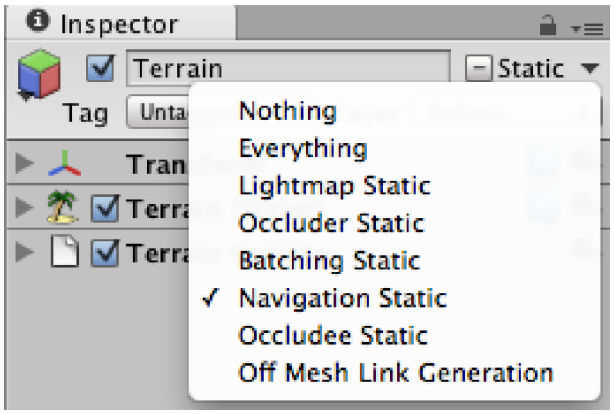


Podéis encontrar los *prefabs* de las piezas dentro de «Assets > Standard Assets > Prototyping > Prefabs».

3.2. Generando el «NavMesh»

Una vez ya hemos creado el escenario, generaremos ahora la información de navegación necesaria para que el «NavMesh» sepa qué partes del escenario son transitables y cuáles no lo son.

El primer paso será seleccionar toda la geometría del escenario y marcar esos *GameObjects* como estáticos. En este particular han de ser del tipo «Navigation Static», pero en este ejemplo podremos atajar haciéndolos estáticos en todos los casos clicando en la casilla «Static», arriba a la derecha en el Inspector.



Al hacer esto, informamos a Unity de que estos *GameObjects* no se moverán nunca durante la partida, lo que es bastante útil a la hora de optimizar, ya que Unity lo tendrá en cuenta internamente de cara a hacer los cálculos de iluminación o pintado de *meshes* y optimizaciones cuando se ejecute el juego. Y, por supuesto, también es un requisito imprescindible para generar la información de navegación del «NavMesh».

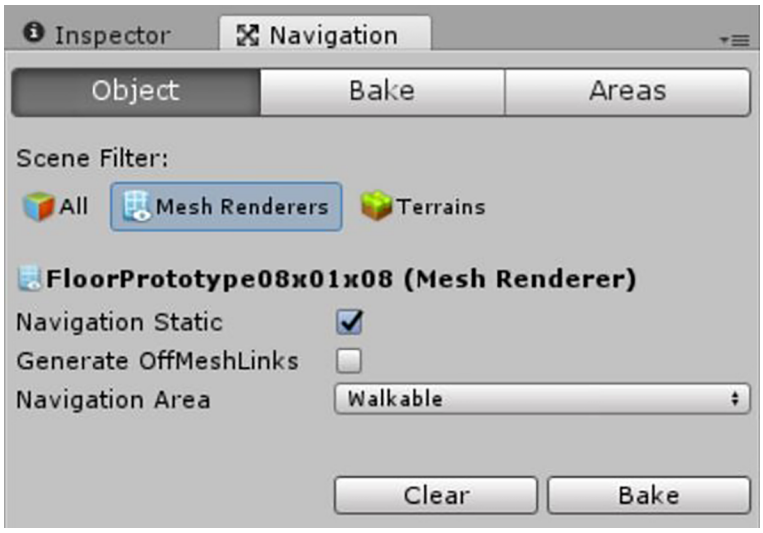
Así pues, ahora cuando seleccionemos cualquiera de esos *GameObject* que hemos hecho estáticos y vayamos a la pestaña «Navigation» veremos la siguiente información:

- «Scene Filter»: Esto nos muestra en la jerarquía de la escena solamente los tipos de elementos que hayamos seleccionado aquí. Destacan principalmente los «MeshRenderers» porque el «NavMesh» solo se genera a partir de *meshes* y no a través de «Colliders» como podría uno pensar inicialmente.

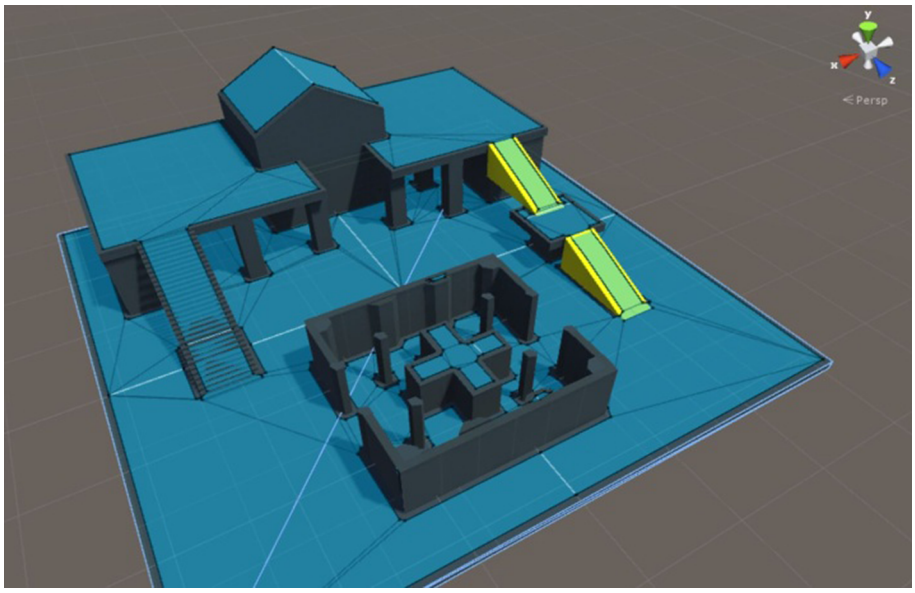
- «Navigation Static»: Si activamos este *check*, marca la geometría seleccionada como «Navigation Static» tal como hemos hecho en el paso inicial, por lo que simplemente marcándolo, nos habríamos ahorrado un paso.

- «Navigation Area»: Esta es la parte clave, aquí seleccionaremos si queremos que sobre esta geometría (en la parte más alta de su eje *Y*) se pueda caminar o no.

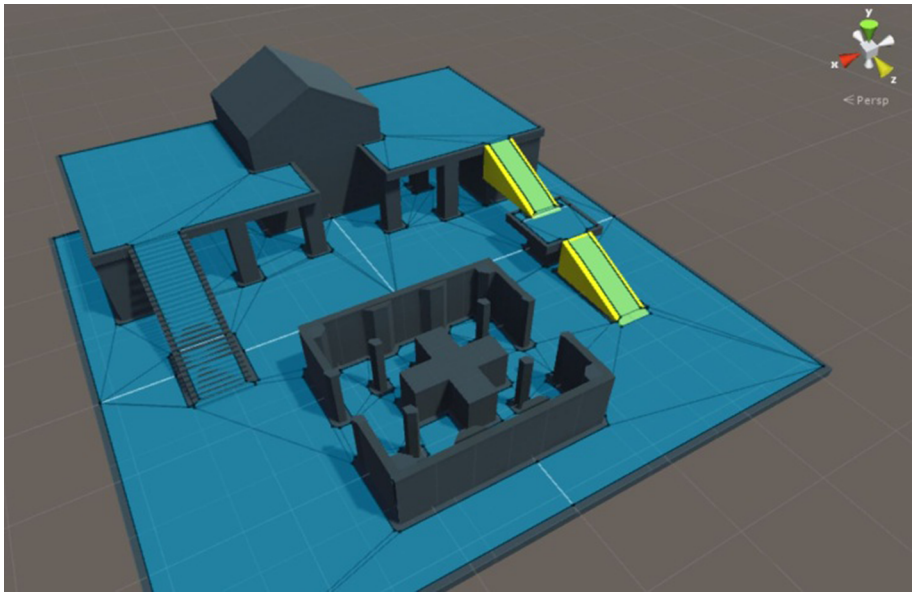
- «Bake»: Cuando le demos a «Bake», Unity recalcula la información del «NavMesh» en su totalidad, machacando como fuera esta previamente.



Así que, si seleccionamos toda la geometría y le decimos que sea *walkeable*, nos quedará un «NavMesh» como el de la imagen (donde toda la parte azul es por donde se podrá transitar):



Como se puede apreciar, parece que podríamos caminar tanto en el tejado de la casa como en la parte superior de la cruz del jardín o en algunos muros, por lo que deberíamos marcar todas las zonas que sabemos que no serán transitables como «Not Walkable». Una vez marcadas, volvemos a darle a «Bake» y parece que ahora todo tiene más sentido:

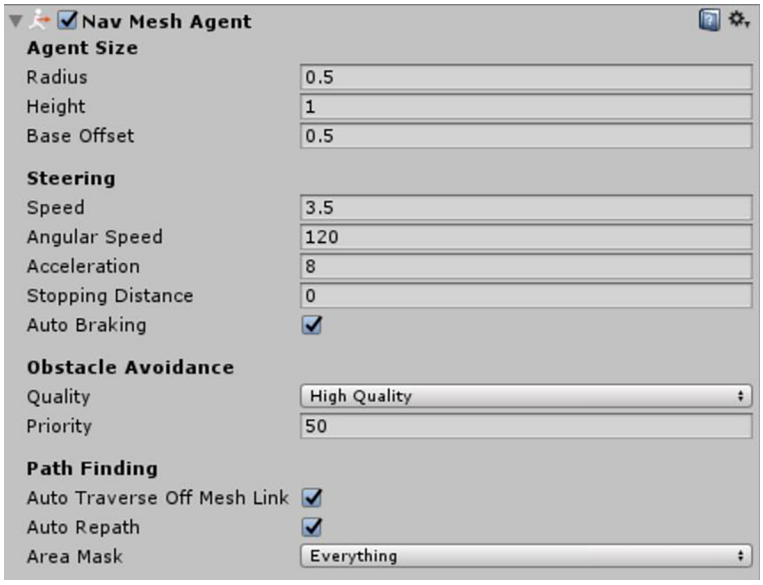


Nota

Toda la información sobre qué zonas son transitables y cuáles no se guarda en una carpeta que se llama como la escena que lo contiene, y crea en su interior un archivo llamado «NavMesh» con un icono como este:

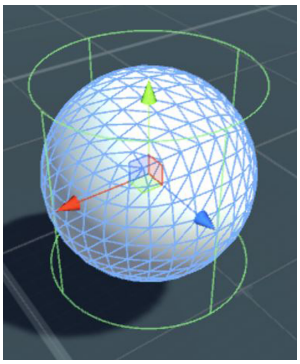
3.3. El «NavMesh Agent»

La estructura «NavMesh» por sí sola no tendría mucho sentido si no va acompañada de un «NavMesh Agent», que es el componente que añadiremos a nuestros *GameObjects* de la escena para que puedan moverse a través del «NavMesh».



Nota

En este ejemplo hemos añadido el componente «NavMeshAgent» a esta esfera, creando así una especie de cilindro a su alrededor que se encargará de detectar las colisiones con los límites del «NavMesh».



Sus principales propiedades son las siguientes:

- «Radius / Height»: Marcan la anchura y la altura del cilindro que envolverá a nuestro *GameObject*.
- «Base Offset»: Indica la altura a la que estará el centro de este cilindro.
- «Speed / Angular Speed»: Son las velocidades a las que se moverá y rotará nuestro objeto al moverse.
- «Acceleration»: El tiempo que le costará alcanzar su velocidad máxima de movimiento.
- «Stopping Distance»: Indica a qué distancia del objetivo se parará el agente antes del llegar al objetivo.
- «Auto Braking»: Si está activado, el agente disminuirá su velocidad poco a poco antes de llegar a su objetivo.

Nota

Podéis encontrar más información sobre el resto de las propiedades del «NavMesh Agent» en la documentación de Unity:
https://docs.unity3d.com/Manual/class-NavMeshAgent.html.

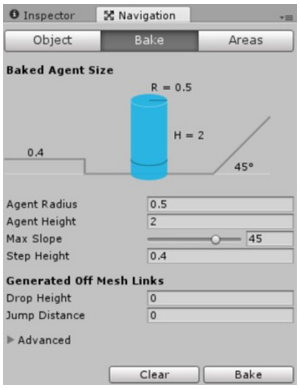
3.4. Propiedades del «Bake»

Aunque ya hemos visto cómo crear un «NavMesh» y qué es el «NavMesh Agent», aún nos faltaría ver con detalle algunas de las propiedades para la generación del terreno caminable.

Cuando hacemos «Bake», estamos generando las zonas transitables suponiendo que utilizaremos un agente con unas características por defecto. Si quisiéramos cambiarlas, tendríamos que decir al «Bake» qué altura queremos que tenga nuestro agente, su anchura, la altura mínima de un escalón para que pueda subirlo o el ángulo de inclinación de las rampas por las que podrá subir.

Nota

Propiedades del agente por defecto para hacer el «Bake» del escenario.



3.5. Código

Ahora que ya sabemos cómo crear un escenario transitable con sus agentes, veamos cómo podemos decirles a esos agentes que se desplacen automáticamente de un punto a otro.

```
1.  public class MoveToClickPoint: MonoBehaviour
2.  {
3.     NavMeshAgent agent;
4.
5.     void Start()
6.     {
7.        agent = GetComponent<NavMeshAgent>();
8.     }
9.
10.    void Update()
11.    {
12.       if (Input.GetMouseButtonDown(0))
13.       {
14.
15. RaycastHit  hit;
16. Ray camRay = Camera.main.ScreenPointToRay(Input.mousePosition);
17.
18.           if (Physics.Raycast(camRay, out hit, 100))
19.           {
20.              agent.destination = hit.point;
21.           }
22.        }
23.     }
24. }
```

Nota

Si queréis que detecte la colisión, recordad que las *meshes* del escenario deberán tener «Colliders»; si no, no detectará colisión alguna.

Si le añadimos este pequeño *script* al *GameObject* que contenga el «Nav Mesh Agent», cuando arranque el juego lo primero que hará será guardarse la referencia a ese «Nav Mesh Agent», y luego estará esperando a que cuando hagamos clic con el ratón lanzar un rayo desde ese punto de la pantalla hacia el escenario y ver si ha colisionado con algo. En ese caso, le dirá al agente que ese es su nuevo destino.

Como veis, la parte clave de ese *script* es «agent.destination», donde iremos *seteando* hacia dónde queremos que vayan nuestros agentes.

4. *IA State Machine*

En esta sección vamos a ver cómo crear una máquina de estados finita con la que crear la inteligencia artificial de nuestros enemigos o de cualquier NPC (*non playable character*) de nuestros juegos.

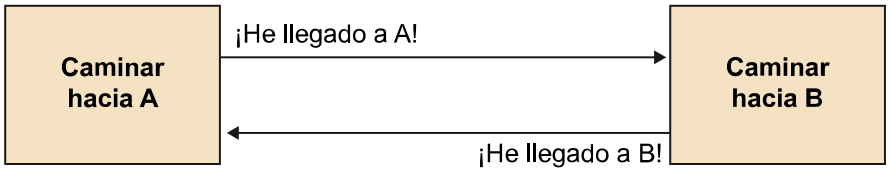
Para ello, utilizaremos interfaces y crearemos tantas clases como estados necesitemos.

Veremos solamente la base de la estructura creando un ejemplo simple, y luego ya en la sección práctica crearemos una inteligencia artificial un poco más compleja.

4.1. *State Machine*

Podríamos definir una máquina de estados como un conjunto de estados relacionados entre sí en los que cada estado puede tener, o no, una funcionalidad determinada.

Imaginemos un personaje que va caminando del punto A al punto B, y que cuando llega al B, vuelve al A, y así sucesivamente. Esto podríamos representarlo visualmente mediante el siguiente grafo:



Esto ya es en sí una máquina de estados donde hay dos estados (verde y azul) y las transiciones entre ellos. Así pues, el personaje estará en el estado «caminar hacia B» hasta que llegue a B, cuando pasará al estado «caminar hacia A».

4.2. Interfaces

Una clase de tipo «Interface» contiene las definiciones de un grupo de funcionalidades que una clase o una estructura pueden implementar heredando de ella.

Podríamos decir que una «Interface» es una clase que se utilizará como «contrato» o «plantilla» y en la que se definirán ciertas funciones que todas las clases futuras que hereden de ella deberán obligatoriamente implementar.

Esto se utiliza para facilitar la programación y para asegurarnos de que todas las clases que hagamos que hereden de esta tengan una implementación de todas sus funciones, con lo que evitamos dejarnos nada.

Ejemplo

Este código tan sencillo crearía una interfaz «vehículo» con una función «acelerar» y otra «frenar». De ella heredan la clase «coche» y la clase «camión», y las dos deberán implementar esas funciones, pero cada una tendrá una funcionalidad distinta en cada una de ellas.

```
1. public interface IVehiculo
2. {
3.     void Acelerar();
4.     void Frenar();
5. }
```

```
1.  public class Coche: IVehiculo
2.  {
3.      public float velocidad_actual = 0;
4.      public float velocidad = 120;
5.
6.      public void Acelerar()
7.      {
8.          velocidad_actual += velocidad * Time.deltaTime;
9.      }
10.     public void Frenar()
11.     {
12.         velocidad_actual - velocidad * Time.deltaTime;
13.     }
14. }
15.
16. public class Camion: IVehiculo
17. {
18.     public float velocidad_actual = 0;
19.     public float velocidad = 90;
20.
21.     public void Acelerar()
22.     {
23.         velocidad_actual += 0.5f * velocidad * Time.deltaTime;
24.     }
25.     public void Frenar()
26.     {
27.         velocidad_actual -= 0.5f * velocidad * Time.deltaTime;
28.     }
29. }
```

Funciona como la herencia clásica, pero añadiendo la obligación de que los hijos tengan implementadas las funciones del padre. Aunque no se las vaya a llamar nunca o estén vacías, han de estar ahí.

4.3. Código

Usando interfaces, vamos a crear el código que implemente a un enemigo quieto vigilando una zona y que cuando entremos dentro de su zona de acción se ponga a dispararnos.

La estructura será la siguiente:

- Tendremos un cubo verde en la escena, al que llamaremos «enemy», que contendrá un «Sphere Collider» que actúe como «Trigger», que sea diez veces más grande que el cubo.

- Crearemos una interfaz llamada «IEnemyState», que será la base para los dos estados que crearemos luego: «IdleState» y «AlertState». El estado de

«Idle» no hará nada, y el de «Alert» pintará el cubo de rojo y la orientará hacia el *player*.

- Por último, crearemos un *script* llamado «enemyAI», que será el encargado de saber en qué estado nos encontramos.

```
1.  public interface IEnemyState
2.  {
3.      void UpdateState();
4.      void GoToAlertState();
5.      void GoToIdleState();
6.
7.      void OnTriggerEnter(Collider col);
8.      void OnTriggerStay(Collider col);
9.      void OnTriggerExit(Collider col);
10. }
```

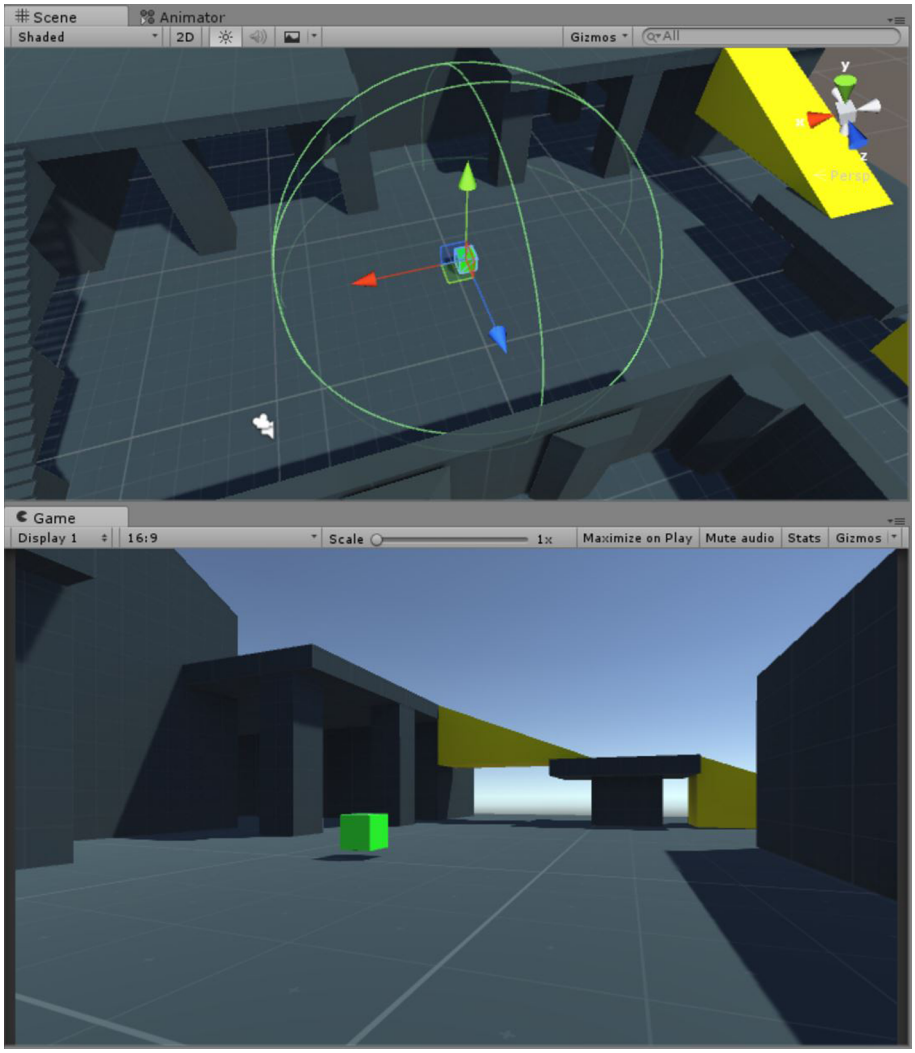
```
1.  public class IdleState: IEnemyState
2.  {
3.      enemyAI myEnemy;
4.
5.      // Cuando llamamos al constructor, guardamos
6.      // una referencia a la IA de nuestro enemigo
7.      public IdleState(enemyAI enemy)
8.      {
9.          myEnemy = enemy;
10.     }
11.
12.     // Aquí va toda la funcionalidad que queramos
13.     // que haga el enemigo cuando esté en este estado.
14.     public void UpdateState()
15.     {
16.         myEnemy.myMeshRenderer.material.color = Color.green;
17.     }
18.
19.     public void GoToAlertState()
20.     {
21.         myEnemy.currentState = myEnemy.alertState;
22.     }
23.
24.     // Como ya estamos en el estado Idle, no
25.     // llamaremos nunca a esta función desde
26.     // este estado
27.     public void GoToIdleState() {}
28.
29.     // Si el player entra en nuestro trigger
30.     public void OnTriggerEnter(Collider col)
31.     {
32.         if (col.gameObject.tag == "Player")
33.             GoToAlertState();
34.     }
35.
36.     // En el estado de Idle como el player está
37.     // fuera del trigger, no haremos nada aquí
38.     public void OnTriggerStay(Collider col) {}
39.     public void OnTriggerExit(Collider col) {}
40. }
```

```
1.  public class AlertState: IEnemyState
2.  {
3.      enemyAI myEnemy;
4.
5.      // Cuando llamamos al constructor, guardamos
6.      // una referencia a la IA de nuestro enemigo
7.      public AlertState(enemyAI enemy)
8.      {
9.          myEnemy = enemy;
10.     }
11.
12.     // Aquí va toda la funcionalidad que queramos
13.     // que haga el enemigo cuando esté en este
14.     // estado.
15.     public void UpdateState()
16.     {
17.         myEnemy.myMeshRenderer.material.color = Color.red;
18.     }
19.
20.     // Como ya estamos en el estado Shooting, no
21.     // llamaremos nunca a esta función desde este estado
22.     public void GoToAlertState() {}
23.
24.     public void GoToIdleState()
25.     {
26.         myEnemy.currentState = myEnemy.idleState;
27.     }
28.
29.     // En este estado el player ya está dentro
30.     // así que aquí no haremos nada.
31.     public void OnTriggerEnter(Collider col) {}
32.
33.     // Orientaremos el cubo mirando siempre al
34.     // player mientras él esté dentro
35.     public void OnTriggerStay(Collider col)
36.     {
37.         Vector3 lookDirection = col.transform.position -
38.                                 myEnemy.transform.position;
39.
40.         myEnemy.transform.rotation =
41.                     Quaternion.FromToRotation(Vector3.forward,
42.                                               lookDirection);
43.     }
44.
45.     // Si el player sale de nuestro radio de acción
46.     // volvemos a Idle
47.     public void OnTriggerExit(Collider col)
48.     {
49.         if (col.gameObject.tag == "Player") GoToIdleState();
50.     }
51. }
1.  public class enemyAI: MonoBehaviour
2.  {
3.      [HideInInspector] public IdleState idleState;
4.      [HideInInspector] public AlertState alertState;
5.      [HideInInspector] public IEnemyState currentState;
6.
7.      [HideInInspector] public MeshRenderer myMeshRenderer;
8.
9.      void Start()
10.     {
11.         // Creamos los dos estados de nuestra IA.
12.         idleState = new IdleState(this);
13.         alertState = new AlertState(this);
14.
15.         // Le decimos que inicialmente empezará
16.         // en Idle.
17.         currentState = idleState;
18.
19.         // Guardamos su MeshRenderer para poder cambiar
20.         // el color del cubo.
21.         myMeshRenderer = GetComponent<MeshRenderer>();
22.     }
23.
24.     void Update()
25.     {
26.         // Como nuestros estados no heredan de
27.         // MonoBehaviour, no se llama a su update
28.         // automáticamente, y nos encargaremos
29.         // nosotros de llamarlo a cada frame.
30.         currentState.UpdateState();
31.     }
32.
33.     // Ya que nuestros states no heredan de
34.     // MonoBehaviour, tendremos que avisarles
35.     // cuando algo entra, está o sale de nuestro
36.     // trigger.
37.     void OnTriggerEnter(Collider col)
38.     {
39.         currentState.OnTriggerEnter(col);
40.     }
41.     void OnTriggerStay(Collider col)
42.     {
43.         currentState.OnTriggerStay(col);
44.     }
45.     void OnTriggerExit(Collider col)
46.     {
47.         currentState.OnTriggerExit(col);
48.     }
```

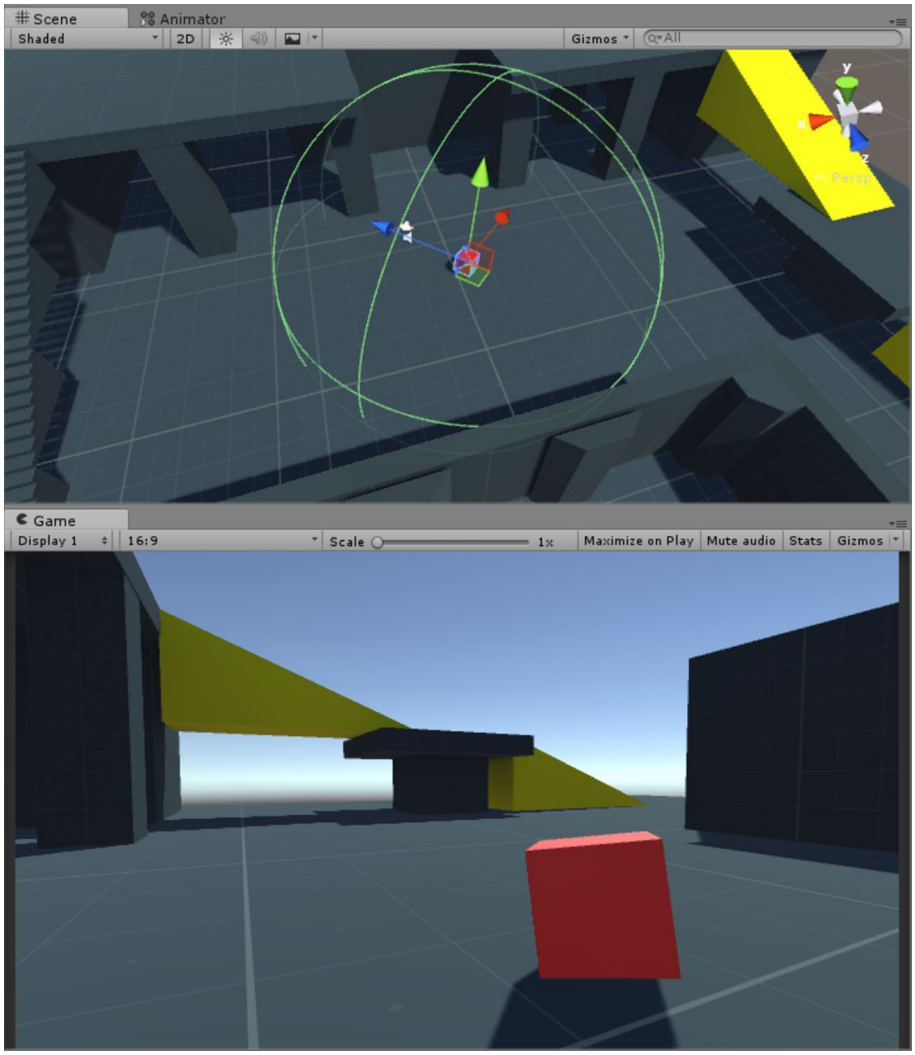
De esta manera, el «enemyAI» tan solo contiene referencias a todos los estados de la IA, *updateando* siempre el «currentState», y son los propios estados los que gestionan sus acciones y las transiciones entre ellos.

Cada estado tiene una referencia al enemigo en el que está, para poder gestionar el cambio de estados y para poder hacer todas las modificaciones pertinentes en el enemigo si hiciera falta.

«Idle State»:


Scene
Animator
Shaded
2D
Gizmos
All
y
x
z
Persp
Game
Display 1
16:9
Scale
1x
Maximize on Play
Mute audio
Stats
Gizmos

«Alert State»:


Scene
Animator
Shaded
2D
Gizmos
Persp
Game
Display 1
16:9
Scale
1x
Maximize on Play
Mute audio
Stats
Gizmos

5. Proyecto: Un juego *First Person Shooter*

Por fin ha llegado el momento esperado, ya estamos listos para crear nuestro primer *First Person Shooter* utilizando todo lo aprendido hasta ahora.

En la versión particular que haremos, crearemos una instalación futurista donde seremos un soldado que ha de escapar de la vigilancia de los drones de seguridad que lo mantienen encerrado.

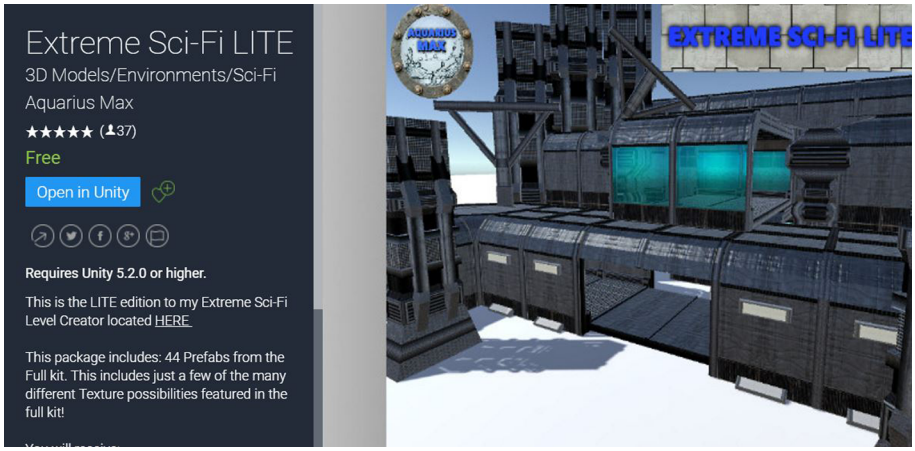
5.1. El escenario

De nuevo, nuestra intención no es modelar 3D, así que utilizaremos recursos gratuitos de la Unity Store como elementos de arte para nuestro juego. En este caso nos descargaremos el paquete *Extreme Sci-Fi LITE*, que contiene piezas suficientes para crear nuestra estación militar futurista.

Nota

Podéis descargar el *plugin* buscando su nombre en «Window > Asset Store», o desde el siguiente enlace:

https://www.assetstore.unity3d.com/en/#!/content/50727.



Una vez descargado e instalado el paquete, podremos acceder a sus *prefabs* en la carpeta «Assets/Extreme Sci-Fi LITE/Prefabs/».

En la parte de la creación del escenario lo mejor es que dejéis volar vuestra imaginación y creéis el escenario que más os guste. Lo que sí que debería es tener zonas transitables para que creemos el «NavMesh» y ser lo suficientemente grande para que podamos tener varios enemigos.

Nota

Decorar el escenario no es algo trivial. Dotar de vida y realismo a nuestros niveles es algo esencial si queremos que nuestro juego destaque.

Un truco que os ayudará es mantener apretada la tecla «V» al mover un *GameObject* para que el *gizmo* con las tres flechas se nos desplace hasta ese punto y nos permita hacer «Snap» con otros elementos del escenario muy fácilmente.

Reto 1

Cread la instalación encima de un «Terrain» y añadid además otros elementos decorativos. Esto le dará mucho más realismo y vida al escenario. En la Store hay muchos *assets* gratuitos que podréis utilizar.

5.2. El personaje principal

Una vez creada la instalación, ahora deberíamos poder movernos por ella y explorarla. Para ello, simplemente añadiremos el *prefab* del «First Person Controler» en nuestra escena en la posición que queráis que empiece la partida. Colocarlo al principio de un pasillo que simule ser la entrada a la instalación militar parece una buena opción.

Cuando os deis una vuelta por el escenario ya decorado, id mirando si los elementos decorativos tienen cajas de colisión, y si las tienen, si están bien colocadas.

También os servirán estos primeros paseos para ver si las proporciones del escenario y las del «FPSController» tienen coherencia. Intentad siempre que una unidad de Unity sea equivalente a un metro. Así os será más fácil mantener los *assets* y los personajes con tamaños y proporciones realistas.

Nota

Aunque el «Character Controller» ya es un «Collider» en sí, este no interactúa con los «Raycast», por lo que también se le debería añadir un «Capsulle Collider» a nuestro personaje si queremos que los enemigos nos detecten a través de un «Raycast».

Reto 2

Un elemento interactivo que le dará mucha vida a nuestro entorno y lo podemos hacer fácilmente gracias a los *assets* que ya tenemos; podemos añadir puertas que se abran automáticamente siempre que alguien esté cerca de ellas y que se cierren cuando ya no hay nadie.

5.3. El arma

Por ahora tenemos un *First Person*, ¡pero aún nos falta la parte de *Shooter*! Añadámosle pues un arma a nuestro personaje para que pueda disparar a diestro y siniestro.

De nuevo, utilizaremos un paquete gratuito de armas modernas de la Asset Store llamado *Modern Weapons Pack*.

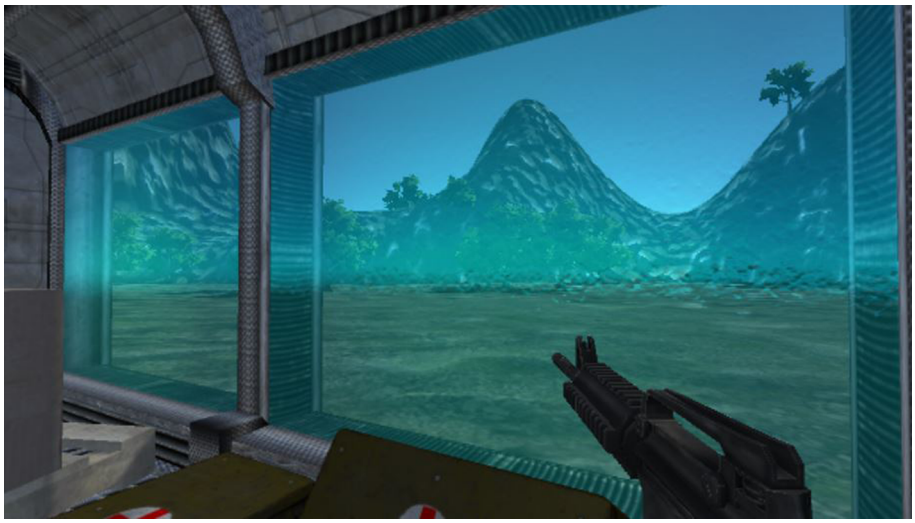
Nota

Podéis descargar el *plugin* buscando su nombre en «Window > Asset Store», o desde el siguiente enlace:

https://www.assetstore.unity3d.com/en/#!/content/14233.

Una vez descargado e instalado el paquete, podremos acceder a sus *prefabs* en la carpeta «Assets/ Modern Weapons Pack/XXXXX/FBX/».

El arma deberemos hacerla hija de la cámara, y colocarla de tal manera que podamos ver parte de esta y que quede situada en un lateral. Debería quedar más o menos así visualmente:



Ahora tan solo nos queda hacerla disparar. Para ello, lanzaremos un rayo desde el centro de la pantalla en la dirección en la que estemos mirando y detectaremos si ha colisionado con algo.

La función estática «Physics.Raycast» nos devolverá «true» si ha colisionado con algo, y guardará la información de esa colisión en la variable «hit».

```
1.  void Update()
2.  {
3.      if (Input.GetMouseButtonDown(0))
4.      {
5.          RaycastHit hit;
6.
7.          if (Physics.Raycast(
8.                  Camera.main.ViewportPointToRay(
9.                                  new Vector3(0.5 f, 0.5 f, 0)),
10.                                 out hit))
11.         {
12.            // ¡IMPACTO!
13.         }
14.     }
15. }
```

Nota

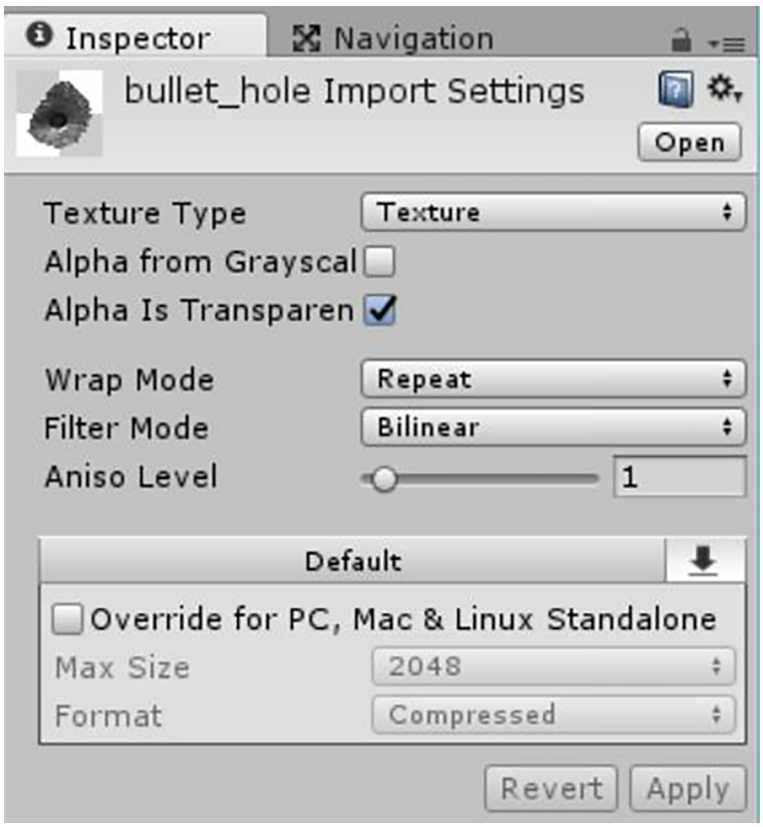
La estructura «RaycastHit» contiene información sobre el punto de colisión en el que ha impactado el rayo, como su posición, la normal de su superficie o las coordenadas de textura que tiene ese punto.

Aunque actualmente ya estamos disparando, aún no tenemos *feedback* alguno. Vamos a solucionarlo lanzando un sonido cuando disparemos y creando un agujero de bala allá donde impacte nuestro rayo.

Empezaremos creando un *quad*. Recordad que podemos crearlo fácilmente desde «GameObject > 3D Objects > Quad».

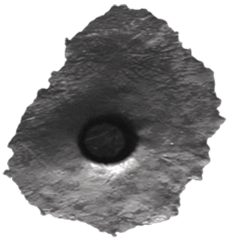
Los *quads* solo se pintan por el lado opuesto a su normal. Esto es importante saberlo, ya que al crearlo podría parecer que no se está pintando, pero lo único que sucede es que estamos mirando por el lado equivocado.

Le asignaremos a este *quad* una textura de agujero de bala. Recordad también que cuando importemos un «png» con transparencia es importante seleccionarlo en la pestaña «Project» y que le digamos que su transparencia la adquiere del alpha de la textura.



Nota

Googleando un poco y buscando cosas como «bullet hole png» encontraremos *decals* de agujeros de bala con transparencia que serán perfectos para nuestro juego.

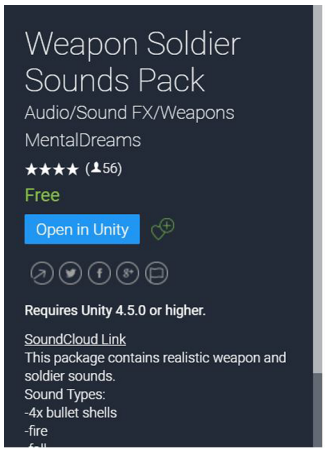


Decal

Un *decal* es aquel *quad* (normalmente con transparencia) que se coloca sobre una superficie, respetando su normal. Se suele utilizar para simular cosas como una pintada de un grafiti o cuando agujereamos una pared al disparar.

Con este *quad* crearemos un *prefab* que utilizaremos como un *decal* y que iremos instanciando cada vez que el rayo impacte con una superficie.

Por otro lado, crearemos un «Audio Source» en el arma, y le pasaremos un sonido de disparo.



Nota

Podéis conseguir el sonido de disparo de este paquete militar gratuito de la Store en el siguiente enlace:

https://www.assetstore.unity3d.com/en/#!/content/29662.

Nota

Siempre que creemos una variable pública de un tipo concreto (utilicemos como ejemplo el «Audio Source»), podremos arrastrar a su hueco en el Inspector el objeto que contiene ese componente, y Unity creará una referencia a ese componente específicamente y NO al *GameObject* completo.

Entonces ya solo nos queda hacer el *script* «shooter», que se encargará de gestionar los disparos del *player*; tendrá una referencia al *prefab* con el *decal* que acabamos de crear y otra al «Audio Source» que hemos creado en el arma.

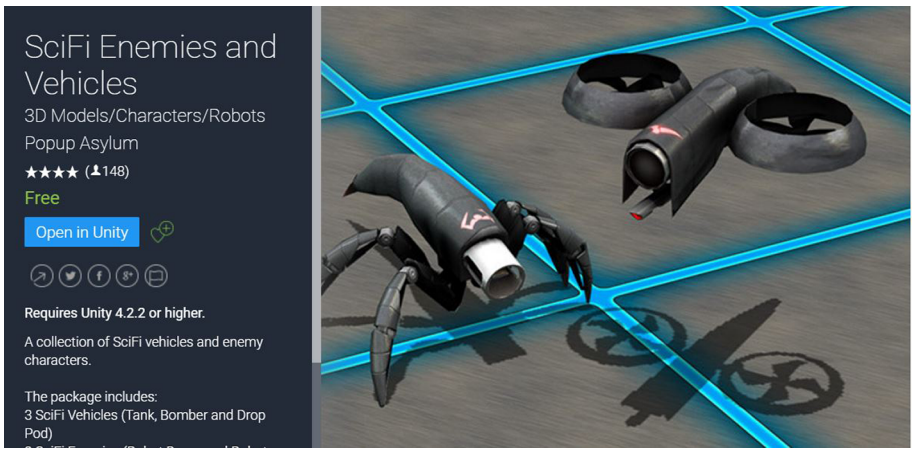

```
1.  public class shooter: MonoBehaviour
2.  {
3.      public GameObject decalPrefab;
4.      public AudioSource fireSound;
5.
6.      void Update()
7.      {
8.          if (Input.GetMouseButtonDown(0))
9.          {
10.             RaycastHit hit;
11.             if (Physics.Raycast(
12.                                 Camera.main.ViewportPointToRay(
13.                                   new Vector3(0.5 f, 0.5 f, 0)),
14.                                   out hit))
15.             {
16.                 GameObject.Instantiate( decalPrefab,
17.                                 hit.point + hit.normal * 0.01 f,
18.                                 Quaternion.FromToRotation(
19.                                                 Vector3.forward,
20.                                                 -hit.normal))
21.                 as GameObject;
22.
23.                 fireSound.Play();
24.             }
25.         }
26.     }
27. }
```

Reto 3

Tal cual tenemos el código ahora, estaremos instanciando un nuevo *quad* cada vez que disparemos, y este se quedará vivo para siempre, creando posiblemente un grave problema de rendimiento. Para evitarlo, intentad modificar el código para crear tan solo diez *decals* y que cuando vayamos a instanciar el undécimo se elimine el primero que creamos, y así sucesivamente.

5.4. Los enemigos

Ahora que ya tenemos el escenario y a nuestro personaje principal controlado como un *First Person*, ya solo nos quedan los enemigos. Para encarnar a nuestros rivales, nos descargaremos de la Asset Store un paquete de enemigos robóticos llamado *SciFi Enemies and Vehicles*.



Nota

Podéis descargar el paquete buscando su nombre en «Window > Asset Store», o desde el siguiente enlace:

https://www.assetstore.unity3d.com/en/#!/content/15159.

Una vez descargado e instalado el paquete, podremos acceder al *prefab* que utilizaremos como enemigo en la carpeta «Assets/PopupAsylum/PA_SciFiCombatants/_Imported3D/Characters/PA_Drone.prefab».

Antes de ir colocando enemigos por la escena, añadiremos solo uno y le pondremos todos los componentes y *scripts* que necesite. Luego crearemos un *prefab* propio con él, y entonces ya podremos ir colocando los enemigos por la escena.

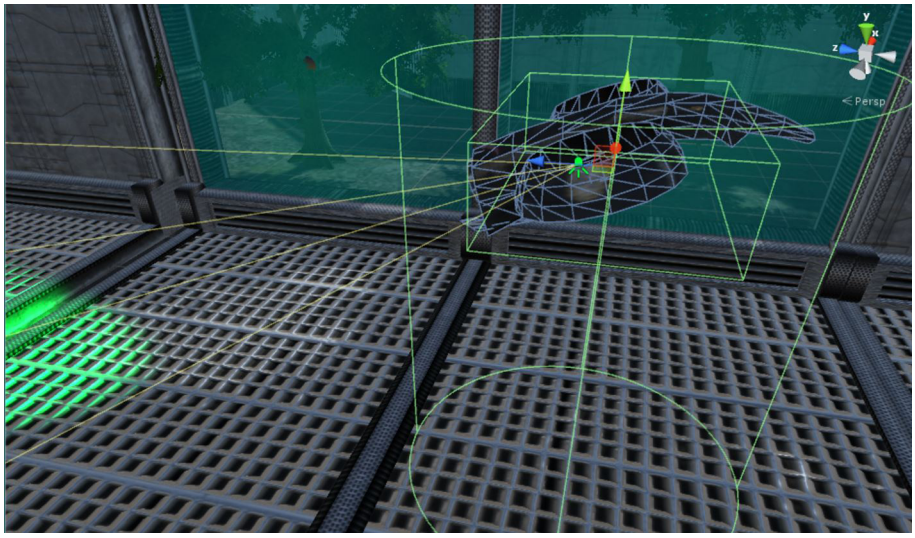
Empecemos añadiéndole el componente «NavMesh Agent» y adecuando su altura, radio y *offset* del cilindro de colisión acorde con la altura a la que queramos que esté volando nuestro dron. En nuestro caso, hemos escalado el *prefab* 2,5 veces su tamaño original y rotado un poco el hijo en el eje *X* para que parezca que siempre va inclinado hacia adelante.

También le hemos añadido una caja de colisión al hijo para detectar cuando le disparemos. Y una «Spotlight» que usaremos como chivato para saber en qué estado se va a ir encontrando la inteligencia artificial en todo momento.

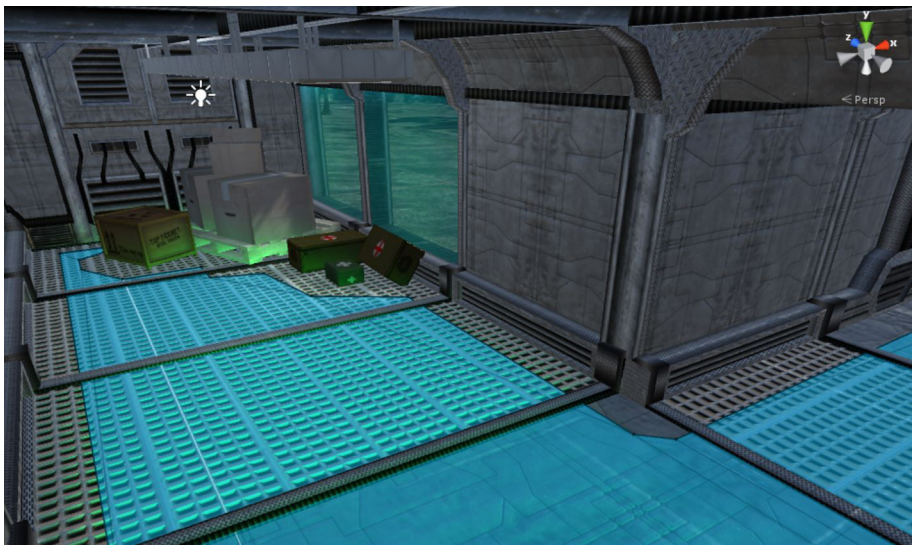
Nota

Hemos añadido la «SpotLight» haciendo clic derecho encima del *GameObject* en la Jerarquía, diciéndole «Light > SpotLight» y orientándola como si enfocara el morro de la nave.

En el módulo «Un juego de plataformas 3D» veremos cómo funcionan las luces y qué tipos hay, pero para este juego tan solo utilizaremos este tipo, que se comporta como si fuera una linterna.



Antes de seguir, crearemos el «NavMesh» del escenario para que nuestros enemigos puedan desplazarse por él. Ahora que ya sabemos los valores del «NavMesh Agent» del enemigo, los utilizaremos para crear el «Bake» del escenario tal como hemos visto anteriormente.



Nota

Recordad que las zonas de decoración como las cajas o botiquines deben marcarse como «NotWalkable» para que los enemigos no las traspasen y sepan esquivarlas.

Reto 4

Sería interesante tener también una zona de exteriores por donde «pasear» por el terreno. Quizá una puerta que se abre y por la que sales a una terraza, a un jardín o a una vasta zona desértica. Pero generando también «NavMesh» en el terreno y esquivando los árboles o las rocas.

Reto 5

El «NavMesh» nos permite crear zonas de tránsito con diferentes pesos. Podríamos crear partes del escenario que fueran más difíciles de transitar por las que los enemigos caminen más lentos, con lo que cuando calculen su camino quizá opten por evitar pasar por ellas. Estas zonas podrían ser charcos de barro, zonas de nieve densa o incluso zonas con fuerzas gravitacionales o interferencias para que los robots vayan más lentos.

Ya casi tenemos todos los elementos del enemigo completados. Solo nos quedarían sus *scripts*. Utilizando la lógica de las *State Machines*, crearemos una inteligencia artificial que tendrá los siguientes estados:

- «Patrol»: Este será el estado base de nuestro enemigo. Si nadie le molesta, irá paseando por la escena de un punto a otro («Waypoints») patrullando, haciendo la ronda. Si el *player* le dispara, pasará al estado de «Attack». Este estado tendrá una luz por defecto que será la verde.

- «Alert»: En el momento en el que el *player* entre dentro de su radio de acción, el enemigo se parará y empezará a rotar sobre sí mismo buscando al *player*; en el momento en el que lo vea se quedará fijo mirándolo y pasará a modo «Attack». Durante este modo «Alert» la luz se pondrá de color amarillo. Si tras dar una vuelta no ha visto al *player*, volverá al estado «Patrol». Si el *player* le dispara, pasará al estado de «Attack».

- «Attack»: En este estado también se quedará quieto y simplemente disparará al *player* hasta que este desaparezca de su radio de acción. En ese caso volverá a «Alert». En este estado la luz será roja.

Para hacer esto, necesitaremos también que el enemigo tenga un «SphereCollider» que usaremos como «Trigger» para detectar si el *player* ha entrado en su radio de acción. Este «Trigger» debería estar en la *Layer* «Ignore Raycast» para que luego no nos dé problemas al dispararle y para que colisione con este «Collider» y no con el otro.

Nota

Si utilizáis las marcas de bala que explicamos anteriormente, el *quad* que utilizaremos como *decal* también deberá estar en la *Layer* «Ignore Raycast» porque si no, al disparar encima de un *decal*, detectará la colisión con este y no con lo que tenga detrás.

Empecemos pues con el *script* «enemyAI», que tendrá el *GameObject* padre y que será el encargado de gestionar toda la inteligencia artificial y sus estados.

```
1.  public class enemyAI: MonoBehaviour
2.  {
3.      [HideInInspector] public PatrolState patrolState;
4.      [HideInInspector] public AlertState alertState;
5.      [HideInInspector] public AttackState attackState;
6.      [HideInInspector] public IEnemyState currentState;
7.
8.      [HideInInspector] public NavMeshAgent navMeshAgent;
9.
10.     public Light myLight;
11.     public float life = 100;
12.     public float timeBetweenShoots = 1.0 f;
13.     public float damageForce = 10 f;
14.     public float rotationTime = 3.0 f;
15.     public float shootHeight = 0.5 f;
16.     public Transform[] wayPoints;
17.
18.     void Start()
19.     {
20.         // Creamos los estados de nuestra IA.
21.         patrolState = new PatrolState(this);
22.         alertState = new AlertState(this);
23.         attackState = new AttackState(this);
24.
25.         // Le decimos que inicialmente empezará patrullando
26.         currentState = patrolState;
27.
28.         // Guardamos la referencia es nuestro NavMesh Agent
29.         navMeshAgent = GetComponent < NavMeshAgent > ();
30.     }
31.
32.     void Update()
33.     {
34.         // Como nuestros estados no heredan de
35.         // MonoBehaviour, no se llama a su update
36.         // automáticamente, y nos encargaremos
37.         // nosotros de llamarlo a cada frame.
38.         currentState.UpdateState();
39.
40.         // Morir
41.         if (life < 0) Destroy(this.gameObject);
42.     }
43.
44.     // Cuando el player nos dispara, nos quitamos vida
45.     // y avisamos al estado en el que estemos de que
46.     // nos han disparado.
47.     public void Hit(float damage)
48.     {
49.         life -= damage;
50.         currentState.Impact();
51.         Debug.Log("Enemy hitted: " + life);
52.     }
53.
54.     // Ya que nuestros states no heredan de
55.     // MonoBehaviour, tendremos que avisarles
56.     // cuando algo entra, está o sale de nuestro trigger.
57.     void OnTriggerEnter(Collider col) {
58.         currentState.OnTriggerEnter(col);
59.     }
60.     void OnTriggerStay(Collider col) {
61.         currentState.OnTriggerStay(col);
62.     }
63.     void OnTriggerExit(Collider col) {
64.         currentState.OnTriggerExit(col);
65.     }
66. }
```

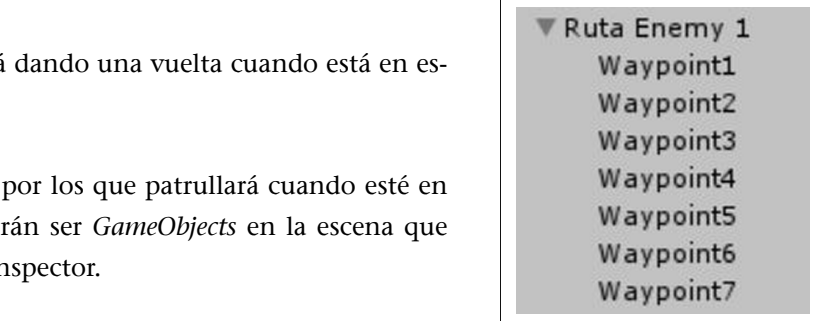
Acordaos de arrastrar la luz a la variable «myLight» del *script* «enemyAI» en el Inspector.

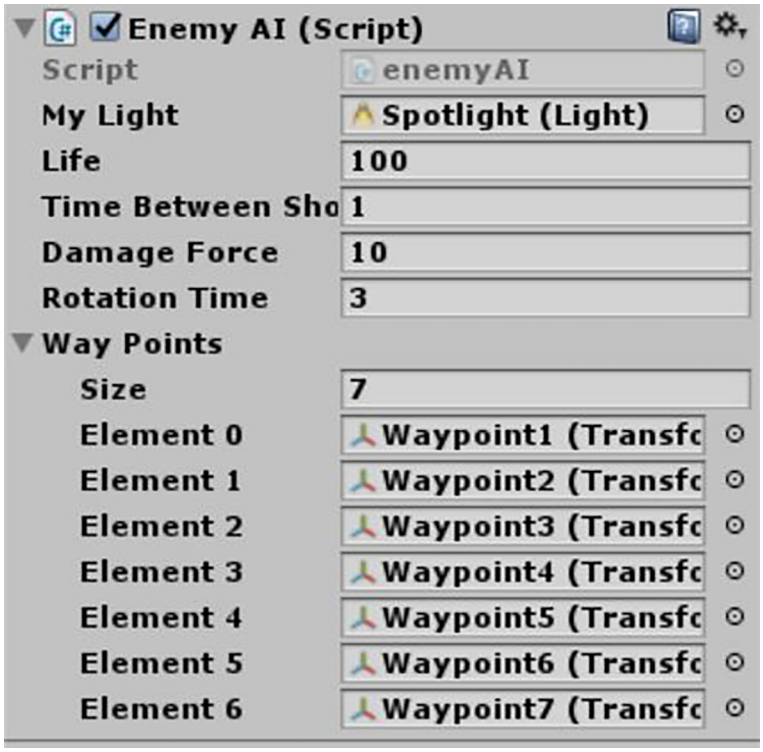
Las variables de este *script* son las siguientes:

- «Life»: La vida del enemigo.
- «TimeBetweenShoots»: Cuánto tarda entre un disparo y otro cuando ataca.
- «DamageForce»: Cuánta vida quita cada uno de sus disparos.
- «RotationTime»: El tiempo que está dando una vuelta cuando está en estado «Alert».
- «Waypoints»: Es la lista de puntos por los que patrullará cuando esté en estado «Patrol». Estos puntos deberán ser *GameObjects* en la escena que arrastraremos al *script* a través del Inspector.

Lista de los «Waypoints» en la escena

Cada enemigo deberá tener sus «Waypoints». ¡No pueden ser hijos del enemigo porque si no, se moverían con él!





Veamos entonces cómo será la interfaz de nuestros estados y el código de cada uno de ellos.

```
1. public interface IEnemyState {
2.     void UpdateState();
3.     void GoToAttackState();
4.     void GoToAlertState();
5.     void GoToPatrolState();
6.     void OnTriggerEnter(Collider col);
7.     void OnTriggerStay(Collider col);
8.     void OnTriggerExit(Collider col);
9.     void Impact();
10.  }
```

Hemos añadido a nuestra interfaz los tres estados de nuestra inteligencia artificial y una función «Impact», que se llamará cada vez que dañemos al enemigo.

```
1.  public class PatrolState: IEnemyState
2.  {
3.      enemyAI myEnemy;
4.      private int nextWayPoint = 0;
5.
6.      // Cuando llamamos al constructor, guardamos
7.      // una referencia a la IA de nuestro enemigo
8.      public PatrolState(enemyAI enemy)
9.      {
10.         myEnemy = enemy;
11.     }
12.
13.     // Aquí va toda la funcionalidad que queramos
14.     // que haga el enemigo cuando esté en este estado.
15.     public void UpdateState()
16.     {
17.         myEnemy.myLight.color = Color.green;
18.
19.         // Le decimos al NavMeshAgent cuál es el punto
20.         // al que ha de dirigirse.
21.         myEnemy.navMeshAgent.destination =
22.                         myEnemy.wayPoints[nextWayPoint].position;
23.
24.         // Si hemos llegado al destino, cambiamos la
25.         // referencia al siguiente Waypoint
26.         if (myEnemy.navMeshAgent.remainingDistance <=
27.             myEnemy.navMeshAgent.stoppingDistance)
28.         {
29.             nextWayPoint = (nextWayPoint + 1) %
30.                                       myEnemy.wayPoints.Length;
31.         }
32.     }
33.
34.     // Si el player nos ha disparado
35.     public void Impact() {
36.         GoToAttackState();
37.     }
38.
39.     public void GoToAlertState() {
40.         // Paramos su movimiento
41.         myEnemy.navMeshAgent.Stop();
42.         myEnemy.currentState = myEnemy.alertState;
43.     }
44.
45.     public void GoToAttackState() {
46.         // Paramos su movimiento
47.         myEnemy.navMeshAgent.Stop();
48.         myEnemy.currentState = myEnemy.attackState;
49.     }
50.
51.     // Como ya estamos en el estado Patrol, no
52.     // llamaremos nunca a esta función desde este estado
53.     public void GoToPatrolState() {}
54.
55.     public void OnTriggerEnter(Collider col) {
56.         if (col.gameObject.tag == "Player")
57.            GoToAlertState();
58.     }
59.     public void OnTriggerStay(Collider col) {
60.         if (col.gameObject.tag == "Player")
61.            GoToAlertState();
62.     }
63.
64.     public void OnTriggerExit(Collider col) {}
65. }
```

```
1.  public class AlertState: IEnemyState
2.  {
3.      enemyAI myEnemy;
4.      float currentRotationTime = 0;
5.
6.      // Cuando llamamos al constructor, guardamos
7.      // una referencia a la IA de nuestro enemigo
8.      public AlertState(enemyAI enemy)
9.      {
10.         myEnemy = enemy;
11.     }
12.
13.     // Aquí va toda la funcionalidad que queramos
14.     // que haga el enemigo cuando esté en este estado.
15.     public void UpdateState()
16.     {
17.         myEnemy.myLight.color = Color.yellow;
18.
19.         // Rotamos al enemigo una vuelta completa en el tiempo
20.         // indicado por rotationTime
21.         myEnemy.transform.rotation *= Quaternion.Euler(0f,
22.            Time.deltaTime * 360 * 1.0f / myEnemy.rotationTime,
23.                                                        0f);
24.         // Si hemos dado la vuelta
25.         if (currentRotationTime > myEnemy.rotationTime)
26.         {
27.             currentRotationTime = 0;
28.             GoToPatrolState();
29.         }
30.         Else
31.         {
32.             // Si aun estamos dándo vueltas lanzamos
33.             // un rayo desde una altura de 0.5m desde
34.             // la posición del enemigo hacia dónde mira
35.             RaycastHit hit;
36.             if (Physics.Raycast(
37.                   new Ray(
38.                      new Vector3(myEnemy.transform.position.x,
39.                                  0.5f,
40.                                  myEnemy.transform.position.z),
41.                      myEnemy.transform.forward * 100f),
42.                      out hit))
43.             {
44.                 if (hit.collider.gameObject.tag == "Player")
45.                 {
46.                         Debug.Log(hit.collider.name);
47.                         GoToAttackState();
48.                    }
49.                 }
50.             }
51.
52.             currentRotationTime += Time.deltaTime;
53.         }
54.
55.     // Si el player nos ha disparado
56.     public void Impact() {
57.         GoToAttackState();
58.     }
59.
60.     // Como ya estamos en el estado Alert, no
61.     // llamaremos nunca a esta función desde este estado
62.     public void GoToAlertState() {}
63.     public void GoToAttackState() {
64.         myEnemy.currentState = myEnemy.attackState;
65.     }
66.     public void GoToPatrolState() {
67.         // Volvemos a ponerlo en marcha
68.         myEnemy.navMeshAgent.Resume();
69.         myEnemy.currentState = myEnemy.patrolState;
70.     }
71.
72.     // Al estar buscando no haremos caso del trigger.
73.     public void OnTriggerEnter(Collider col) {}
74.     public void OnTriggerStay(Collider col) {}
75.     public void OnTriggerExit(Collider col) {}
76. }
```

```
1.  public class AttackState: IEnemyState
2.  {
3.      enemyAI myEnemy;
4.      float actualTimeBetweenShoots = 0;
5.
6.      // Cuando llamamos al constructor, guardamos
7.      // una referencia a la IA de nuestro enemigo
8.      public AttackState(enemyAI enemy) {
9.          myEnemy = enemy;
10.     }
11.
12.     // Aquí va toda la funcionalidad que queramos
13.     // que haga el enemigo cuando esté en este estado.
14.     public void UpdateState()
15.     {
16.         myEnemy.myLight.color = Color.red;
17.         actualTimeBetweenShoots += Time.deltaTime;
18.     }
19.
20.     // Si el player nos ha disparado no haremos nada.
21.     public void Impact() {}
22.
23.     // Como ya estamos en el estado Attack, no
24.     // llamaremos nunca a estas funciones desde este estado
25.     public void GoToAttackState() {}
26.     public void GoToPatrolState() {}
27.
28.     public void GoToAlertState() {
29.         myEnemy.currentState = myEnemy.alertState;
30.     }
31.
32.
33.     // El player ya está en nuestro trigger
34.     public void OnTriggerEnter(Collider col) {}
35.
36.     // Orientaremos el enemigo mirando siempre al
37.     // player mientras le ataquemos
38.     public void OnTriggerStay(Collider col)
39.     {
40.         // Estaremos mirando siempre al player.
41.         Vector3 lookDirection =
42.            col.transform.position - myEnemy.transform.position;
43.
44.         // Rotando solamente en el eje Y
45.         myEnemy.transform.rotation =
46.           Quaternion.FromToRotation(Vector3.forward,
47.              new Vector3(lookDirection.x, 0, lookDirection.z));
48.
49.         // Le toca volver a disparar
50.         if(actualTimeBetweenShoots > myEnemy.timeBetweenShoots)
51.         {
52.             actualTimeBetweenShoots = 0;
53.             col.gameObject.GetComponent<shooter>().Hit(
54.                                         myEnemy.damageForce);
55.             }
56.         }
57.
58.         // Si el player sale de su radio, pasa a modo Alert.
59.         public void OnTriggerExit(Collider col)
60.         {
61.             GoToAlertState();
62.         }
63.     }
64. }
```

Reto 6

Un solo tipo de enemigo en toda la escena es un poco soso. Podríais crear variaciones de este cambiando sus valores de tamaño, color, daño, vida, velocidad, cadencia de disparo, color de la luz cuando patrulla, etc.

¡Y esto es todo! ¡Ya tenemos nuestro primer *First Person Shooter*!

Claro que aún podríamos mejorarlo añadiéndole un buen HUD que muestre la vida del *player*. O añadiéndole munición limitada, o la posibilidad de poder recargar el arma, o que los enemigos muestren una barra de vida encima de sus cabezas, o también podemos crear ítems que recarguen la munición o la vida.

En un juego como este siempre se pueden añadir *features* que os motiven a seguir mejorando y aprender cosas nuevas. El demonio está en los detalles y cuanto más pulido está un juego, mejor sabor de boca deja, así que no os quedéis aquí e intentad mejorarlo en todo lo que podáis.

5.5. Soluciones a los retos propuestos

Reto 1

El terreno que creemos deberá tener una gran zona plana donde colocar nuestra instalación. Añadir un poco de vegetación siempre suma, pero hay que ser inteligente y pensar que llenar el terreno de árboles que luego no se verán tendrá un coste de rendimiento innecesario.

Respecto a la decoración, hay muchísimos elementos gratuitos en la Store. Cuáles usar en cada caso dependerá del tipo de proyecto y de la temática que se esté utilizando en cada momento.

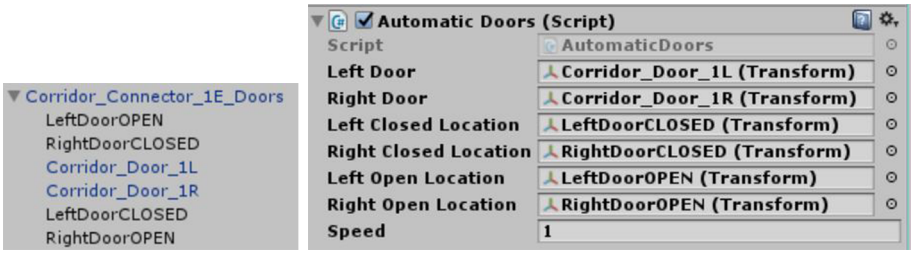
Estos serían algunos paquetes interesantes que os servirían para este escenario:

- Ítems: https://www.assetstore.unity3d.com/en/#!/content/807.
- Tuberías: https://www.assetstore.unity3d.com/en/#!/content/64170.
- Cajas y barriles: https://www.assetstore.unity3d.com/en/#!/content/73101.
- *Medical box*: https://www.assetstore.unity3d.com/en/#!/content/26967.
- *Boxes*: https://www.assetstore.unity3d.com/en/#!/content/63740.

Reto 2

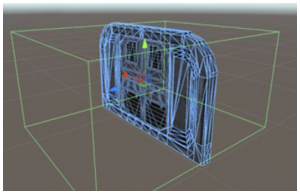
Una de las muchas maneras posibles sería reutilizar el *prefab* «Corridor_Connector_1E_Doors» y añadirle un «Box Collider» como «Trigger» que detecte si hay alguien cerca.

Y con un *script* que contenga una referencia de cada puerta y de las posiciones que deberían tener cada una de ellas cuando están abiertas o cerradas, e ir cambiando su posición a cada *frame* hasta que lleguen a la posición deseada.



Nota

El «Box Collider» que hará de «Trigger» de nuestra puerta.



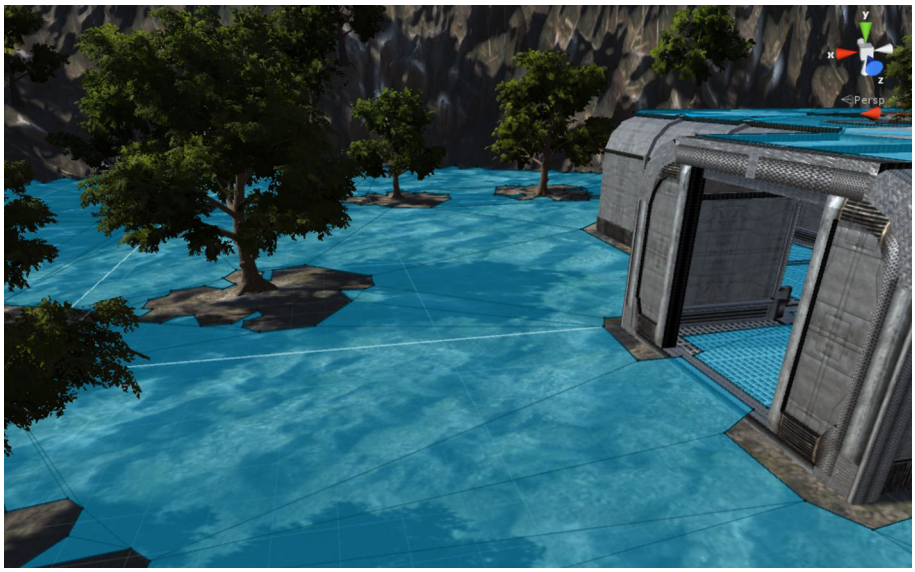
```
1.  public class AutomaticDoors: MonoBehaviour
2.  {
3.      public Transform leftDoor;
4.      public Transform rightDoor;
5.      public Transform leftClosedLocation;
6.      public Transform rightClosedLocation;
7.      public Transform leftOpenLocation;
8.      public Transform rightOpenLocation;
9.
10.     public float speed = 1.0 f;
11.
12.     bool isOpening = false;
13.     bool isClosing = false;
14.     Vector3 distance;
15.
16.     void Update()
17.     {
18.         if (isOpening)
19.         {
20.             distance = leftDoor.localPosition -
21.                        leftOpenLocation.localPosition;
22.
23.             if (distance.magnitude < 0.001 f)
24.             {
25.                 isOpening = false;
26.                 leftDoor.localPosition =
27.                                 leftOpenLocation.localPosition;
28.                 rightDoor.localPosition =
29.                                 rightOpenLocation.localPosition;
30.             }
31.             else
32.             {
33.                 leftDoor.localPosition =
34.                    Vector3.Lerp(leftDoor.localPosition,
35.                                 leftOpenLocation.localPosition,
36.                                 Time.deltaTime * speed);
37.                 rightDoor.localPosition =
38.                   Vector3.Lerp(rightDoor.localPosition,
39.                                 rightOpenLocation.localPosition,
40.                                 Time.deltaTime * speed);
41.             }
42.         }
43.         else if (isClosing)
44.         {
45.             distance = leftDoor.localPosition -
46.                        leftClosedLocation.localPosition;
47.
48.             if (distance.magnitude < 0.001 f)
49.             {
50.                 isClosing = false;
51.                 leftDoor.localPosition =
52.                               leftClosedLocation.localPosition;
53.                 rightDoor.localPosition =
54.                               rightClosedLocation.localPosition;
55.             }
56.             else
57.             {
58.               leftDoor.localPosition =
59.                  Vector3.Lerp(leftDoor.localPosition,
60.                               leftClosedLocation.localPosition,
61.                               Time.deltaTime * speed);
62.               rightDoor.localPosition =
63.                 Vector3.Lerp(rightDoor.localPosition,
64.                               rightClosedLocation.localPosition,
65.                               Time.deltaTime * speed);
66.             }
67.         }
68.     }
69.
70.     void OnTriggerEnter(Collider col) {
71.         isOpening = true;
72.         isClosing = false;
73.     }
74.     void OnTriggerStay(Collider col) {
75.         isOpening = true;
76.         isClosing = false;
77.     }
78.     void OnTriggerExit(Collider col) {
79.         isClosing = true;
80.         isOpening = false;
81.     }
82. }
```

Un reto extra podría ser añadir sonidos siempre que la puerta se abra o se cierre.

Reto 3

```
28.  public class shooter: MonoBehaviour
29.  {
30.      public GameObject decalPrefab;
31.
32.      GameObject[] totalDecals;
33.      int actual_decal = 0;
34.
35.      void Start()
36.      {
37.          totalDecals = new GameObject[10];
38.      }
39.
40.      void Update()
41.      {
42.          if (Input.GetMouseButtonDown(0))
43.          {
44.              RaycastHit hit;
45.              if (Physics.Raycast(
46.                                  Camera.main.ViewportPointToRay(
47.                                    new Vector3(0.5 f, 0.5 f, 0)),
48.                                    out hit))
49.              {
50.                  Destroy(totalDecals[actual_decal]);
51.                  totalDecals[actual_decal] =
52.                         GameObject.Instantiate( decalPrefab,
53.                                 hit.point + hit.normal * 0.01 f,
54.                                 Quaternion.FromToRotation(
55.                                                 Vector3.forward,
56.                                                 -hit.normal))
57.                         as GameObject;
58.
59.                  actual_decal++;
60.                  if (actual_decal == 10) actual_decal = 0;
61.              }
62.          }
63.      }
64.  }
```

Reto 4

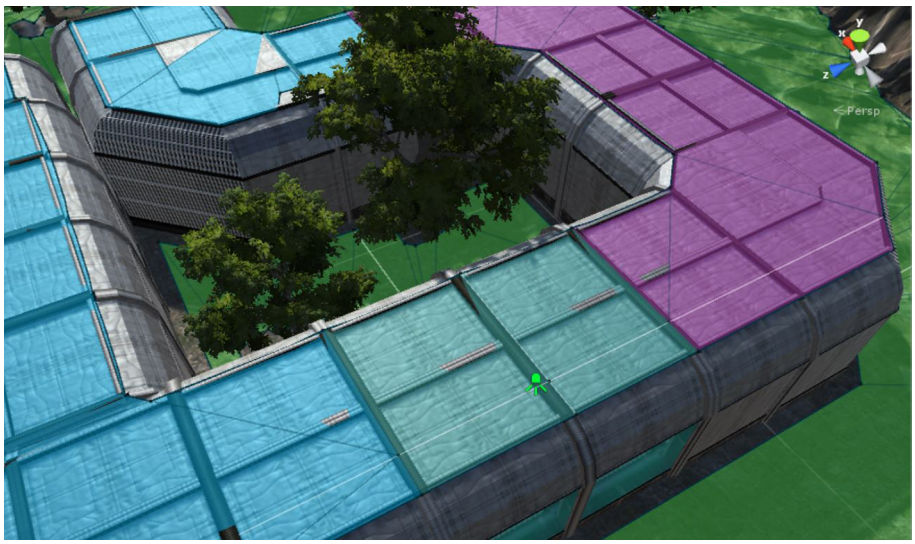


Reto 5

El primer paso sería crear los tipos de áreas que queramos y el coste que tendrá caminar por ellas.

Luego seleccionaremos aquellas *meshes* que queramos que tengan esas áreas de transición y volveremos a *bakear*.

El resultado será un «NavMesh» con diferentes colores, cada uno de los cuales representará cada una de las áreas que hemos ido añadiendo.



Reto 6

En los juegos de bajo presupuesto es muy habitual reutilizar recursos y con una misma *mesh*, cambiar la textura, los colores y los valores, y tener un enemigo nuevo. ¡Cuántas veces hemos visto en un video juego que los enemigos de un nivel son azules, pero en el siguiente hay uno rojo que quita más vida o es más rápido!

Resumen

En este módulo didáctico hemos profundizado un poco más en las herramientas que Unity nos ofrece. Hemos cómo el «Character Controller» simplifica significativamente la gestión del movimiento de un personaje, encargándose de parte de las físicas y colisiones por nosotros.

Más allá de este «Character Controller», Unity nos ofrece dos opciones ya implementadas, el «First Person Controller» y el «Third Person Controller». En este caso hemos utilizado el «First Person» para ver cómo estos *scripts* añaden una capa de control por encima para gestionar otros aspectos del movimiento de un personaje, como serían los pasos, el movimiento de la cabeza o del arma, sus sonidos, etc.

Este mismo «Character Controller» también lo podríamos haber utilizado para el movimiento de los enemigos, pero Unity nos facilita aún más el trabajo con el componente «NavMesh Agent». Este, como hemos visto, navega por encima del «NavMesh» que previamente se ha debido construir utilizando la geometría de la escena y asignándole pesos a las zonas transitables o definiendo directamente zonas «no transitables».

Gracias a este agente, nos ahorramos implementar algoritmos como el A*, que añaden complejidad al código y que son complicados de implementar.

Eso sí, de lo que no nos vamos a librar es de crear el comportamiento de nuestros enemigos. Por eso hemos visto cómo implementar máquinas de estados finitas con las que simular nuestras inteligencias artificiales utilizando interfaces.

Uno de los objetivos de este módulo es que el alumno investigue por sí mismo cómo funcionan internamente los *scripts* que Unity nos va ofreciendo en sus componentes. De la manera como están creados se puede sacar mucha información valiosa. También se aprende mucho de hacernos nuestras propias modificaciones para ir probando o para cambiar el *prefab* y que se comporte como nosotros queremos.

Así pues, hemos visto todos los pasos básicos para diseñar un *First Person Shooter*. Hemos creado nuestro propio entorno, escenario, enemigos y sus comportamientos. Y dejamos en manos del alumno mejorar esta base y darle más cuerpo al juego.