
Un juego de plataformas 3D

PID_00244519

Rafael González Fernández
Pierre Bourdin Kreitz

Tiempo mínimo de dedicación recomendado: 5 horas



Universitat
Oberta
de Catalunya

Índice

Introducción	5
Objetivos	6
1. Animator	7
1.1. El archivo .fbx	7
1.2. El avatar	12
1.3. Las animaciones	13
1.4. Animator	15
1.5. «Animator Controller»	16
2. Particle Systems	27
2.1. Propiedades	27
2.2. Código	32
3. Sonido	33
3.1. «Audio Clip»	33
3.2. «Audio Source»	33
3.3. Efectos y filtros	35
3.4. «Audio Mixer»	36
3.5. Código	37
4. Iluminación	40
4.1. Tipos de luces	40
4.2. <i>Global Illumination</i>	42
5. Proyecto: Un juego de plataformas 3D	44
5.1. El escenario	44
5.2. El personaje	47
5.3. El reloj	50
5.4. El HUD	53
5.5. Soluciones a los retos propuestos	54
Resumen	59

Introducción

En este último módulo veremos cómo crear otro de los controles de juego más habitual, un juego en tercera persona. En este tipo de juegos el control de la cámara es fundamental, especialmente que no se meta dentro de otros objetos para no entorpecer la visión al jugador.

Veremos, además, las partes importantes de Unity que nos quedaban por conocer.

Primeramente, veremos cómo funciona el «Animator» y cómo se gestionan las animaciones dentro de Unity. Veremos la composición de un FBX, qué es un avatar y cómo nos afecta de cara a reutilizar animaciones de un personaje a otro.

Después pasaremos a los sistemas de partículas. Veremos cómo se crean y cómo funcionan internamente. Son objetos sencillos pero con muchísimas opciones, lo que permite crear infinidad de sistemas y añadir complejidad a la hora de trabajar con ellos.

Dedicaremos un capítulo al sonido. Aunque ya sabemos trabajar con él, veremos aquí algunas de sus características para juegos 3D y profundizaremos en las herramientas y filtros que nos ofrece Unity para crear efectos más profesionales.

Por último, veremos la iluminación, que es una parte fundamental del motor de Unity, y también cómo funciona el *baking* automático y qué es la *Global Illumination*.

Una vez hayamos visto todos estos aspectos, en la parte práctica del módulo crearemos un juego sencillo, con control en tercera persona, con el que poner en práctica parte de lo que hemos aprendido.

Objetivos

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

- 1.** Entender cómo funciona el «Animator» de Unity, cómo crear un grafo de animaciones con él y ser capaz de animar un personaje 3D dado un set de animaciones.
- 2.** Conocer los entresijos de los sistemas de partículas y ser capaz de crear uno desde cero o de modificar uno ya existente.
- 3.** Conocer en profundidad el sistema de sonido de Unity y las ventajas de utilizar el «Audio Mixer».
- 4.** Saber trabajar con el nuevo motor de iluminación de Unity y conocer los diferentes tipos de luces que podemos utilizar. Entender cómo funciona la *Global Illumination* y qué son las «Light Probes».
- 5.** Utilizar el «Third Person Controller» y el «FreeLookCameraRig» para crear un juego en tercera persona aplicando además los conocimientos ya adquiridos en anteriores módulos.

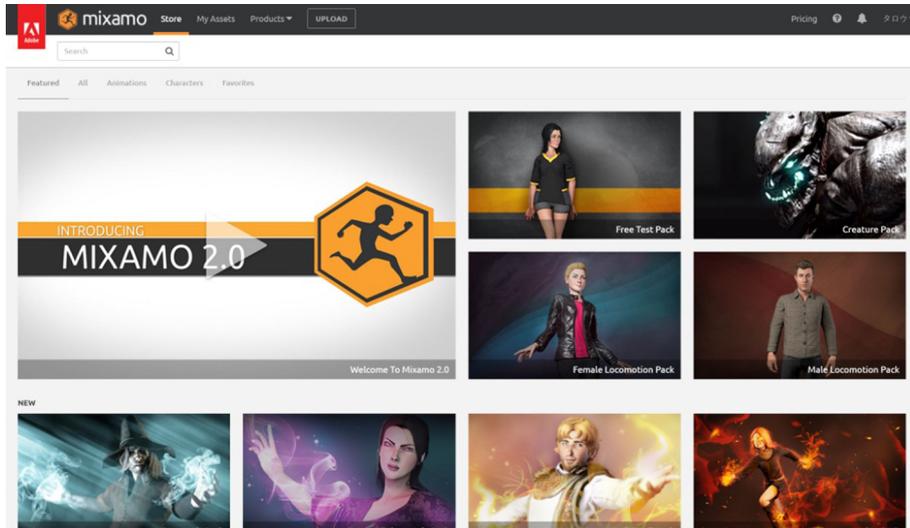
1. Animator

Un juego sin animaciones, donde todo es rígido, es como un día sin sol. Las animaciones dotan de realidad y vida a cualquier personaje o elemento del escenario. Hasta ahora no hemos trabajado con personajes 3D, pero si lo hubiéramos hecho, estos personajes habrían quedado como maniquís al no tener animaciones. Esto lo vamos a solucionar ahora mismo.

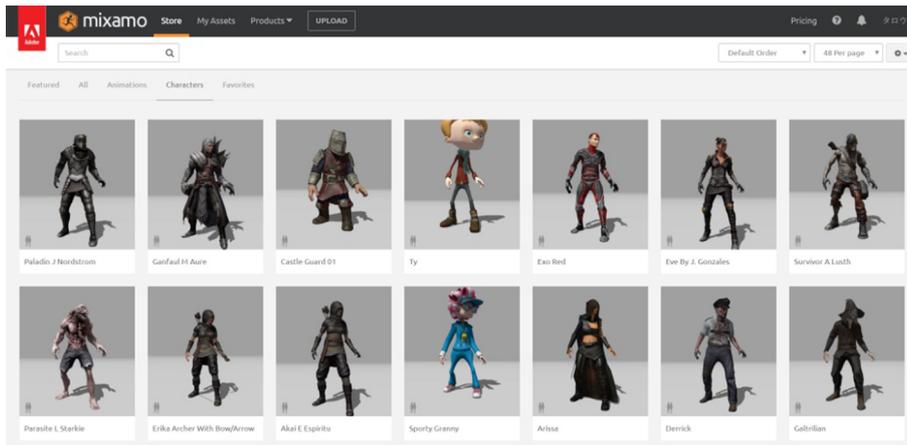
1.1. El archivo .fbx

Antes de adentrarnos a explicar qué partes componen un archivo .fbx, vamos a descargar un personaje en línea gratuito junto a un paquete de animaciones para poder ir teniendo una referencia como ejemplo y que nuestra explicación no sea solo teórica.

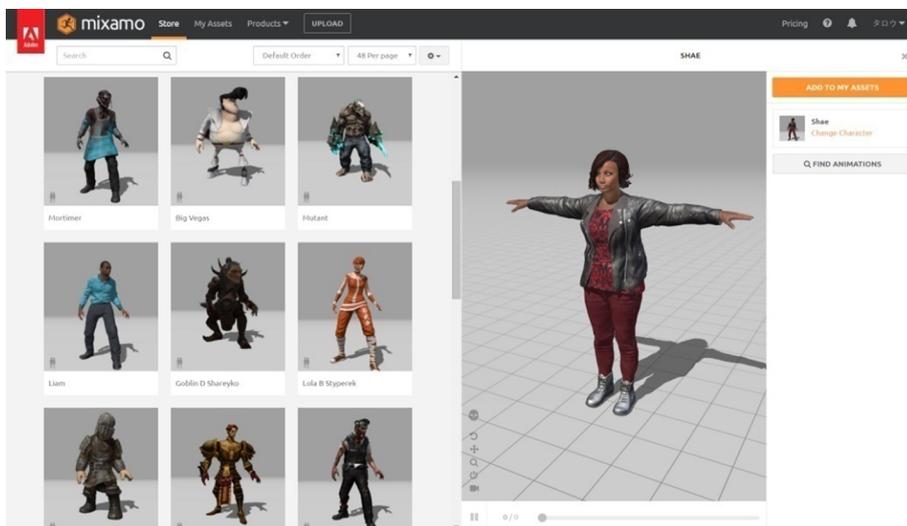
Para ello, iremos a la web <http://www.mixamo.com> y crearemos una cuenta gratuita. Mixamo es un portal de Adobe que ofrece gratuitamente varios modelos 3D de personajes completamente modelados, texturizados, *rigeados* y animados de muy buena calidad.



Una vez *logueados*, deberemos ir a la Store, y allí a la sección «Characters».

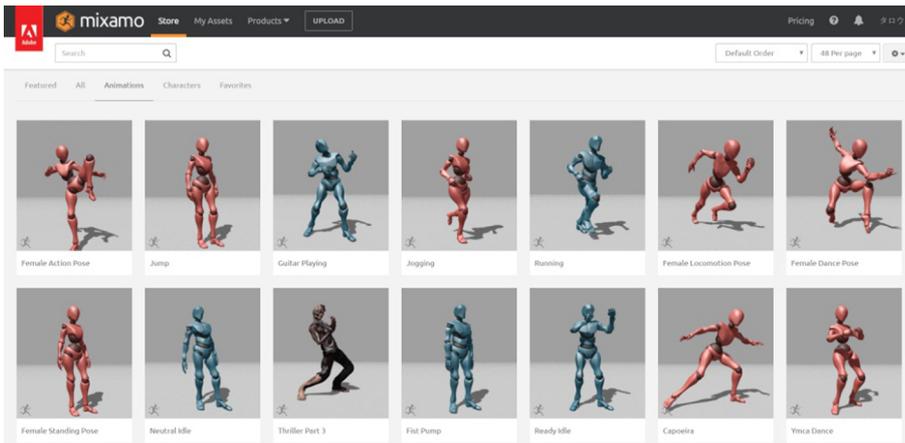


De todos los que hay, elegid el que más os guste. Para este ejemplo utilizaremos el modelo «Shae».

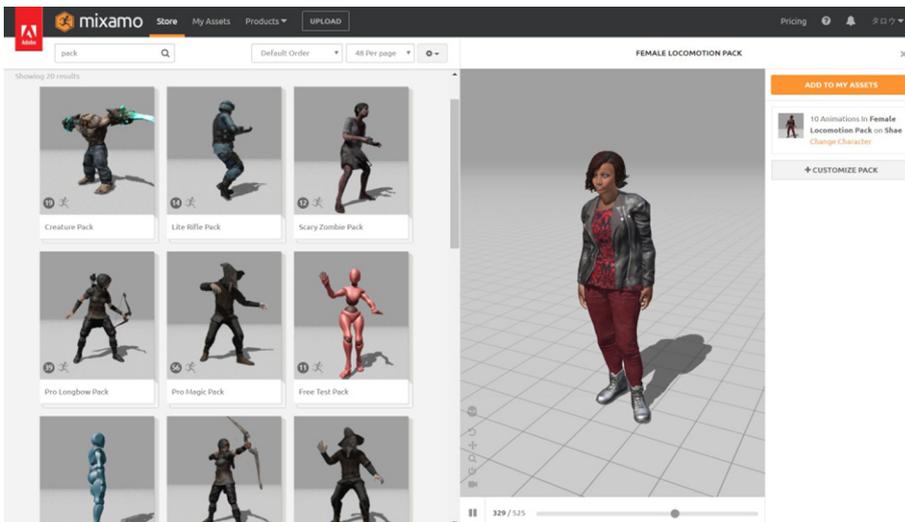


Una vez seleccionado, se abrirá un pequeño visualizador que nos permitirá ver con más detalle el modelo. Si es de nuestro agrado, tendremos que darle a «Add to my assets» para añadirlo a nuestra biblioteca de personajes.

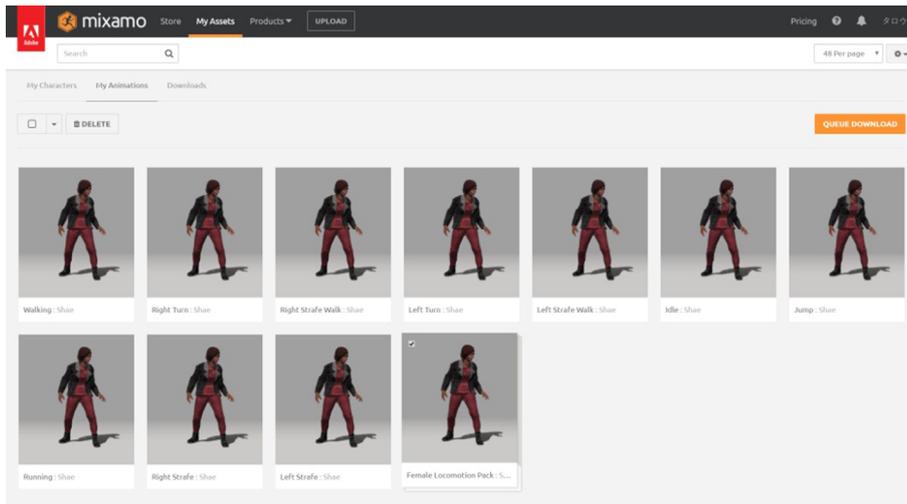
Ahora que ya tenemos un personaje, vayamos al Store de nuevo, pero esta vez a la sección de animaciones.



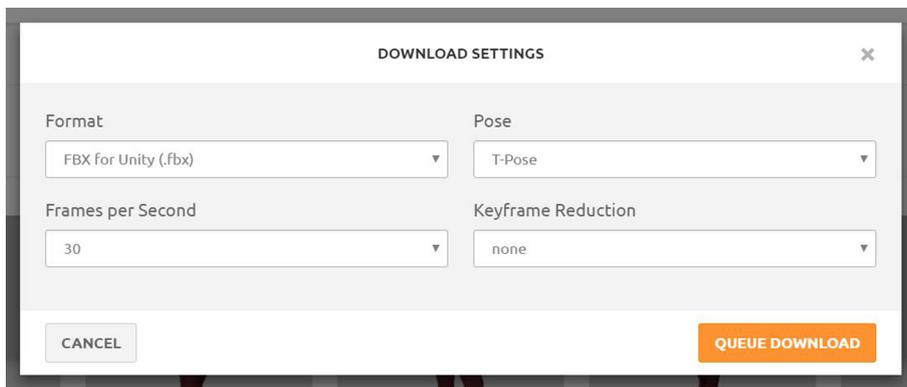
Aquí podéis ver todas las animaciones que Mixamo nos ofrece gratuitamente. Para facilitarnos el trabajo, también han creado varios paquetes con esas animaciones sueltas. Nosotros utilizaremos el *pack Female Locomotion Pack*. Para encontrarlo, podéis filtrar buscando por la palabra *pack* en su buscador.



Este es un *pack* con las animaciones básicas de movimiento de un personaje femenino. Si nos gusta, clicaremos en «Add to my assets» y ya podremos ir a la sección «My Assets», donde podremos acceder tanto a los personajes que hayamos añadido como a las animaciones.

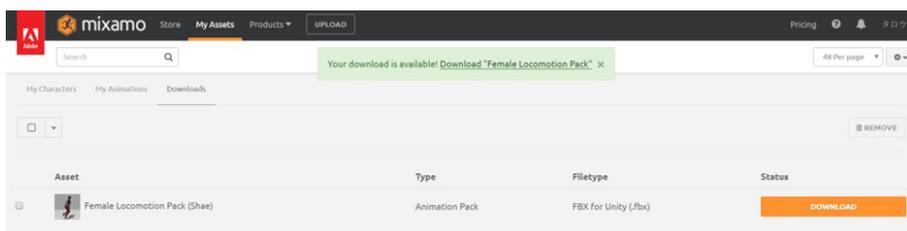


Vayamos a «My animations», seleccionamos el *pack* completo y hacemos clic en «Queue Download».

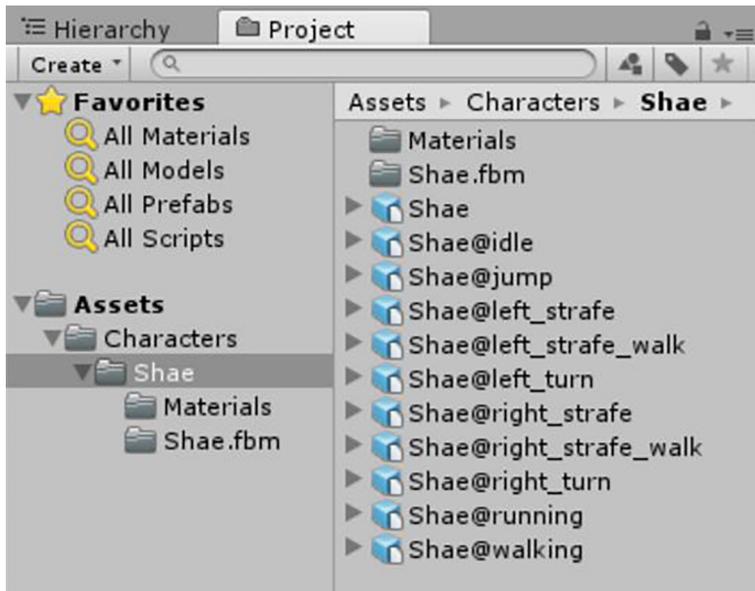


Aquí podremos seleccionar si queremos que los FBX que descarguemos estén optimizados para Unity, si queremos que el modelo venga en pose T o sin modelo; si queremos que estén a 60 fps o a 30 fps, o si queremos algún tipo de compresión.

Con los valores por defecto pero cambiando el formato a «FBX for Unity» ya estaremos listos. Una vez lo añadimos a la cola de descargas, estará un tiempo procesando mientras crea los FBX, y cuando acabe ya los tendremos listos para descargar.



Una vez descargado tan solo tendremos que descomprimirlo dentro del proyecto de Unity en la carpeta que queramos.

**Nota**

Cuando Mixamo antes nos pregunta si lo queremos optimizado para Unity, lo que hace es cambiar el nombre del archivo añadiendo el del modelo 3D delante de la @ y luego el nombre de la animación. Con esto Unity sabe que, si seleccionamos la animación, en el previsualizador del Inspector utilizará el modelo 3D del nombre antes de la @.

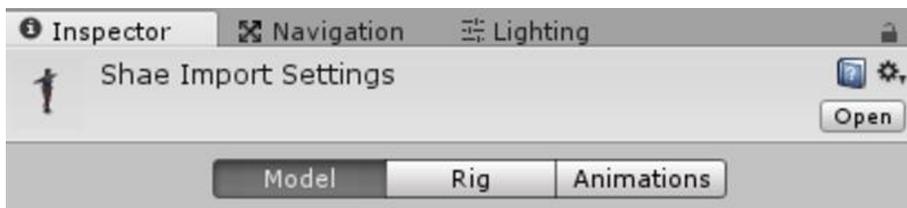
Nota

Para una información más detallada sobre que información contienen los FBX animados y cómo funciona internamente toda la parte de animación de un FBX, podéis consultar la documentación de Unity en la web:

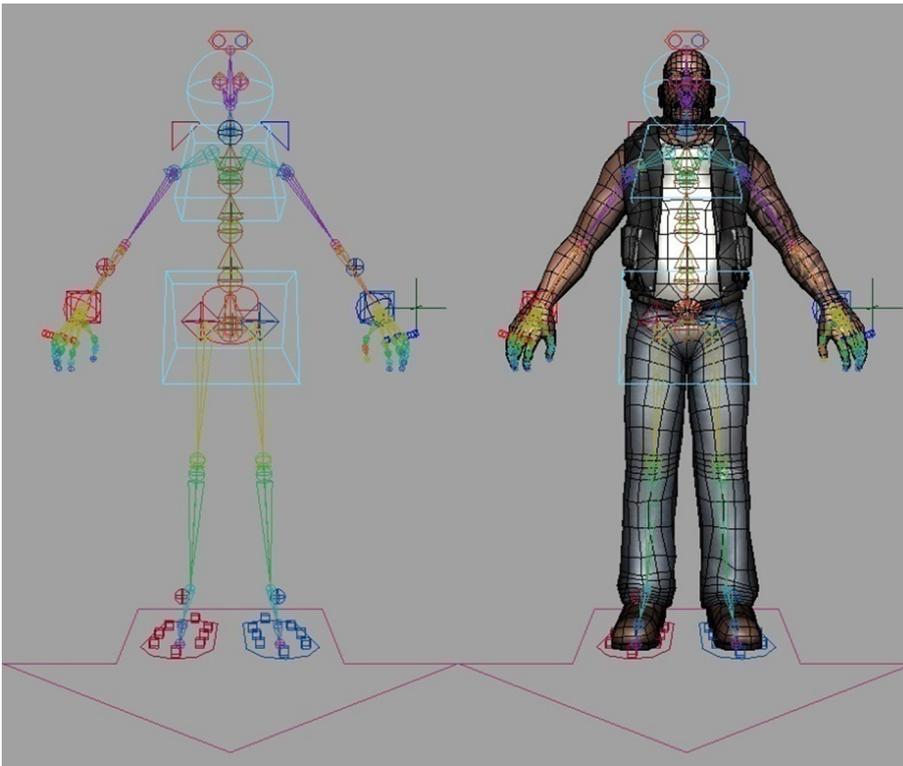
<https://docs.unity3d.com/Manual/AnimationsImport.html>.

Podemos ver que el Zip contiene varios FBX, uno con solamente el modelo, y otros que contienen el esqueleto y una animación en concreto.

Cuando seleccionamos un archivo .fbx vemos que este está dividido en tres partes: «Model», «Rig» y «Animaciones». Aunque no todos los FBX tienen por qué tener las tres a la vez.



- «Model»: Contiene la *mesh* con la geometría de ese modelo. Contiene la información de todos sus vértices, el mapeo de las texturas y los triángulos que forman esa *mesh*.
- «Rig»: Contiene la información del esqueleto asociado a ese modelo y el «Rig» o peso de esos vértices respecto a la geometría del modelo. Qué vértices se moverán y con qué influencia lo harán al mover cada uno de los huesos del esqueleto.
- «Animations»: Contiene la lista de animaciones asociadas a ese esqueleto. Existe un conjunto de *frames* y asociada a cada *frame* habrá una pose de ese esqueleto. Agrupados en conjuntos de *frames* tenemos cada una de las animaciones, pudiendo tener varias animaciones en cada FBX.

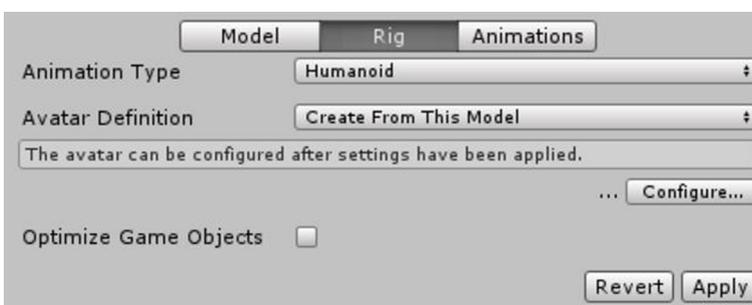


1.2. El avatar

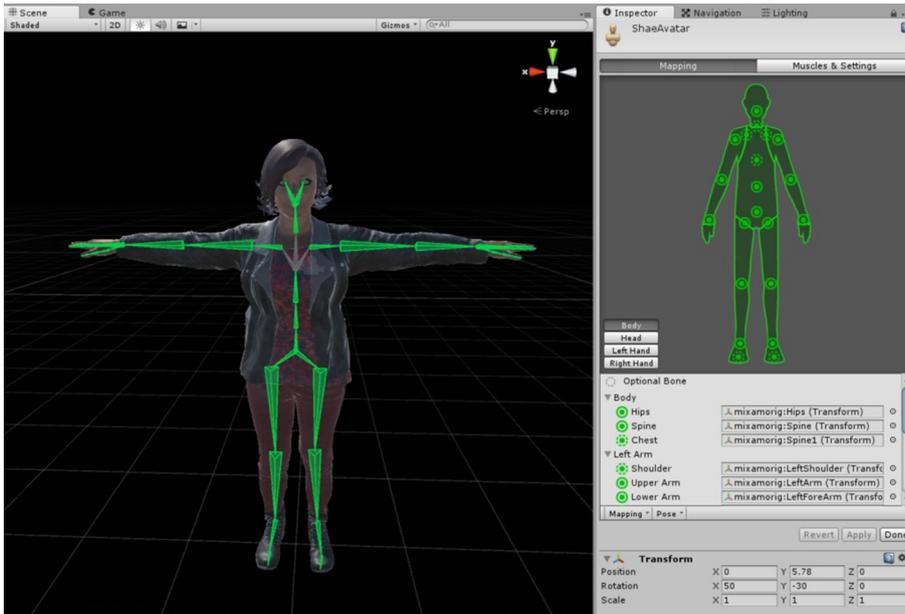
Unity tiene un sistema propio de gestión de animaciones humanoides llamado Mecanim que permite reutilizar animaciones entre unos esqueletos y otros siempre que estos sigan una estructura similar.

La base para trabajar con Mecanim es la creación de un avatar que contiene la información del *rigging*, pero adaptada a las necesidades de Mecanim.

Cuando importamos un modelo, Unity nos dejará editar qué tipo de avatar queremos en nuestro «Rig», si uno de tipo «Humanoid» para poder sacar partido a Mecanim, o uno de tipo «Generic» que funcionará de la manera tradicional.



Como norma general, si estamos utilizando personajes humanoides, ya sean humanos, trolls, gnomos o aliens, que tengan dos piernas, dos brazos, con rodillas, codos, etc., se recomienda utilizar un avatar del tipo «Humanoid».



Nota

Si entramos en la configuración del avatar, veremos cómo Unity asocia varios huesos en zonas como el torso, los brazos, las piernas, las manos, los pies y la cabeza. Esto luego nos será muy útil para crear máscaras con las que bloquear ciertas partes del cuerpo y que no ejecuten esa animación o ejecuten otra distinta.

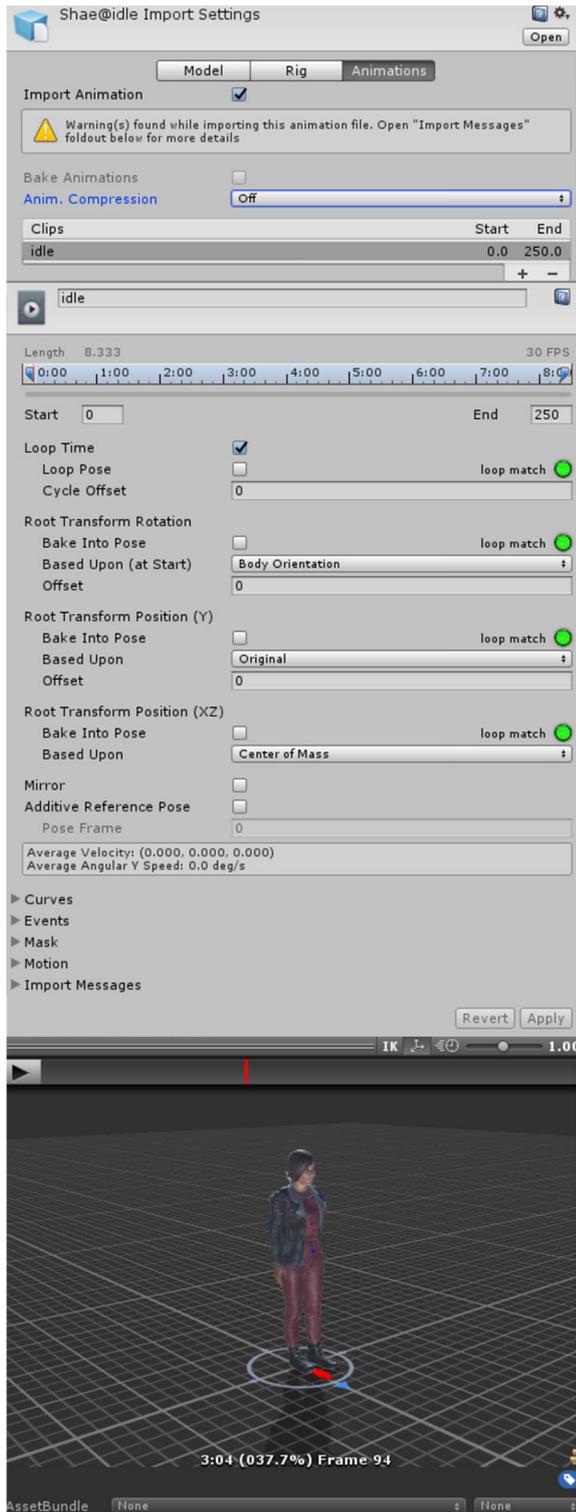
En la sección «Muscles & Settings» podremos cambiar las limitaciones de rotación que tendrán nuestras articulaciones en el caso de que queramos un comportamiento específico para un personaje.

Para los casos como animales, aliens de tres piernas o incluso algo tan sencillo como un humano con cola, y como ya se salen del estándar, se recomienda utilizar un avatar «Generic».

Así pues, un primer paso será seleccionar todos los FBX que hemos añadido al proyecto y decirles que en su «Rig» genere un nuevo avatar de tipo «Humanoid» a partir del mismo modelo de esqueleto que tiene cada archivo.

1.3. Las animaciones

Una vez tenemos vinculado el esqueleto a la geometría, ya solo nos queda ver cómo funcionan las animaciones en Unity y cómo podemos modificarlas si nos interesa.



- «Anim. Compression»: Suele ir bien para ahorrar espacio, memoria y rendimiento, pero a veces modifica las poses de la animación, que no acaban de ser exactamente igual que en el original. Algunas veces nos interesará tenerla activada y otras veces no.
- «Clips»: Es una lista con los clips de animación creados a partir de todos los *frames* que tenemos animados. Por ejemplo, imaginemos que tenemos una serie de 150 *frames* que contiene una animación de salto. De ahí podríamos

sacar tres clips. Del 0 al 50: entrada al salto; del 51 al 100: volando; del 101 al 150: volviendo al suelo.

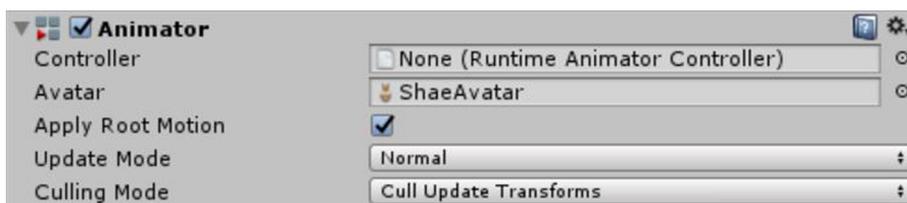
- «Loop Time»: Cuando marquemos esta casilla, le estaremos diciendo a Unity que el clip seleccionado es un bucle que se deberá repetir una y otra vez. Hay animaciones que se ejecutarán solo una vez, como por ejemplo «dar un puñetazo», y otras que serán un bucle, como por ejemplo «quieto respirando (Idle)».
- «Root Transform Rotation/ Position Y/ Position (XZ)»: Se refieren al movimiento o rotación que pueda sufrir el personaje al ejecutar el clip de animación seleccionado. Por ejemplo, un clip de salto tendrá un desplazamiento en Y.
- «Loop Match»: Si la luz está verde, quiere decir que la animación empieza muy cerca de donde acaba según la categoría en la que esté. Por ejemplo, si la animación fuera rotar 360 sobre uno mismo, la luz de «Transform Rotation» estaría verde. Cuanto más difiera la última pose de la última de verde, pasará a naranja, y de naranja a roja si no coinciden de ninguna manera.
- «Mirror»: Modificará el clip de animación para crear una réplica pero invertida, como si la viéramos reflejada en un espejo. Por ejemplo, si el clip levanta la mano derecha, al hacer «Mirror» levantará la izquierda.

1.4. Animator

Ahora que ya hemos visto cómo se gestiona y almacena la información sobre las animaciones dentro de los archivos .fbx, y cómo las procesa y adapta Unity, vamos a ver cómo implementarlas en nuestros juegos.

Si arrastramos el FBX de Shae a la escena, veremos cómo Unity lo añade automáticamente al nuevo *GameObject* que se ha creado.

Animator es el componente que hará de nexo entre las animaciones y nuestro código.



- «Controller»: El «Animator Controller» es la base del Animator. Es el que controla qué animaciones se están ejecutando y cuáles son las transiciones entre ellas. Enseguida lo veremos con más detalle.
- «Avatar»: Contiene la referencia al avatar del modelo.
- «Apply Root Motion»: Si esta casilla está marcada, la animación moverá físicamente al modelo 3D según lo haga la animación. Así que si estamos ejecutando una animación de caminar, el *GameObject* empezará a desplazarse automáticamente hacia delante aunque nosotros no hayamos hecho nada por código. Si la desactivamos, las animaciones se ejecutarán estáticas en el mismo punto sin aplicarle translaciones ni rotaciones al *GameObject*.

1.5. «Animator Controller»

Dentro del Animator, el «Animator Controller» es el que realmente se encarga del trabajo duro. Se trata de una máquina de estados en la que en cada estado contendrá una animación, y las transiciones entre estados serán las condiciones para cambiar de un estado a otro.

Para crear un controlador de animaciones deberemos hacer «Assets > Create > Animator Controller» o usar el segundo botón en el contenido de la pestaña «Project» y «Create > Animator Controller».

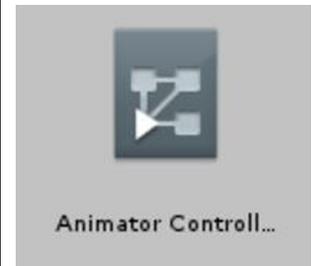
Este componente deberá arrastrarse a la caja «Controller» de nuestro Animator. Para editarlo, tan solo tendremos que hacer doble clic en él y se nos abrirá la pestaña Animator, que será en la que trabajaremos para crear el grafo de las animaciones.

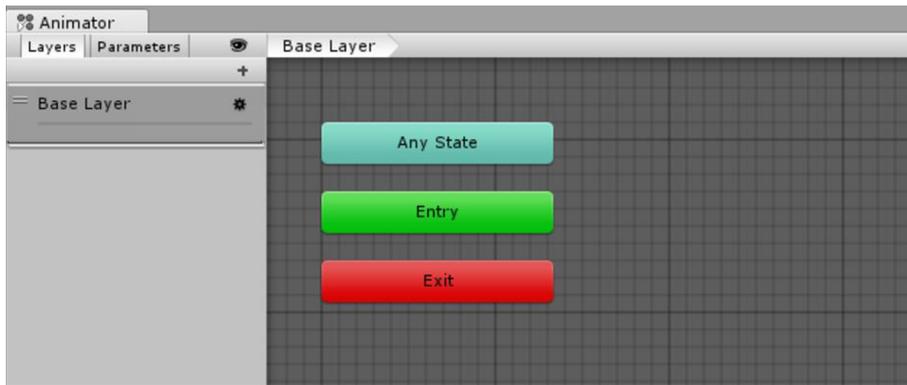
El «Animator Controller» está formado por *Layers* y por *Parameters*. El controlador puede estar formado por diversas *Layers*, y en cada *Layer* tener un grafo de animaciones distinto. Y todas se ejecutarán a la vez.

La «Base Layer» es la *Layer* que viene por defecto y que utilizaremos en este ejemplo.

Nota

Icono del «Animator Controller».



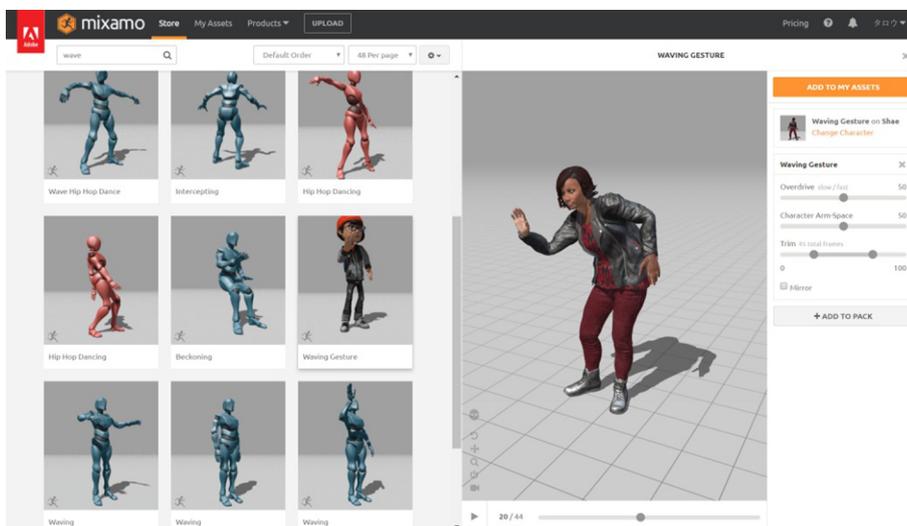


Los *Parameters* serán las variables de control con las que se decidirá si se cambia de un estado a otro.

Veamos a hora un ejemplo práctico.

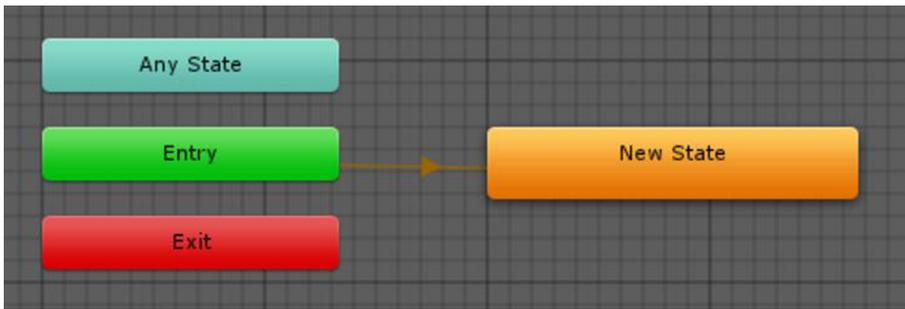
Vamos a crear un Animator muy sencillo en el que tendremos a Shae en «Idle» de pie respirando y cada vez que cliquemos en la pantalla ella nos saludará con la mano. Para ello, volveremos a Mixamo, descargaremos la animación *Waving Gesture* con el personaje de Shae y guardaremos el FBX en la misma carpeta que los anteriores.

Cuando nos pida descargar, es recomendable decirle «Without Skin», ya que no hace falta que nos descarguemos el modelo de nuevo.

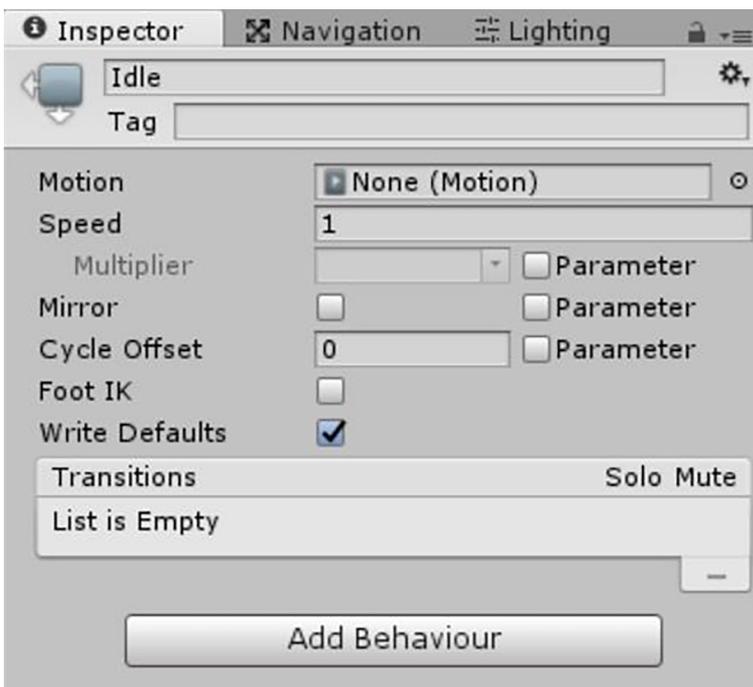


Recordad también que una vez añadido al proyecto, hay que decirle en el «Rig» que genere un avatar «Humanoid» a partir del mismo modelo.

Una vez tenemos la nueva animación añadida al proyecto, vamos a crear dos estados de animación dentro de nuestro «Animator Controller». Para ello, en el fondo cuadrículado de la ventana del Animator clicaremos «segundo botón > Create State > Empty».

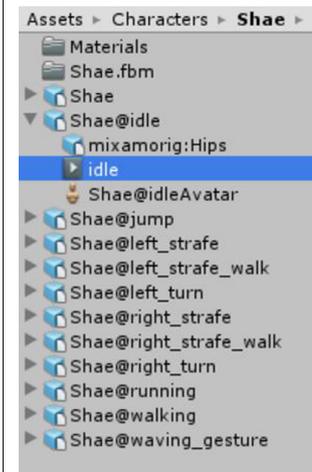


Cuando cliquemos en ese estado, podremos ver sus propiedades en el Inspector:



Nota

También podremos asignarle la animación al estado buscándola en el Proyecto y arrastrándola a la propiedad «Speed» del estado de animación. Otra opción es arrastrar directamente la animación a la zona cuadrículada del «Animator Controller» y Unity nos creará un estado automáticamente con esa animación.

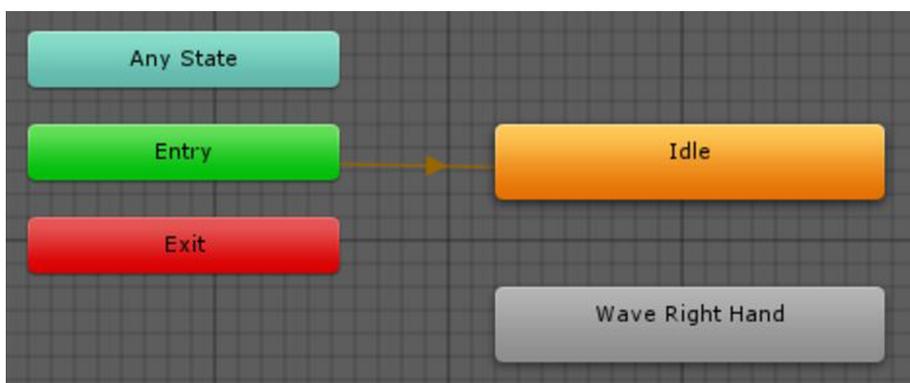


- «Nombre»: En la parte superior izquierda aparece el nombre del estado donde podremos cambiarlo. En este caso lo cambiaremos a «Idle».
- «Motion»: Aquí es donde asignaremos la animación deseada. En este caso si clicamos encima del pequeño círculo que se encuentra a la derecha nos aparecerán todas las animaciones que tenemos importadas en nuestro proyecto. Buscaremos «Idle» y se la asignaremos.
- «Speed»: Es la velocidad a la que se ejecutará la animación.
- «Mirror»: Nos permite girar la animación como si la estuviéramos mirando en un espejo.
- «Transitions»: Luego veremos esta parte con más detalle, pero aquí es donde se añadirán las condiciones que permiten pasar de ese estado a otro.

Ahora que ya tenemos un «Animator Controller» básico con al menos un estado de animación, asignémoslo al componente Animator de nuestro personaje Shae y démosle al «Play».

Podemos ver que el personaje deja de estar en pose T para ejecutar la animación de «Idle» en bucle. Si no lo hiciera en bucle, recordad que seguramente sea porque en el FBX no le hemos dicho que esa animación ha de ser un bucle marcando la casilla «Loop Time» (ver apartado «Las animaciones»).

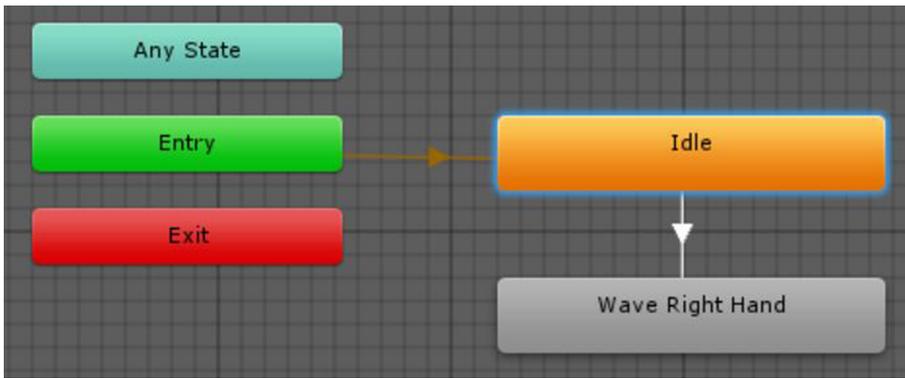
Ahora que ya tenemos a nuestro personaje animado, vamos a añadirle la animación de saludar. Para ello, crearemos un nuevo estado y le añadiremos la animación que acabamos de descargar.



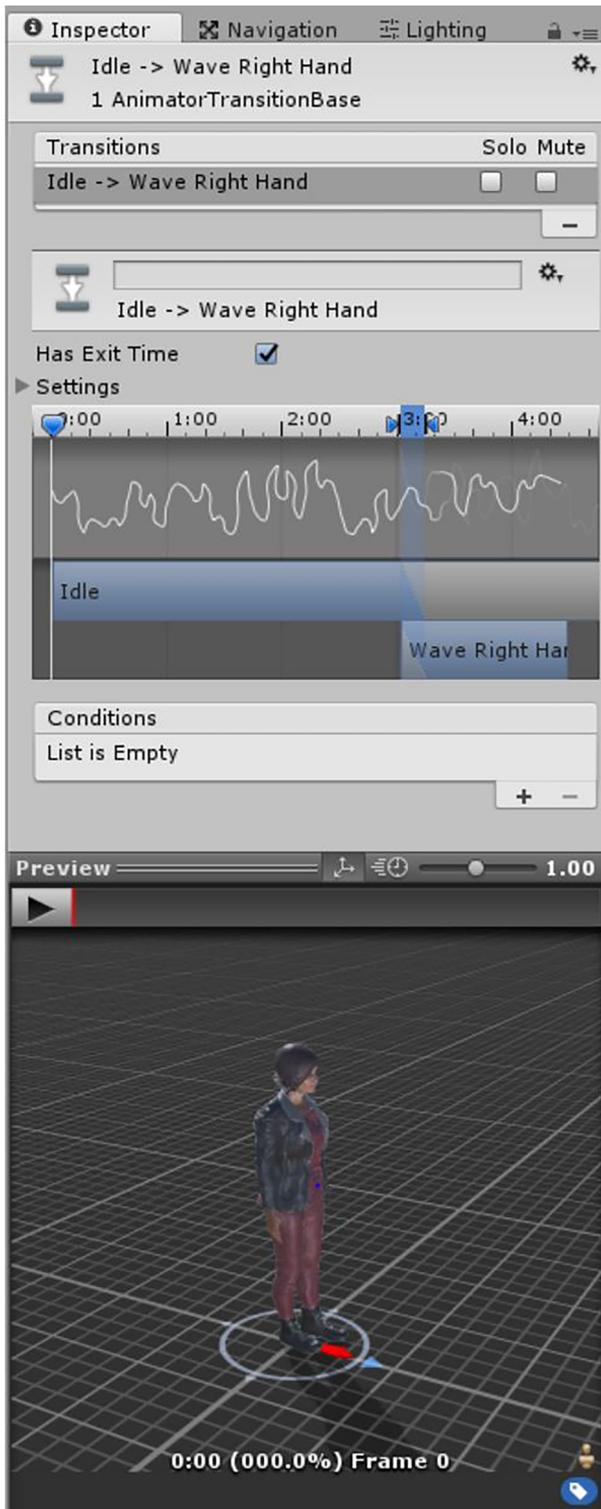
Si hacemos clic con el segundo botón encima de este nuevo estado y seleccionamos «Set as Layer Default State», veremos que ahora es este estado el que se pone en naranja. Esto quiere decir que si ahora le damos al «Play», nuestro personaje empezará directamente ejecutando esta animación. Recordad que esta animación no ha de ser un bucle, así que es normal que solo la ejecute una vez.

Volvamos a dejar «Idle» como estado inicial y veamos cómo funcionan las transiciones.

Si hacemos segundo botón encima del estado «Idle» y clicamos en «Make Transition», veremos que aparece una flecha blanca que persigue al ratón. Esta flecha indica la transición que irá desde el estado en el que la hemos creado hasta el siguiente estado en el que hagamos clic.



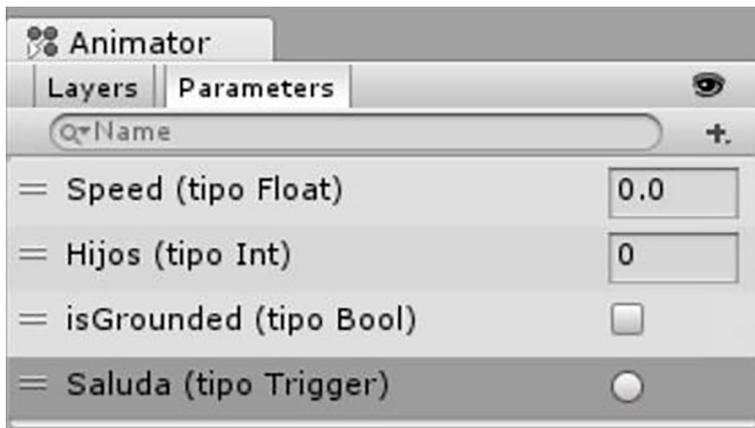
Si clicamos encima de la flecha que indica la transición, accederemos a sus propiedades en el Inspector.



- «Transitions»: Aquí se listarán todas las transiciones que van desde ese estado al siguiente. Puede ser que tengamos más de una posible condición para ir de un estado a otro, por lo que podemos tener más de una transición. En la sección inferior podremos ponerle nombre a las transiciones en el caso de que lo consideremos oportuno.

- «Has exit time»: Si está marcado, implicará que no se ejecutará esta transición hasta que se haya llegado al tiempo marcado por la zona azul del gráfico inferior.
- «Blending»: Esta parte es la zona principal de la transición, formada por una barra de tiempo en la parte superior y una representación de los dos estados justo debajo. La zona azul delimitada entre dos flechas indica cuánto durará el *blending* entre las dos animaciones. Para cambiar su tamaño, tan solo tendremos que arrastrar alguna de las dos flechas. Si queremos moverla, arrastraremos el interior azul entre las dos flechas en la barra de tiempo. Si movemos la rueda central del ratón, podemos hacer *zoom* en la ventana y así trabajar con más precisión.
- «Conditions» es la parte donde irán listadas todas las condiciones que se han de cumplir para que se ejecute la transición. Si no hay ninguna, la transición se hará cuando lleguemos al inicio del *blending*. Si «has exit time» está apagado, la transición se activará en cuanto se cumplan todas las condiciones.
- «Preview»: Si le damos al «Play», podremos ver en tiempo real cómo se hará el *blending* de las dos animaciones y nos servirá para que vayamos adaptando el tiempo de *blend* o su punto de inicio a nuestras necesidades.

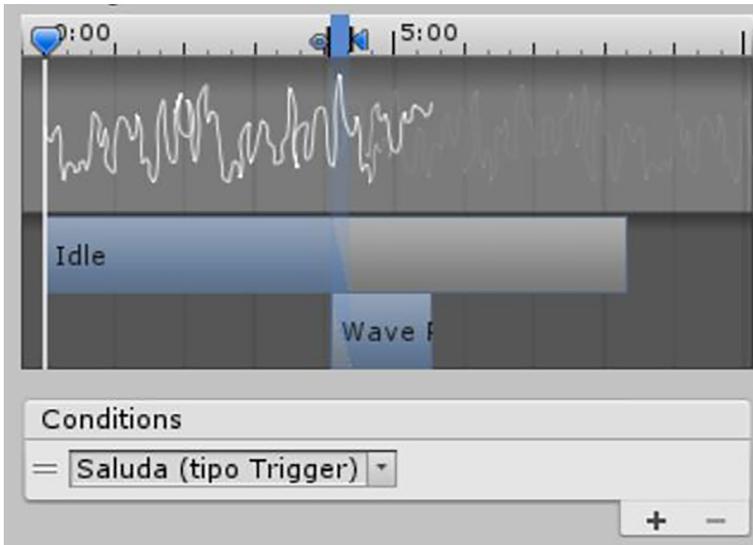
Antes de definir las condiciones de esta transición, hemos de ver qué son los *Parameters*.



En la pestaña «Parameters» crearemos variables con el icono «(+)», que nos servirán para controlar el cambio entre los estados. Estas variables pueden ser de cuatro tipos: «Int», «Float», «Bool» o «Trigger».

«Trigger» es un tipo de parámetro que actúa de manera parecida a un booleano. Cuando se activa, puede activar una animación, pero una vez ha sido utilizado en una transición, se desactiva automáticamente.

Estas variables las utilizaremos en las condiciones de las transiciones. Por ejemplo, en nuestro caso le diremos a la transición que acabamos de crear que se lance cuando se active el «Trigger» «Saluda».



A estas variables se le asignan valores por código, pero si queremos las podemos cambiar desde el Editor.

Si le damos al «Play» y tenemos seleccionado el *GameObject* que contiene el componente Animator, desde la ventana del Animator podremos activar el «Trigger» «Saluda» y ver cómo Shae cambia de animación.

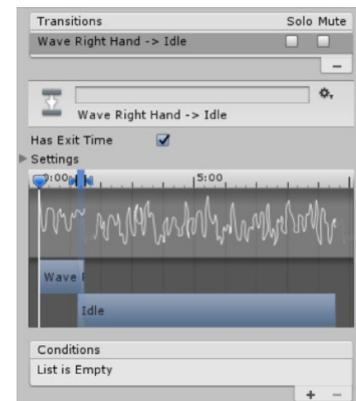
Deberíamos añadir una nueva transición que vaya de «Wave» a «Idle» y que no tenga condición de salida, tan solo el «Has Exit Time», y que su *blending* se ejecute al finalizar la animación de «Wave».

Ahora ya solo nos queda el código. Añadiremos el siguiente *script* al *GameObject* de Shae.

```

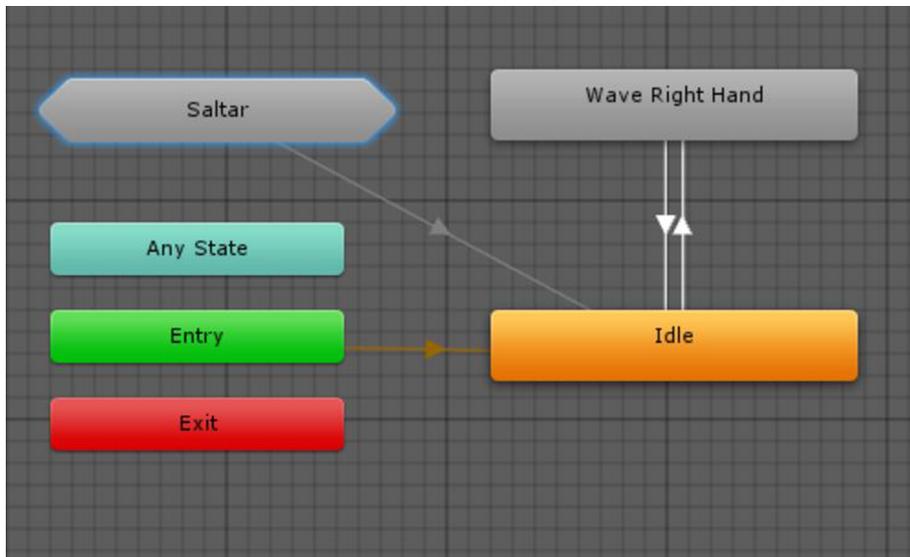
1. public class SaludaMe: MonoBehaviour
2. {
3.     private Animator myAnimator;
4.
5.     void Start()
6.     {
7.         myAnimator = GetComponent<Animator>();
8.     }
9.
10.    void Update()
11.    {
12.        if (Input.GetMouseButtonDown(0))
13.        {
14.            // De esta manera tan sencilla asignamos la variable
15.            myAnimator.SetTrigger("Saluda");
16.        }
17.    }
18. }

```



Aparte de los estados de animación normales, tenemos otros dos tipos de estados.

Uno es una «Sub-State Machine». Se trata de un estado que contiene otro grafo de estados. Se puede utilizar como contenedor para agrupar estados con una temática similar. Por ejemplo, «Saltar» podría ser un estado único, pero según cómo tengamos las animaciones podría ser una «Sub-State Machine».

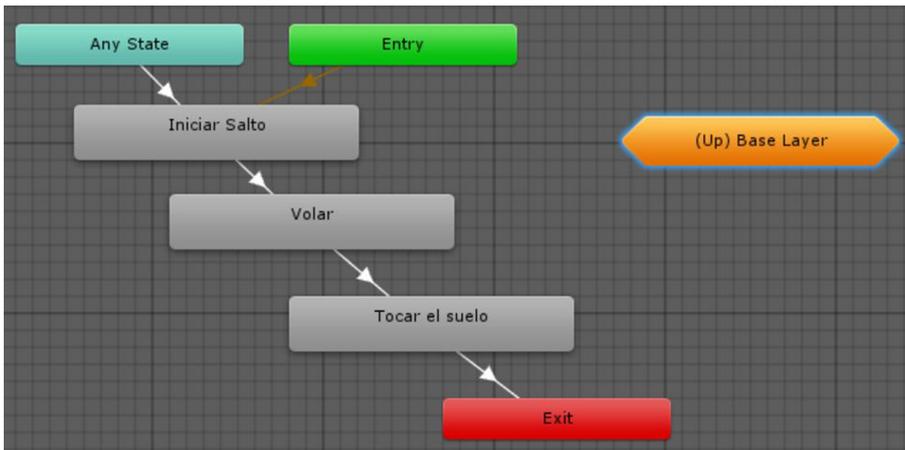


Nota

Hay ciertos estados especiales:

- «Any State» se refiere a «cualquier estado». Si hacemos una transición que salga de ese estado, cuando se cumpla su condición, esté el controlador en la animación que esté, ejecutará esa transición. Es como una transición común a todos los estados.
- «Entry» es el punto de entrada a la *State Machine*. La transición que salga de aquí indicará a qué estado irá nada más entrar en esta «Sub-State Machine».
- «Exit» es el mismo concepto que anteriormente pero para salir de la «Sub-State Machine». Cuando estemos dentro de una «Sub-State Machine» siempre nos aparecerá un estado similar a otra «Sub-State Machine» que hará referencia al grafo de animación justo por encima del que nos encontremos. En el caso de la imagen del ejemplo tendremos acceso a la «Base Layer», que es la capa donde tenemos los estados de «Idle» y «Wave».

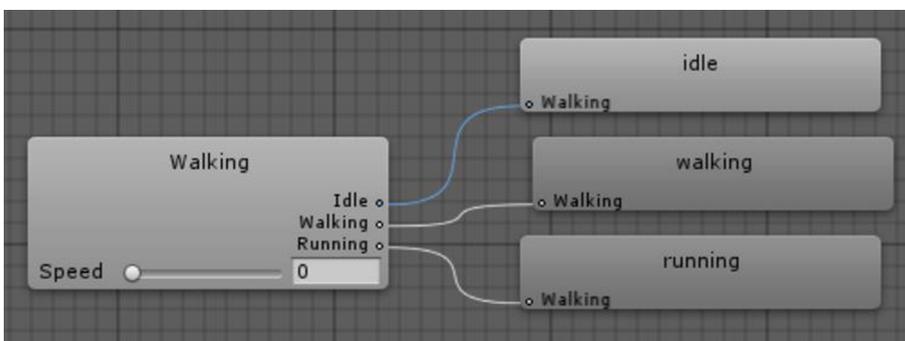
Y dentro de esa «Sub-State Machine» tendríamos los diferentes estados que componen la animación de salto.



Y por último también tenemos los «Blend Trees».

Un «BlendTree» es un tipo de estado de animación que, en vez de tener un clip de animación asociado, está formado por varios clips de animación que a partir de una o varias de las variables de control (*Parameters*) crearán un *blend* de esas animaciones y darán como resultado una sola animación.

Es muy común utilizarlos para animaciones de desplazamiento, como vemos en el siguiente ejemplo, donde hemos creado un «BlendTree» llamado «Walking» que está formado por tres clips de animación: «Idle», «Walking» y «Running».

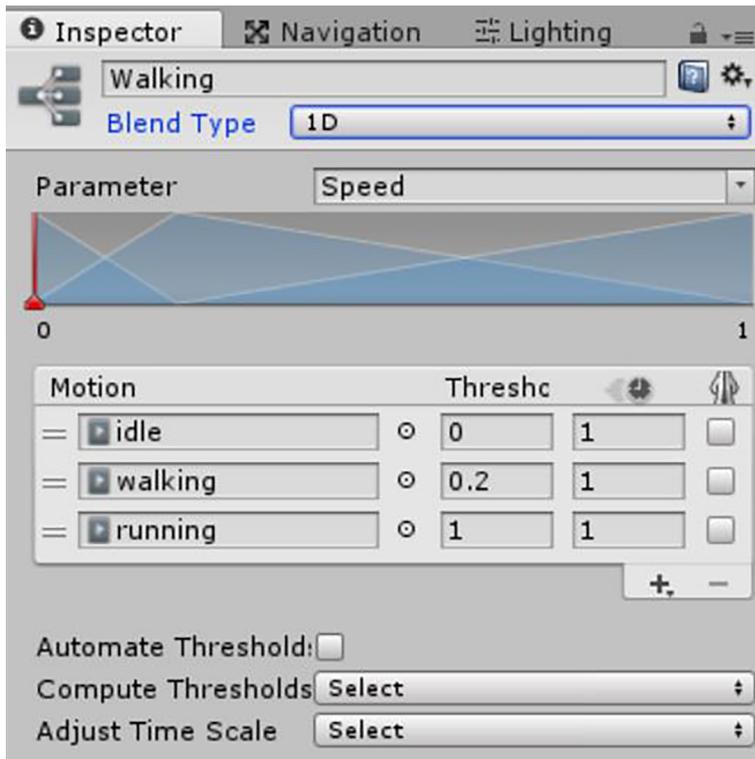


Este es un tipo de «BlendTree» («Blend Type») 1D, donde una sola variable de los *Parameters* marcará cómo haremos el *blend*. En este caso la variable es el *float* «Speed».

Nota

Podéis obtener más información sobre cómo funcionan los «Blend Trees» y sobre los diferentes «Blend Types» que hay en la ayuda de Unity:

<https://docs.unity3d.com/Manual/class-BlendTree.html>.



- El campo «Motion» es una lista con todos los clips de animación que queremos añadirle. Se añaden y se borran con los iconos «(+)» y «(-)».
- El campo «Threshold» indica el valor que ha de tener la variable «Speed» para que se ejecute esa animación al 100 %.

Si le damos al «Play» en la ventana inferior de previsualización y vamos moviendo el *slide* de la variable que tiene la cajita del estado «BlendTree» «Walking», podremos ver cómo la animación se va adaptando a un clip o a otro.

Si mapeásemos el movimiento del *joystick* (entre 0 y 1) a la variable «Speed», estos valores de animación serían los correctos, ya que, si no movemos el *joystick*, Shae ejecutará la animación de «Idle». Si lo movemos un poquito, irá caminando y si lo apretamos, al máximo correrá.

2. Particle Systems

Otro de los elementos habituales en cualquier videojuego son las partículas. Con ellas podemos representar efectos que normalmente no podríamos conseguir ni con una *mesh* 3D ni con *sprites* 2D debido a sus peculiares características.

Suelen utilizarse como complementos para añadir detalles y realismo a las escenas, como por ejemplo las chispas de una bala al tocar una pared, el fuego y el humo de una antorcha, los efectos mágicos, los líquidos en movimientos, las nubes, etc.



Normalmente, cada efecto de partículas está compuesto por diversas partículas individuales con diferentes comportamientos, pero como se gestionan en conjunto, producen el efecto deseado. Por eso se conocen como sistema de partículas.

2.1. Propiedades

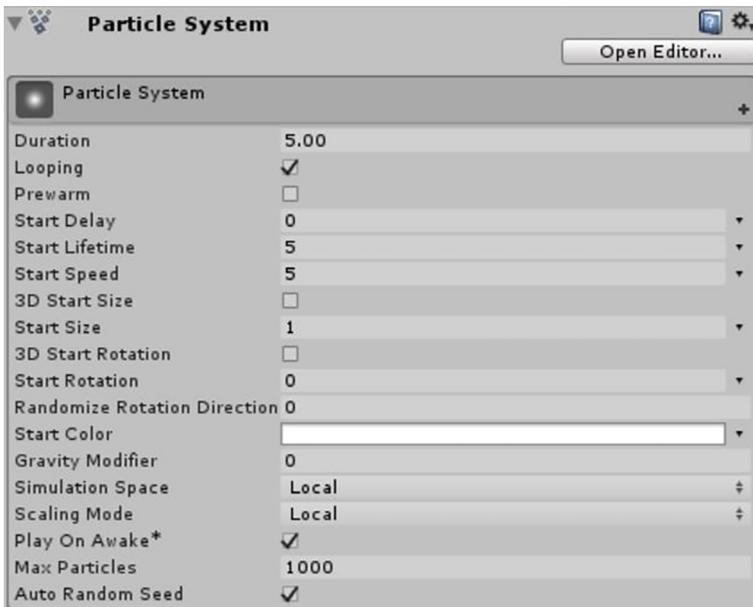
La creación de sistemas de partículas realistas es bastante compleja y requiere un trabajo muy fino, ya que se han de tener en cuenta bastantes elementos a la hora de definir cómo será nuestro sistema.

El sistema consta, por un lado, de un emisor de partículas donde podremos configurar su forma y la frecuencia, fuerza y dirección con la que las dispara. Y, por otro lado, de las partículas en sí, cada una de las cuales tendrá un tiempo de vida, unos valores de color, tamaño, velocidad, escala, rotación, transparencia, etc., que irán cambiando según el tiempo que lleven vivas.

Nota

Podemos imaginarnos los sistemas de partículas como fuentes de partículas, y el tamaño de esa fuente, cómo lanza las partículas, con qué frecuencia, con qué fuerza, etc., lo transformará en un sistema u otro.

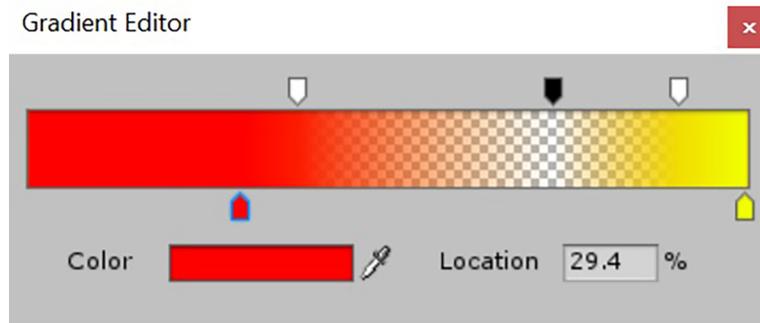
Para añadir un sistema a nuestra escena, tan solo tendremos que ir a «GameObject > Particle System». Una vez añadido, veamos cuáles son sus principales propiedades básicas:



- «Duration»: Marca el tiempo que estará emitiendo nuestro sistema. Pasado ese tiempo, dejará de emitir.
- «Looping»: En el caso de estar marcada, el sistema no parará nunca, y una vez termine volverá a empezar su ciclo.
- «Prewarm»: Esta opción solo es aplicable a los sistemas que tienen *loop*, y hace que las partículas aparezcan por primera vez como si ya se hubiera ejecutado el primer ciclo de emisión.
- «Start Delay»: Segundos que tardará en emitir el sistema desde que se active.
- «Start Lifetime»: Define el tiempo en segundos que durará viva una partícula desde su emisión.
- «Start Speed»: Velocidad a la que empezará desplazándose cada partícula.
- «Start Size»: Lo mismo, pero con su tamaño. La casilla «3D Start size» define si podremos cambiar el tamaño uniformemente o cada eje en particular.
- «Start Rotation»: Igual que el tamaño, pero con la rotación inicial.
- «Randomize Rotation Direction»: Indica el porcentaje entre 0 y 1 que definirá si las partículas se emiten con la rotación indicada (1) o en sentido

opuesto (0). Si le damos valor 0,5, la mitad rotarán en una dirección y la otra mitad en la otra.

- «Start Color»: El color con el que empezarán a emitirse las partículas. También puede utilizarse un gradiente que definirá el color según el tiempo que lleve el sistema emitiendo. Las dos flechas inferiores definen el color que tendrá el gradiente en ese punto. Las flechas superiores definen el nivel de transparencia que tendrá el color en el tiempo transcurrido que lleva el sistema emitiendo.



- «Gravity Modifier»: Define cuánta gravedad se le aplicará a las partículas una vez emitidas.
- «Simulation Space»: Con este parámetro decidiremos si queremos que las posiciones de las partículas sean siempre referentes al emisor («local»), o si por el contrario queremos que sean relativas al mundo («world»). Si quisiéramos crear el efecto del humo de un tubo de escape en un coche en movimiento, por ejemplo, deberíamos seleccionar aquí, «world». La diferencia entre estas dos opciones solo será visible si tenemos un sistema en movimiento mientras se emite.
- «Scaling Mode»: Si queremos que la escala se aplique en referencia a la escala de este *GameObject* concreto, o si queremos que se aplique respecto a la escala conjunta de todos sus padres.
- «Play on Awake»: Si está activada esta casilla, en cuanto el sistema esté activo, se activará el emisor. Si está desactivado, tendremos que activar el sistema por código mediante la función «Play».
- «Max Particles»: Indica el número máximo de partículas activas que podrá tener el sistema a la vez.
- «Auto Random Seed»: Si está activada, cada vez que lancemos el sistema de partículas, el *random* dará unos valores distintos.

Nota

Hay ciertas propiedades que pueden definirse de diversas formas:

«Constant»:



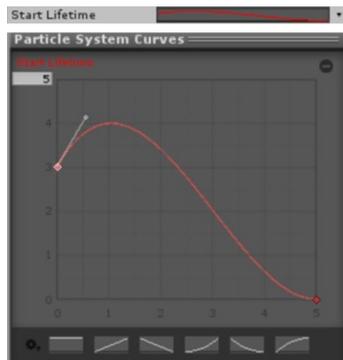
El valor de la propiedad es una constante siempre fija en el tiempo.

Random entre dos constantes:



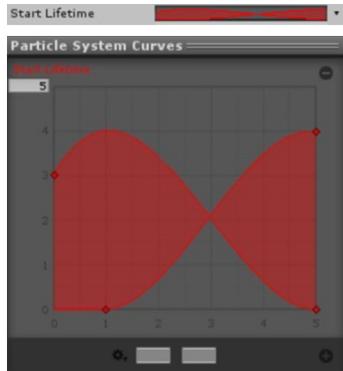
El valor de esta propiedad será definido de manera aleatoria cuando se cree la partícula con un valor comprendido entre esas dos constantes.

Curva:



El valor de esa propiedad irá cambiando según el tiempo que lleve emitiendo el sistema.

Random entre dos curvas:



El valor de la propiedad se definirá aleatoriamente entre los valores que marquen las curvas según el tiempo que lleve emitiendo el sistema.

También tenemos otro tipo de propiedades opcionales, que además suelen ser también más complejas:



- «Emission»: Define el ritmo de emisión de las partículas. Aparte del ritmo de emisión constante, también se pueden añadir «ráfagas» cada cierto tiempo.
- «Shape»: Identifica la forma de nuestro emisor y por qué zonas y cómo nacerán las partículas según esa forma.
- «Velocity/Color/Size/Rotation over Lifetime»: Define cómo se modificarán estos valores a lo largo del tiempo de vida de cada una de las partículas.
- «Inherit Velocity»: Hará que se tenga en cuenta la velocidad que pueda tener el emisor a la hora de modificar la velocidad de las partículas que genere.
- «Color/Size/Rotation by Speed»: Según la velocidad de la partícula, cambiará la propiedad indicada.
- «External Forces»: Cuando estén activadas las partículas, se verán afectadas por el viento de la escena.
- «Collision»: Define si las partículas colisionarán con el escenario o con ciertos planos colocados para que los cálculos sean más baratos.
- «Texture Sheet Animation»: Permite animar la textura de cada una de las partículas.

- «Renderer»: Aquí podemos modificar todas las propiedades del sistema. Por ejemplo, cambiar su material, definir el orden en el que se pintan las partículas o si proyectarán sombra.

2.2. Código

Una vez creados, implementar los sistemas de partículas en nuestros juegos es una tarea relativamente sencilla. Así, por ejemplo, si tuviéramos un sistema que fuera una explosión, tendríamos una referencia al sistema de partículas «ParticleSystem», activaríamos el sistema con «Play» y una vez terminado lo destruiríamos.

```
1. void Explode()
2. {
3.     ParticleSystem explosion = GetComponent<ParticleSystem>();
4.     explosion.Play();
5.     Destroy(this.gameObject, explosion.time);
6. }
```

También podemos consultar otras funciones interesantes, como:

- «isAlive()», para saber si ha acabado de ejecutarse.
- «Pause()», que se encarga de pausar la simulación y el *update* de todas las partículas.
- «Stop()», para por completo el sistema y lo resetea, o solamente deja de emitir pero las partículas ya vivas siguen actualizándose.

3. Sonido

Nos falta algo cuando jugamos un juego que no tiene sonido. El sonido es esencial para dar *feedback* al jugador e informarle de lo que está sucediendo en pantalla. O para ambientar: el crepitar de una antorcha encendida te hace sentir que estás dentro de una mazmorra o en las profundidades de una gruta.

Veamos cómo Unity gestiona el tema del sonido con un poco más de detalle que en los módulos anteriores.

3.1. «Audio Clip»

Un «Audio Clip» es la representación dentro de Unity de un archivo de sonido. El sistema nos permite importar archivos de tipo AIFF, WAV, MP3 u Ogg de manera bastante sencilla.

Si seleccionamos el archivo en el proyecto, podremos modificar sus características. Por ejemplo, si lo queremos mono o normalizado, o si queremos que se cargue en segundo plano. Esto es especialmente útil porque normalmente cuando instanciamos el archivo de sonido por primera vez puede pegar un pequeño tirón de rendimiento al juego al tenerlo que cargar en ese momento. Especialmente si se trata de archivos largos como podrían ser músicas.

Por otro lado, por cada plataforma que tengamos, podremos tener una configuración de calidad distinta para ese sonido. En móviles quizá querremos que el sonido no suene tan bien y ocupe menos en memoria que en la versión de consola.

3.2. «Audio Source»

El sistema funciona de la siguiente manera. En la escena debe haber un objeto que contenga el componente «AudioListener». Normalmente, cuando creamos una cámara este componente viene añadido por defecto.

Este «AudioListener» será el encargado de hacer de «micrófono» en la escena y escuchará todos los sonidos que se emitan en ella. El encargado de emitir esos sonidos es el componente «Audio Source». Cada sonido que se escuche en la escena debe provenir de un «Audio Source».

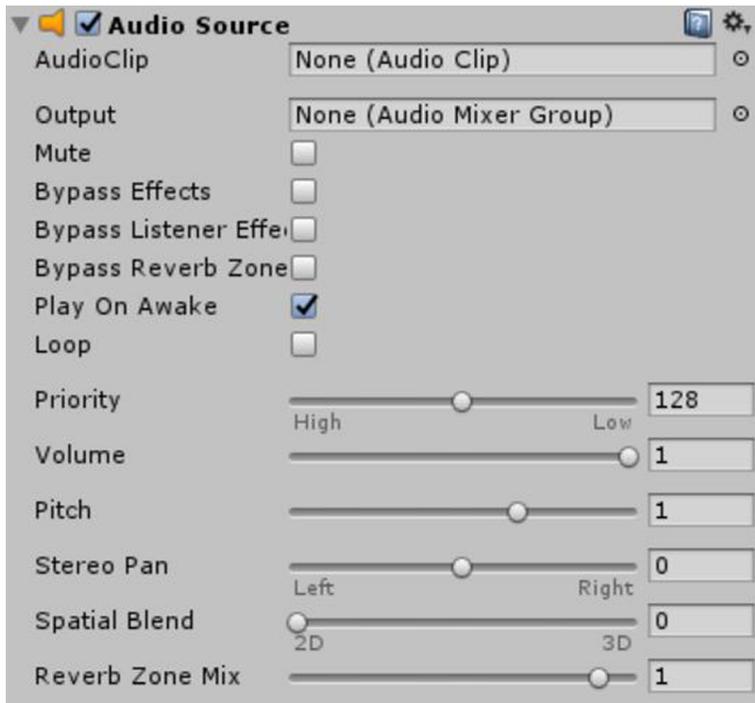
Sus propiedades más importantes son las siguientes:



Propiedades del AudioClip

Nota

Por desgracia, Unity solo permite tener un «AudioListener» activo a la vez. Esto nos obligará a desactivar o eliminar todos aquellos que estén en escena que no queramos utilizar.

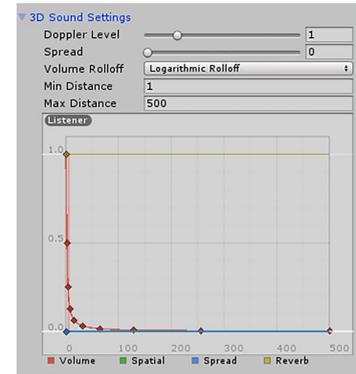


- «AudioClip»: Aquí añadiremos el sonido que queremos reproducir.
- «Output»: Aquí se define el «Audio Mixer» al que queremos asociar este sonido. Luego entraremos en más detalle sobre qué es y cómo funciona el «Audio Mixer».
- «Mute»: Si queremos silenciar el sonido. Normalmente activaremos o desactivaremos esta opción desde código.
- «Bypass Effects / Listener Effects / Reverb Zones»: Si los activamos, el sonido se ejecutará sin que se le apliquen los filtros seleccionados.
- «Play on Awake»: Normalmente viene activada por defecto. Si está activa, el sonido se ejecutará en cuanto se active el *GameObject*. Esto tiene sentido para sonidos de ambiente o músicas de fondo, pero no para acciones del *player*, por ejemplo.
- «Loop»: Si queremos que el sonido se ejecute en bucle.
- «Volume»: El volumen entre 0 y 1, de nuestro audio.
- «Pitch»: Cuanto más alto sea, más agudo será el sonido. Y, al contrario, cuanto más bajo, más grave.
- «Stereo Pan»: Por qué altavoz queremos que suene más nuestro audio.
- «Spatial Blend»: Esta propiedad es importante porque definirá el tipo de sonido que es, si un sonido 2D que se escuchará igual esté donde esté, o si

un sonido 3D en el que realmente importará la distancia que haya entre el «Audio Source» y el «Audio Listener».

Cuando el sonido es 3D nos aparecen las siguientes opciones:

- «Doppler Level»: Cuánto simulará el efecto Doppler este sonido cuando esté en movimiento cerca de un «Audio Listener».
- «Min/Max Distance»: Marca la distancia mínima a la que empezará a oírse (debería ser 0 normalmente), así como la distancia máxima.
- «Volume Rolloff»: Marca cómo será la gráfica de la caída de volumen del sonido según esté colocado el «Audio Listener» entre la distancia mínima y máxima. Podemos editar la caída con la gráfica inferior.



Propiedades del sonido 3D

3.3. Efectos y filtros

Si queremos que desde el punto de vista sonoro nuestro juego tenga un resultado más profesional, siempre podemos añadirle filtros tanto a cada uno de los «Audio Sources» como al «Audio Listener» para que se aplique a nivel general.

Unity nos ofrece varios tipos:

- «Audio Echo Filter»: Crea efecto de eco.
- «Audio Chorus Filter»: Simula varias voces de los mismos sonidos que oímos con un ligero desfase entre ellas.
- «Audio Distorsion Filter»: Genera distorsión.
- «High/Low Pass Filter»: Deja pasar solamente las frecuencias más altas o más bajas a partir del *cut off* de cada uno.
- «Audio Reverb Filter»: Crea un efecto de reverberación personalizado.

También tenemos el componente «Reverb Zones», que crea un espacio en el que se va aplicando más o menos reverberación según entremos o salgamos de la zona. Un ejemplo de su uso sería que al entrar en una cueva haya poca reverberación al inicio pero bastante en su interior.

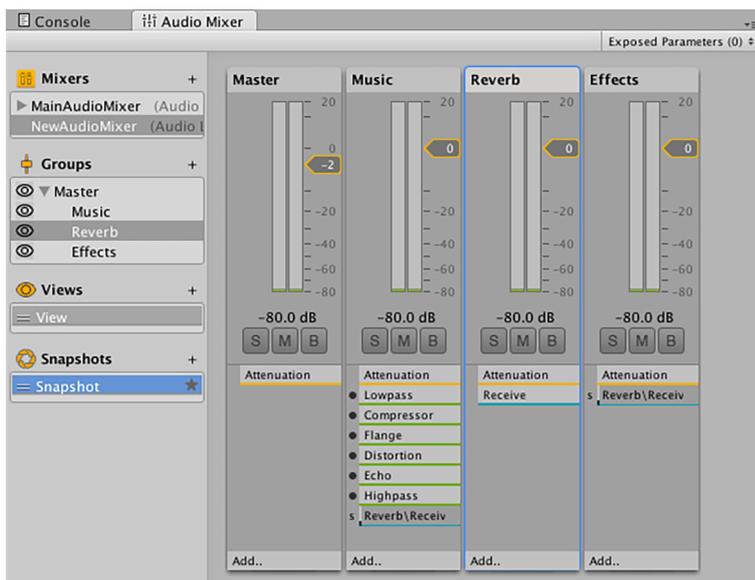
3.4. «Audio Mixer»

Cuando un proyecto empieza a tener un tamaño mayor, suele haber muchos elementos de sonido y hay situaciones en las que, si tuviéramos que gestionar por código todos los sonidos que tenemos o sus volúmenes, sería realmente tedioso.

Un ejemplo de ello sería si queremos ir a la pantalla de pausa y que, cuando se active, el volumen general de los efectos y de la música baje un poco, pero el de los elementos del menú esté al 100 %. O la típica opción de poder subir o bajar el volumen de los efectos, por un lado, y el de la música, por otro.

Para ello, Unity nos ofrece la herramienta «Audio Mixer». Crearemos uno clicando con el segundo botón en la ventana «Project (pestaña «Explorador del Project») > Create > Audio Mixer».

El «Audio Mixer» no se instancia dentro de la escena, sino que se queda dentro de nuestro proyecto. Podemos editar sus características abriendo la ventana que los gestiona en «Window > Audio Mixer».



«Mixers»: En el campo «Mixers» se listan todos los «Audio Mixers» que se han creado. Dándole al icono «(+)» podríamos haber creado uno también.

«Groups»: Aquí se lista la jerarquía de «AudioGroups» dentro del «AudioMixer» seleccionado. Un «AudioGroup» es donde se agruparán un conjunto de «Audio Sources» para así poder modificarlos en conjunto.

Nota

Cada uno de los grupos de un «AudioMixer» lo podemos configurar con las siguientes propiedades:

- «Volumen»: En este caso se medirá en decibelios (dB).

- «Solo»: Si hay algún grupo marcado como «Solo», serán solo los sonidos de este grupo los que estarán sonando.
- «Mute»: Silencia el grupo.
- «Bypass»: Hace caso omiso de todos los efectos que tuviera aplicados.
- «Effects»: A cada grupo se le pueden añadir múltiples efectos que podrán ser modificados desde el Inspector cuando seleccionemos el grupo.

Por ejemplo, podríamos tener el grupo «Master», que agrupa todos los audios del juego. Luego tener uno «Music» y otro «FX». Y dentro de «FX» podríamos tener «efectos ingame» y «efectos menús».

Todos estos grupos aparecen listados en detalle en la parte central de la ventana, en la que cada uno de los grupos tiene unos valores asignados que no tienen por qué coincidir con los de los demás. Por ejemplo, podremos tener el grupo «Music» con un volumen mucho más bajo que el grupo «efectos ingame» porque la música solo está para dar ambiente. Si al grupo «Master» le subimos o bajamos su volumen, todos los demás grupos hijos subirán o bajarán su volumen en consecuencia.

«Snapshopts»: Los «Snapshopts» son *presets* donde guardaremos unos valores concretos para todos los grupos. Por ejemplo: podríamos crear un «Snapshot» llamado «Pause» donde bajáramos los volúmenes de los grupos «Music», «ingame fx», pero dejáramos al 100 % el «Menus FX». Así, cuando le demos al «pause» en nuestro juego, tan solo tendremos que decirle al «AudioMixer» que asigne los valores del «Snapshot» «Pause» a todos los grupos.

3.5. Código

Una vez que hemos visto cómo funcionan las bases del sonido dentro de Unity, solo nos queda ver cómo acceder a todas estas funcionalidades desde nuestro código.

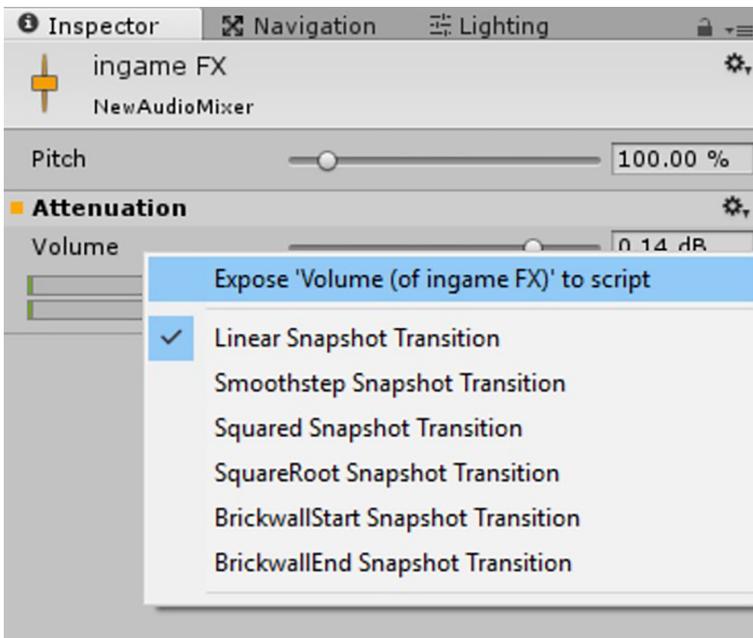
```
1. public UnityEngine.Audio.AudioMixer myAudioMixer;
```

Primero crearemos una variable de tipo «AudioMixer» para poder acceder a ella.

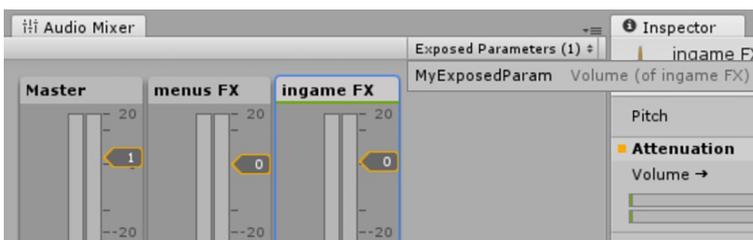
Lo más sencillo sería hacerla pública en un *script* y arrastrarle el «AudioMixer» para tenerlo referenciado.

Por ejemplo, vamos a cambiar el volumen de los «ingame FX» desde código. Para ello, crearemos una variable llamada «ingame_fx_volume» que haga referencia al volumen de ese grupo.

La manera de hacer públicas estas variables es realmente antiintuitiva y es de las pocas veces que Unity hace algo tan enrevesado. Para crear esa variable, deberemos seleccionar el grupo «ingame FX» y en el Inspector hacer clic derecho encima del campo que queramos exponer. En este caso el volumen. Y seleccionamos «Expose ...».



Esto nos habrá añadido en la parte superior derecha de la pestaña del «Audio Mixer» («Exposed Parameters») una nueva variable. Podemos seleccionarla y con el segundo botón del ratón podremos cambiar su nombre a «ingame_fx_volume» tal como habíamos dicho.



Una vez expuesta, podremos modificarla por código con la siguiente función:

```
1. myAudioMixer.SetFloat("ingame_FX_volume", -13.0f);
```

En el caso de que quisiéramos asignar a todos los grupos los valores de un «Snapshot», deberíamos hacer lo siguiente:

```
1. public UnityEngine.Audio.AudioMixer myAudioMixer;
2. public float timeToChangeSnapshot = 1.0f;
3.
4. private UnityEngine.Audio.AudioMixerSnapshot[] snapshotArray;
5. private float[] snapshotWeightsArray;
6.
7. void SetSnapshot(string snapshotName,
8.                 float timeToChangeIt,
9.                 float snapshotWeight = 1.0f)
10. {
11.     UnityEngine.Audio.AudioMixerSnapshot snapshot =
12.         myAudioMixer.FindSnapshot(snapshotName);
13.
14.     snapshotArray = new UnityEngine.Audio.AudioMixerSnapshot[1];
15.     snapshotArray[0] = snapshot;
16.
17.     snapshotWeightsArray = new float[1];
18.     snapshotWeightsArray[0] = snapshotWeight;
19.
20.     myAudioMixer.TransitionToSnapshots(snapshotArray,
21.                                       snapshotWeightsArray,
22.                                       timeToChangeIt);
23. }
```

La función «TransitionToSnapshots» recibe una lista de «Snapshots» y una lista de pesos. Esto es así por si quisiéramos aplicar por ejemplo un *mix* de varios *presets* y modular qué porcentaje de cada uno querriamos.

En nuestro ejemplo, como solo tenemos uno, el porcentaje es 1, ya que queremos que todos los grupos pasen a tener exactamente los valores del «Snapshot». Y le hemos dicho que la transición a esos valores dure 1 segundo.

4. Iluminación

La iluminación es uno de esos factores que se suelen pasar por alto a la hora de diseñar y decorar un nivel, pero cabe tener en cuenta que una buena iluminación puede hacer resaltar hasta el escenario más pobre.

Unity ha ido cambiando partes del sistema de iluminación prácticamente en cada actualización, y es que es un factor realmente importante.

Desde la versión 5.0, Unity introdujo *Global Illumination*, lo que supuso un cambio en cómo se gestionaba la iluminación hasta entonces.

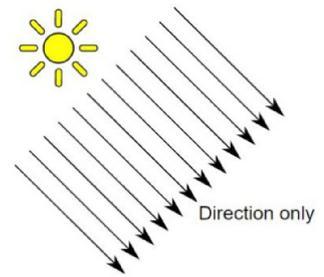
4.1. Tipos de luces

Hasta ahora habíamos estado trabajando únicamente con la luz direccional que viene por defecto añadida a la escena. Veamos ahora los cuatro tipos de luces que podemos utilizar:

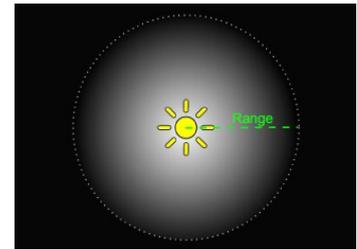
- «Directional Light»: Este tipo de luz ya viene en la escena por defecto, pero podemos añadirla desde «GameObject > Light > Directional Light». Su intención es simular la luz del sol; emite luz en una sola dirección. La podemos colocar en la posición que queramos de la escena. Solo le afectan las rotaciones. En las últimas versiones de Unity implementaron un *skybox* dinámico que hacía que a medida que rotásemos la luz direccional de la escena, el sol cambiara de posición y el cielo cambiara de color para simular atardeceres, amaneceres y la noche.
- «Point Light»: Esta luz se comporta como una bombilla, emite en todas las direcciones. Aquí no importan sus rotaciones, pero sí su posición.
- «Spot Light»: Esta luz es similar a la anterior, pero con restricciones. Realmente se comporta como un flexo o una linterna. Emite luz en forma de cono.
- «Area Light»: Este es un tipo especial de luz que no se aplica en tiempo real debido a su alto coste. Suele añadirse directamente a la información del *lightmap*. Esta luz emite en todas las direcciones a través de un lado de un rectángulo. Podríamos imaginarla como si un televisor tuviera toda la pantalla en blanco y estuviera iluminando una habitación a oscuras.

Todas heredan del componente «Light», que tiene las siguientes propiedades principales:

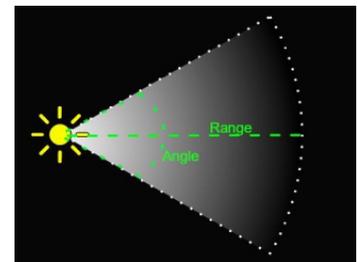
- «Type»: Define si esa luz será «directional», «point», «spot» o «area».
- «Baked»: Aquí podemos elegir si queremos que la luz se calcule en tiempo real o se precalcule en tiempo de edición para mejorar luego el rendimiento y obtener resultados de iluminación más realistas.
- «Color»: El color de la luz.
- «Intensity / Bounce Intensity»: La intensidad de la luz, y la intensidad que tendrá una vez que rebote en alguna superficie.
- «Shadow Type»: Esta propiedad define si queremos que la luz proyecte sombras o no. Recordad que este tipo de sombras suele ser costoso en cuanto a *render*. Sobre las sombras, podemos tener «Hard Shadows» (sombras con los bordes duros) o «Soft Shadows» (con los bordes difuminados). Dentro de sus características podremos indicar lo transparente que será



«Directional Light»



«Point Light»



«Spot Light»



«Area Light»

Nota

Siempre que queremos que una *mesh* proyecte sombras cuando le dé una luz, deberemos marcar en su componente «Mesh Renderer» en la casilla «Cast Shadows».



En el caso de que queramos que las sombras se pinten encima de esa *mesh*, deberemos marcar la casilla «Receive Shadows».

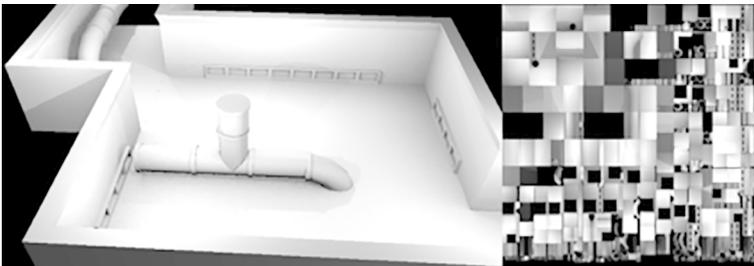
(«Strength»), y su resolución (cuanto más baja, más pixelada, pero cuanto más alta, más cara será).

Otro tipo de iluminación que también afectaría a la escena pero que no es un método de iluminación tradicional serían los materiales emisivos. Esto lo conseguimos dándole un valor mayor que 0 al valor del campo «Emisive». Y seleccionando su color. Esta iluminación emisiva de una textura afectará solamente a la geometría marcada como estática.

4.2. *Global Illumination*

Los cálculos requeridos para iluminar una escena siempre han sido caros en cualquier motor de juego. Una de las técnicas más utilizadas para mejorar el rendimiento ha sido la utilización de *lightmaps* para iluminar la geometría.

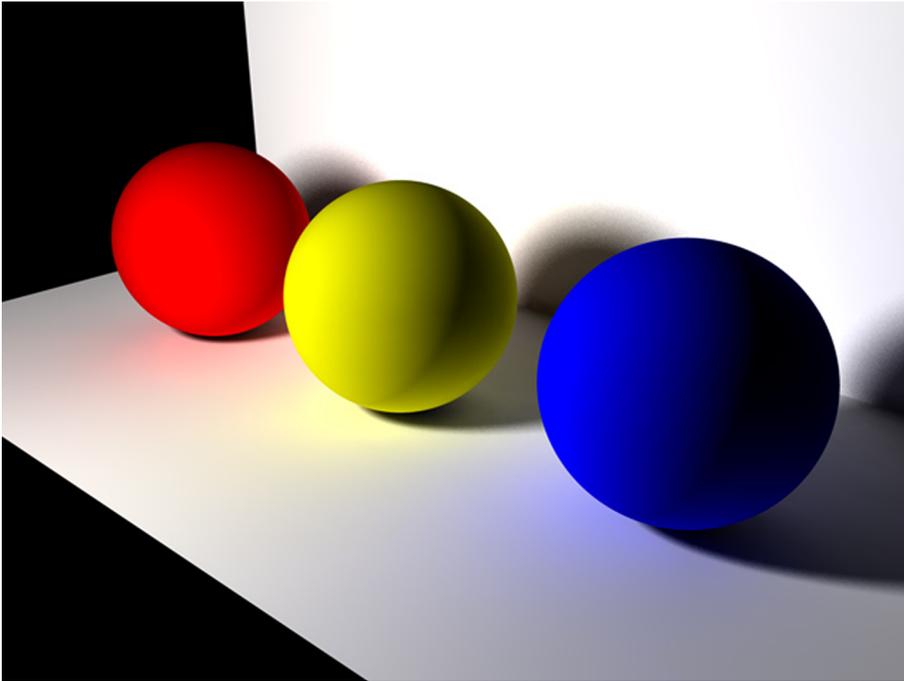
Estos *lightmaps* se calculan una vez creada la escena, posicionando todas las luces que sabemos que no se van a mover durante el juego y calculando cómo afectarán a esa geometría estática del escenario. Esa información se guarda en una textura que cuando se ejecuta el juego se multiplicará a las texturas de la geometría para que se pinten al 100 % o, por el contrario, se oscurezcan un poco para darles sombras o se tiñan de algún color en el caso de tener luces de colores.



Así pues, esta escena de la imagen está iluminada en tiempo de Editor y se ha precalculado cómo afecta la luz a toda esa geometría y generado esa textura a la derecha. Cuando ejecutemos el juego, no se hará ningún cálculo para iluminar toda esa geometría, ya que ya ha sido iluminada, con lo que se consigue así una mejora de rendimiento.

Al cálculo de *lightmaps* Unity lo llama «Baked GI». En la versión 5.0 introdujo el concepto de *Global Illumination*, que consiste en darle un poco más de realismo a la manera como se iluminan las escenas dentro de Unity haciendo que la luz rebote en las superficies.

Por ejemplo, si tenemos una bola de billar roja al lado de una pared blanca, si la iluminamos directamente, la luz rebotará en la pelota y manchará un poco de luz roja en la pared.



Este tipo de iluminación más realista llamada «Radiosity» solo se utilizaba para *renders* o vídeos CG en los que se podía tardar mucho rato para hacer el *render*. En los videojuegos estaba prohibidísima debido a sus altos costes de cálculo.

Unity implementa un sistema que llaman «Precomputed Realtime GI» y que consiste en guardar la información de cómo rebotaría la luz en una escena estática para que luego, *ingame*, al mover un objeto dinámico por ella, este se vea afectado por esos rebotes. Esto se consigue utilizando «Light Probes».

El ejemplo que utiliza Unity en su documentación lo ilustra perfectamente. Supongamos que tenemos una pared roja estática para la que hemos precalculado cómo rebotaría la luz en ella. Si ahora movemos una pelota blanca (objeto en movimiento) delante de ella, la pelota se verá teñida de rojo. Esto solo se aplica a objetos dinámicos. Si, por ejemplo, tuviéramos la pared blanca y la pelota roja, no pasaría nada, ya que los cálculos son sobre cómo afecta la pared a los objetos en movimiento, y en este caso la pelota roja se iluminaría de blanco, pero no se apreciaría.

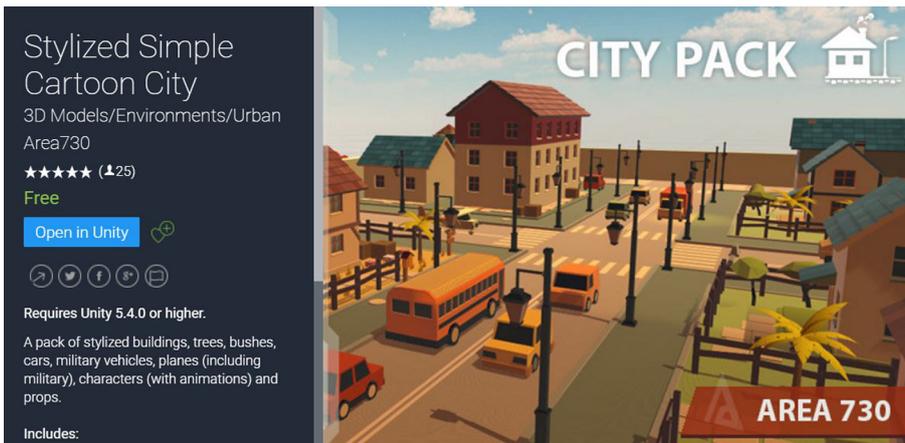
5. Proyecto: Un juego de plataformas 3D

No podíamos cerrar este módulo sin implementar el otro gran tipo de control que nos queda, un juego en tercera persona.

En este caso crearemos un pequeño pueblecito donde 16 horas transcurren en tan solo 1 minuto y medio. Cuando llegan las 12 de la madrugada, nuestro personaje muere, así que tendrá que ir recolectando objetos que volverán el reloj a las 8 AM.

5.1. El escenario

Para crear nuestro pueblecito, volveremos a utilizar recursos gratuitos de la Assets Store. Usaremos el paquete *Stylized Simple Cartoon City*, que podréis descargar en este enlace: <https://www.assetstore.unity3d.com/en/#!/content/50095>.



En este *pack* encontraremos casas, carreteras, coches, autobuses, camiones, farolas, árboles, piscinas, personas..., es decir, lo básico para crear un pequeño pueblo.

En la ruta «Assets > Area730 > Stylized city > Scenes» encontraremos dos escenas. Una con varios de los *assets* (pero no todos) bien organizados y otra llamada «Town» con un ejemplo de pueblo ya creado. En nuestro caso duplicaremos la escena «Town», y trabajaremos sobre esta copia.

Tened en cuenta que los tamaños de este escenario son bastante grandes, por lo que os recomendamos seleccionar todo el escenario, hacerlo hijo de un *GameObject* padre llamado «Town» y reducir la escala de este *GameObject* hasta que tenga unas medidas razonables (unas seis veces más pequeño).

Como el pueblo está en medio del vacío, crearemos un «Terrain» para que cuando la cámara mire hacia fuera del pueblo veamos las montañas y algunos árboles.

Por este pueblecillo correrá y saltará nuestro personaje.



Reto 1

El pueblecito que viene por defecto es bonito pero pequeño y con varias zonas que no nos interesan. Sería interesante crear un escenario un poco más grande y con elementos colocados de tal manera que podamos acceder con el personaje a cualquier zona. Por ejemplo, deberíamos poder subir a los tejados, así que podríamos colocar casas más pequeñas cerca o aviones torcidos, para poder ir saltando hasta poder llegar.

Como el paso del tiempo es un punto importante en cómo se ve el escenario, vamos a empezar por aquí. Cojamos la luz direccional, coloquémosla en un lugar donde no nos moleste y asignémosle una rotación de (100,30,0) para ir trabajando.

Esta luz nos hará de sol, así que vamos a indicarle a Unity que será ella la que marque la posición del sol. Para ello, iremos a «Window > Lighting», le diremos que nuestro *skybox* sea el «Default Skybox» y arrastraremos nuestra luz direccional al campo «Sun». Ahora, cuando rotemos la luz, cambiará el *skybox* y la posición del sol.

Para controlar esto crearemos un *GameObject* vacío y lo llamaremos «Day-NightManager»; le añadiremos el siguiente *script* con el mismo nombre.

```

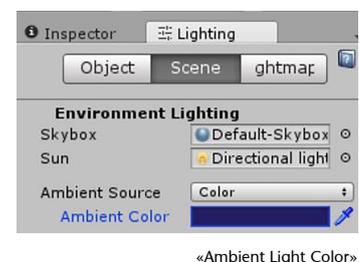
1. public class dayNightManager: MonoBehaviour
2. {
3.     public GameObject directional;
4.     public float totalSecondsTo16hours = 90;
5.     public float startDayRotationX = 0;
6.     public float endDayRotationX = 270;
7.     private float elapsedTime = 0;
8.
9.     void Start() { Reset(); }
10.
11.    void Update()
12.    {
13.        elapsedTime += Time.deltaTime;
14.
15.        float rot = (endDayRotationX - startDayRotationX)
16.                    / totalSecondsTo16hours;
17.        directional.transform.Rotate(-rot*Time.deltaTime,0,0);
18.
19.        // Debug
20.        if (Input.GetKeyDown(KeyCode.R))
21.            Reset();
22.    }
23.
24.    void Reset()
25.    {
26.        totalSecondsTo16hours--;
27.        elapsedTime = 0;
28.        directional.transform.rotation = Quaternion.Euler(
29.            startDayRotationX,
30.            directional.transform.rotation.eulerAngles.y,
31.            directional.transform.rotation.eulerAngles.z);
32.    }
33. }

```

- «directional» es la referencia a la luz direccional que tendremos que arrastrarle desde la escena.
- «totalSecondsTo16Hours» es el rato en segundos que tardarán en pasar esas 16 horas.
- «startDayRotationX/endDayRotationY» indican el valor inicial y final de la rotación de la luz direccional cuando son las 8 h y cuando son las 12 h.

La función «Reset» deja el tiempo transcurrido a 0 y vuelve a colocar la luz en la rotación original. Y resta un segundo al tiempo total de juego para que cada vez sea más difícil conseguir el siguiente reloj. «Update» va sumando el tiempo transcurrido a la variable «elapsedTime» y va rotando a cada *frame* la parte proporcional de la rotación total que ha de hacer.

Ahora que ya tenemos un *script* que simula el paso del tiempo, como hay momentos del juego en los que será de noche, iluminaremos un poco la escena. La iluminación básica pasará por cambiar el color de la «Ambient Light». Esta propiedad la cambiaremos en la pestaña «Lighting» cambiando la «Ambient Source» a «Color» y seleccionando un color azul oscuro como luz de ambiente.



Así, aunque no haya nada de luz, veremos un poco porque todo en vez de estar negro tendrá una tonalidad azul un poco clara, como sucede en las películas.

Reto 2

Si queremos que el juego tenga mucha más vida, deberíamos iluminar la escena añadiendo diferentes tipos de luces. Por ejemplo, las farolas deberían ser «Spot Lights», sus cristales tener un material emisivo, o algunas de las ventanas podrían estar iluminadas con un «Area light», por poner algunos ejemplos.

Reto 3

Si ya queremos hacerlo bien de verdad, lo ideal sería que a partir de cierta hora se vayan activando las luces o encendiendo las ventanas y los faros.

5.2. El personaje

Ahora que ya tenemos el escenario, vamos a añadir nuestro personaje.

En el módulo anterior utilizamos el «First Person Controller», y en este utilizaremos el «Third Person Controller» que nos ofrece Unity. Para ello, recordad que hay que añadir el *package* desde «Assets > Import Package > Characters». Una vez añadido al proyecto, deberemos acceder a su *prefab* desde la carpeta «Assets > Standard Assets > Characters > ThirdPersonCharacter > Prefabs > ThirdPersonController.prefab» y añadirlo a nuestra escena.



Una vez que lo coloquemos donde queramos que empiece la partida y le cambiemos el tamaño hasta adecuarlo al escenario, si le damos al «Play» veremos que el personaje caerá infinitamente. Esto sucede porque el escenario no tiene «Colliders». Deberemos ir colocando en la escena los «Colliders» correspondientes. Para ir más rápido, podéis seleccionar toda la geometría y en el Inspector añadir a todos a la vez un «Mesh Collider».

Reto 4

Utilizar «Mesh Colliders» es costoso, por lo que prácticamente nunca se usan en juegos a no ser que sea estrictamente necesario o se los añadamos a alguna *mesh* específica. Lo normal y lo que deberemos hacer es utilizar «BoxColliders» o «SphereColliders» para encapsular partes de la geometría.

Ahora que nuestro personaje ya tiene un suelo por donde moverse, si ejecutáis veréis que el control no está adaptado a ninguna cámara. Vamos a añadirle una cámara de rotación libre que es bastante estándar en juegos en tercera persona.

Para ello, añadiremos el *package* de cámaras desde «Assets > Import Package > Cameras». Una vez añadido al proyecto, arrastraremos a la escena el *prefab* «Assets > Standard Assets > Cameras > Prefabs > FreeLookCameraRig.prefab».

Este *prefab* contiene un *GameObject* padre con los *scripts* que controlan el movimiento de la cámara, un hijo que es el pivote sobre el que girará nuestra cámara y la cámara como hija del pivote. Este pivote normalmente lo colocaremos en la cabeza o en el torso del personaje para que la cámara pueda rotar sobre él y lo vaya persiguiendo como el control de un GTA.

Tanto el «Third Person Controller» como el «Free Look Camera Rig» están pensados para trabajar conjuntamente. Tan solo hemos de seleccionar el «Third Person Controller» y ponerle el *tag* «Player» para que la cámara automáticamente se coloque detrás de él.

Para colocar bien el pivote y la cámara, lo mejor es poner los dos *prefabs* en el mismo punto y recolocar el pivote y la cámara a la altura y la distancia deseada.

Veréis que la cámara tarda mucho en seguir al personaje. Esto es porque debemos adaptar las velocidades del *script* «Free Look Cam» del «Free Look Camera Rig».



Nota

El *script* «Free Look Cam» nos permite controlar la velocidad de la cámara para perseguir al *player*, la velocidad de giro y el ángulo máximo y mínimo de inclinación de esta para no dar la vuelta verticalmente.

Nota

Recordad que deberíamos exportarlo «fbx for Unity» y una vez importado cambiarle el avatar de «Generic» a «Humanoid» para que podamos utilizarlo con cualquier set de animaciones.

Ahora que ya tenemos el control del personaje adaptado a nuestras necesidades, vamos a cambiar su modelo. Para ello, volveremos a utilizar la web Mixamo para descargarnos un personaje acorde con este tipo de escenario y estilo 3D. Para este ejemplo hemos elegido el que se llama «Ty»:



Este modelo viene sin animaciones, por lo que vamos a utilizar las que ya vienen con el «ThirdPersonController». Para que el *script* sepa quién es nuestro personaje y detecte su «Rig», hemos de pasarle el avatar de Ty que está dentro de su FBX al Animator del «ThirdPersonController».

Si hemos hecho bien todos los pasos, ya deberíamos tener a Ty paseando y saltando por nuestra ciudad con la cámara de rotación libre.



Reto 5

Ahora que ya tenemos el personaje y el control listos, deberíamos encontrar los ajustes adecuados de velocidad, salto, posición de la cámara, velocidad de giro, etc.

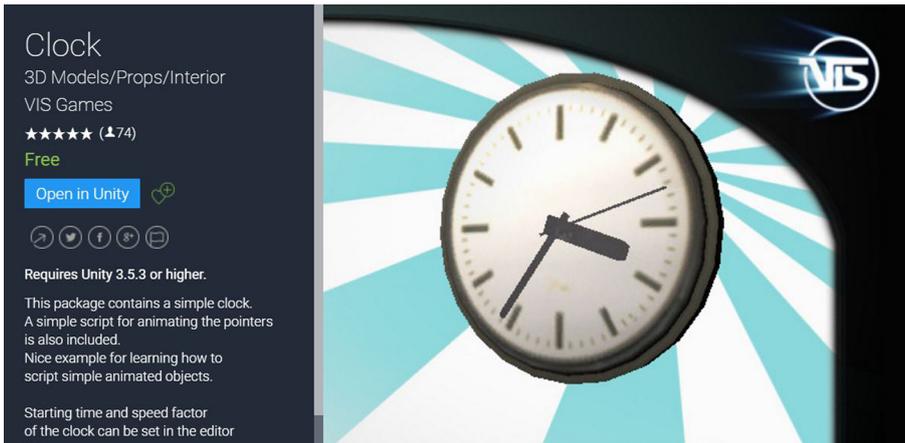
Nota

Este reloj viene con un *script* que permite que las agujas giren. Si le ponemos velocidad 10, girará más rápido y se adecuará a nuestro juego, en el que el tiempo pasa rapidísimo.

5.3. El reloj

Ya casi estamos, tan solo nos queda crear el ítem que resetee el tiempo y que hace que Ty no muera en el intento. Este ítem, una vez lo recojamos, reaparecerá en otro lugar del escenario a la espera de ser recogido.

Como elemento recolectable utilizaremos un reloj que podremos descargar en la Store o desde este enlace: <https://www.assetstore.unity3d.com/en/#!/content/4250>.



Cuando tengamos el reloj en la escena, le añadiremos un *script* que lo hará girar sobre sí mismo. Eso lo conseguiremos con esta simple línea:

```
1. this.transform.Rotate(0, 360*Time.deltaTime/rotationTime, 0);
```

Y para darle un poco más de vida, le añadiremos dos sistemas de partículas: uno para resaltar que es un ítem importante, y otro para que lo veamos una vez el ítem cambie de lugar.

Como ayuda tendremos dos *packs* de partículas en los que poder elegir y tener una base para empezar a modificarlos y que se adapten a nuestras necesidades. Son los siguientes:

- *KY Magic Effects Free*: <https://www.assetstore.unity3d.com/en/#!/content/21927>.
- *Particle Ribbon*: <https://www.assetstore.unity3d.com/en/#!/content/42866>.

KY Magic Effects
Free

Particle Systems/Magic
Kakky

★★★★★ (1199)

Free

[Open in Unity](#)

Requires Unity 4.2.2 or higher.

Unity5 support!
This package includes 10 game effects! This is the free version!!
Uses the system of shuriken mainly.
Also include primitive types reduced [Draw calls] for mobile.



Magic Effects Free

Particle Ribbon

Particle Systems/Magic
Moonflower Carnivore

★★★★★ (184)

Free

[Open in Unity](#)

Requires Unity 5.2.3 or higher.

[YouTube Video](#)

[Forum thread](#)

v1.05: Unity 5.5-exclusive new effects:
Charge_01.6, Drag_01 Warp-strike,
Liberate_02.1.2 Smoke, Projectile_01,
Spiral_02.1.3



PARTICLE RIBBON

<http://u3d.as/lqz>

Moonflower Carnivore

Reto 6

Modificad alguno de los sistemas para crear otro que nos permita ver dónde ha *res-pawneado* el reloj desde lejos. Lo ideal es crear algo parecido a una columna de partículas, o que algunas suban muy alto para que se vean por detrás de los tejados.

Y ahora a por la parte interesante del ítem, el *script* que controla si el *player* entra dentro de él, resetea el tiempo, y lo recoloca en otra parte del escenario listo para que el *player* lo vuelva a buscar. Sería el siguiente:

```

1. public class dayNightItem: MonoBehaviour
2. {
3.     public dayNightManager timeManager;
4.     public float rotationTime = 2.0 f;
5.     public Transform gameAreaCenter;
6.     public float gameAreaSizeX = 50;
7.     public float gameAreaSizeZ = 50;
8.
9.     void Start() {
10.         Respawn();
11.     }
12.
13.     void Update()
14.     {
15.         this.transform.Rotate(0,
16.                               360*Time.deltaTime/rotationTime,
17.                               0);
18.     }
19.
20.     void OnTriggerEnter(Collider col)
21.     {
22.         if (col.tag == "Player")
23.             Respawn();
24.     }
25.
26.     void Respawn()
27.     {
28.         this.transform.position = new Vector3(
29.             gameAreaCenter.position.x + Random.Range(-gameAreaSizeX/ 2.0f,
30.                                                       gameAreaSizeX/ 2.0f),
31.             50,
32.             gameAreaCenter.position.z + Random.Range(-gameAreaSizeZ/2.0f,
33.                                                       gameAreaSizeZ/2.0f));
34.
35.         RaycastHit hit;
36.         if (Physics.Raycast(new Ray(this.transform.position,
37.                                     Vector3.down),
38.                             out hit,
39.                             100 f))
40.             this.transform.position = hit.point;
41.
42.         timeManager.Reset();
43.     }
44. }

```

Nota

«dayNightManager» es una referencia al *mánager* que controla el tiempo y la luz, que deberemos arrastrar de la escena al Inspector.

«rotationTime» será el tiempo, en segundos, que tardará en dar una vuelta completa nuestro reloj.

«gameAreaCenter» es una referencia a un *GameObject* vacío que debe estar en el centro del área en la que queramos que vayan apareciendo relojes.

«gameAreaSizeX / gameAreaSizeZ» es el tamaño en el eje *X* y *Z* de esa área.

Lo primero que hace el «Start» es recolocar el reloj en una zona aleatoria.

La función «Respawn» coloca el ítem en una posición aleatoria dentro del área de *spawn* según los valores que le hayamos dado al *script*. A una altura de 50 metros.

Luego lanza un rayo desde su base 100 metros hacia abajo y mira si colisiona con algo. En el punto que colisione, lo coloca. Así nos evitamos que pueda aparecer debajo de un puente o dentro de alguna geometría. Por último, le dirá al *mánager* que resetee el tiempo y la luz para que volvamos a empezar.

«Update» va rotando el reloj sin parar, y «OnTriggerEnter» *respawneará* el ítem cuando el *player* lo toque.

5.4. El HUD

Para tener la funcionalidad completa, tan solo nos quedaría crear un HUD que nos muestre los minutos que nos quedan restantes de día antes de que lleguen las 12 h de la noche. Para ello, nos apoyaremos en la fuente gratuita *Lecompte Free Font*, que encontraremos en la web: <https://www.assetstore.unity3d.com/en/#!/content/72173>.



Esta fuente contiene imágenes en lugar de letras, lo que nos permitirá crear un HUD adecuado a nuestro juego de manera muy rápida con tan solo un campo de texto.



Nota

Al poner esta fuente en un campo de texto, si escribimos la letra *E* saldrá el «símbolo AM», o si ponemos la letra *G* saldrá el icono de un reloj. Id probando las diferentes opciones a ver si alguna os gusta más para vuestra versión del juego.

Reto 8

¡No os olvidéis de hacer el «Game Over» cuando lleguemos a las 12 h de la noche!

Reto 7

Colocar directamente los minutos que nos van quedando hasta las doce es sencillo. Tan solo hay que colocar en el campo de texto el contenido de los minutos restantes («totalSecondsTo16hours - elapsedTime») con la información contenida en el «day-NightManager». Lo interesante sería mostrar la hora en la que estamos en formato reloj. Que empiece a las 8 h AM y termine a las 12 h PM.

Una vez que tenemos el HUD creado, el tiempo corriendo, los relojes autogenerándose y a Ty corriendo y saltando por el escenario..., ¡por fin tenemos el juego completado!

5.5. Soluciones a los retos propuestos

Reto 1

Aquí cada uno tiene libertad para crear el escenario que más le guste. Incluso añadir *assets* de otros *packs* de la Store. La parte interesante es buscar situaciones de escenario con las que facilitar al *player* acceder a cualquier parte de la escena.

Este sería un ejemplo de cómo poder subir a un tejado utilizando los *assets* que ya tenemos:



Reto 2

Lo más inteligente sería tomar la geometría de la farola y hacer un *prefab*. Y este *prefab* volver a colocarlo substituyendo las farolas actuales.

A este *prefab* le añadiríamos una «SpotLight» debajo del farolillo, con una ratio ancha y bastante «Range» para que ilumine lo suficiente. También podemos, si queremos, añadir que proyecte sombras duras, ya que habrá bastantes y es mejor así porque consumirán menos. También podríamos cambiar el material del cristal de los farolillos («Object012») y crear uno nosotros que tenga el color emisor a 1, para que dé la sensación de que está iluminado.

Y lo mismo para las ventanas de las casas, los cristales o los faros de los coches. A estos últimos también podemos añadirle un par de «Spotlights» para cuando sea de noche.



Reto 3

Lo primero que deberemos hacer es crear una variable llamada «minutesLeft» en nuestro «dayNightManager», que iremos actualizando según vaya avanzando el día para que todos los elementos de la escena sepan cuánto tiempo nos queda para «morir».

```
1. secondsLeft = totalSecondsTo16hours - elapsedTime;
```

Y luego creamos un *script* que mire el tiempo transcurrido, cambie automáticamente su material y encienda la luz o la apague según le corresponda.

```
1. public class LightActivator: MonoBehaviour
2. {
3.     public Material normalMaterial;
4.     public Material lightedMaterial;
5.     public GameObject myLights;
6.     public dayNightManager dayNight;
7.     public float minutesLeftToActivate = 30;
8.
9.     private MeshRenderer myRenderer;
10.
11.     void Start() {
12.         myRenderer = GetComponent < MeshRenderer > ();
13.         TurnOff();
14.     }
15.
16.     void Update() {
17.         if (dayNight.secondsLeft <= minutesLeftToActivate)
18.             TurnOn();
19.     }
20.
21.     void TurnOn() {
22.         myLights.SetActive(true);
23.         myRenderer.material = lightedMaterial;
24.     }
25.
26.     void TurnOff() {
27.         myLights.SetActive(false);
28.         myRenderer.material = normalMaterial;
29.     }
30. }
```

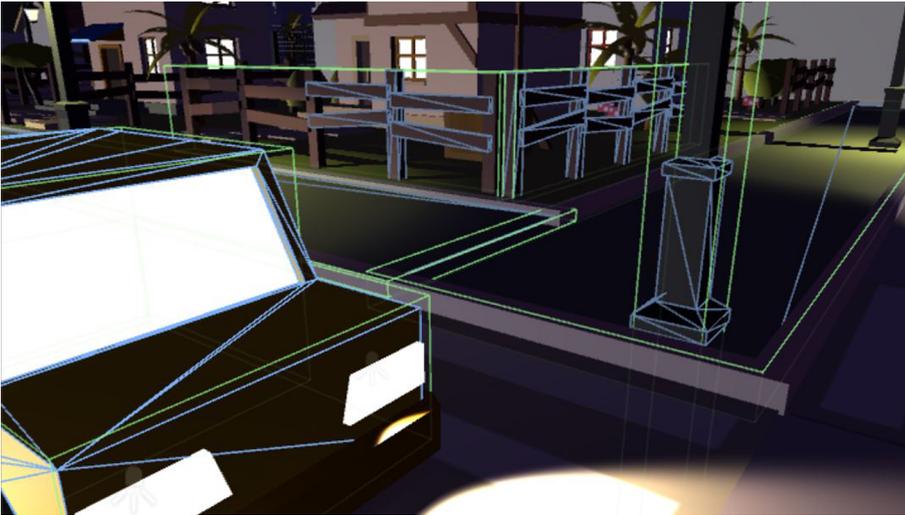
Para darle más variedad a nuestro juego, en el Inspector pondremos un valor distinto a «minutesLeftToActivate» para cada casa, coche, farola, etc., y así se irán activando progresivamente.

Podríamos optimizar un poco este código haciendo que solamente se haga «TurnOn» o «TurnOff» una vez, controlándolos con un par de booleanos y que no se estén ejecutando a cada *frame* con el consiguiente coste innecesario.

Reto 4

Aunque es un trabajo tedioso, es necesario si queremos que nuestro juego se ejecute al máximo *frame rate* posible. Cosas como las farolas o las vallas se pueden simplificar directamente en un solo «BoxCollider». Otras, como los coches o las casas, necesitarán más de un «BoxCollider» para darle la forma.

Y para otros elementos como los tejados, podremos hacer que tengan un «meshCollider» porque su geometría es sencilla y porque como nos subiremos por ellos no queremos tener complicaciones con «Colliders» mal colocados.



Reto 5

Esta parte no tiene una respuesta fija. Cada jugador tiene un estilo y unos gustos distintos, aunque más o menos todos sabemos ver si algo «no encaja». El salto aquí será importante tenerlo bien balanceado en cuanto a fuerza para que podamos llegar a saltar encima de cajas, coches y demás elementos.

Reto 6

En nuestro caso hemos utilizado el sistema «Liberate_01» y hemos modificado su velocidad *over lifetime* en *Y*, cambiando la gráfica para que suba a más de 30 unidades. Cambiado el color de los tres sistemas que lo componen de blanco a azul para que se asemeje al sistema de rayos que tiene el reloj en su centro para destacarlo, y puestas las trazas anaranjadas. También hemos aumentado el número de partículas que emite para que tenga más intensidad.



Reto 7

```
1. void SetText()
2. {
3.     float secondsXHour = totalSecondsTo16hours / 16.0 f;
4.     int hour = (int)((totalSecondsTo16hours - secondsLeft)
5.                     / secondsXHour);
6.     int minutes = (int)((((totalSecondsTo16hours - secondsLeft)
7.                            / secondsXHour) - hour) * 60);
8.     string label;
9.
10.    if ((hour + 8) > 12)
11.        label = (hour + 8 - 12).ToString("00") + ":" +
12.                minutes.ToString("00") + "F G";
13.    else
14.        label = (hour + 8).ToString("00") + ":" +
15.                minutes.ToString("00") + "E G";
16.
17.    text.text = text_shadow.text = label;
18. }
```

La función calcula los segundos *ingame* que dura cada una de las 16 horas antes del anochecer. Y con ese valor y sabiendo los segundos de juego que llevamos, conseguimos el número de horas y minutos deseados. Luego simplemente hay que formatearlos a AM o PM según corresponda.

Hay dos campos de texto para que en el juego tengamos esa pequeña sombra negra por detrás del reloj. Eso lo conseguimos colocando detrás el mismo texto pero en negro y ligeramente desplazado.

Reto 8

Dentro del «dayNightManager» deberíamos controlar que cuando los segundos lleguen a 0, se pare el juego y se muestre el mensaje de «Game Over».

```
1. if (secondsLeft < 0)
2. {
3.     // gameover es el campo Text que tiene escrito
4.     // el mensaje que se mostrará en el centro de la pantalla
5.     gameover.SetActive(true);
6.     Time.timeScale = 0;
7. }
```

Resumen

En este módulo didáctico hemos acabado de profundizar en las partes básicas que nos quedaban pendientes del motor Unity.

Hemos visto cómo funcionan las animaciones en un juego 3D y cómo Unity las gestiona, gracias al Animator y a sus grafos de estados. Hemos visto también la ventaja que tiene utilizar avatares de tipo «Humanoid» para poder reutilizar animaciones de un personaje a otro.

También hemos aprendido cómo funcionan los sistemas de partículas y sus peculiaridades, poniendo mucho cuidado en comprender cada una de sus múltiples opciones de cara a generar todo tipo de sistemas.

Hemos profundizado en el sistema de sonido, aprendiendo qué es el «Audio Mixer» y cuáles son sus ventajas, cómo se trabaja con sonidos 3D y cómo podemos añadir efectos o filtros a estos para que el resultado tenga mucha más calidad.

Por último, hemos trabajado con la iluminación, uno de los factores clave en el aspecto visual de un juego. Hemos aprendido qué es la *Global Illumination* y qué son las «Light Probes», y hemos visto los diferentes tipos de luces que podemos utilizar en nuestros juegos.

Para terminar, en la parte práctica hemos creado un juego en tercera persona gracias a los *prefabs* «Third Person Controller» y «Free Look Camera Rig», que nos han ahorrado mucho trabajo.

Hemos seguido utilizando *assets* de la Store y hemos fomentado que el alumno curioseee con el contenido de estos paquetes y consulte, investigue o modifique cosas de los *scripts* que vienen con ellos.

En este momento, ya tenemos unas buenas bases sobre todo lo que nos ofrece Unity, y hemos visto lo relativamente rápido que resulta conseguir buenos resultados si sabemos cómo hacerlo.

¡Ahora ya podéis empezar a crear vuestros propios juegos!

