



**Máster en Desarrollo de Videojuegos
(Especialidad de Programación)**

Motores: Unity

Construyendo un FPS. I Parte.

[09 - 10 - 2019]

Ismael Sagredo: isagredo@ucm.es



Universidad Complutense de Madrid



Contenido

Preparativos.....	2
First-Person Shooter (FPS).....	3
Prototipando un FPS: Crear nuevo proyecto	6
MODIFICAMOS EL COMPORTAMIENTO DEL CHARACTER CONTROLLER.....	10
Objetos con Física.....	11
La Escena de Inicio.....	14
¿Qué sería de un fps sin armas?.....	14
Mecánica de disparo	14
Sonido 3D	18
¿Se te cuelan los pollos por la pared?	18
Doble Cámara de renderizado para un FPS.....	19





Preparativos

A lo largo de las siguientes sesiones de prácticas, realizaremos una serie de ejercicios utilizando el motor de juegos **Unity3D**.

Todas las sesiones se realizarán sobre un **mismo proyecto**, de forma que vaya creciendo poco a poco hasta llegar a ser algo parecido a un juego (en la medida de lo posible ^_^).

Debido a esto último, se aconseja que os guardéis en un **pendrive** o algo similar los proyectos que vayáis acabando después de cada clase.

Si por alguna razón **no podéis** hacer una copia de seguridad de vuestro trabajo, y cuando volváis al laboratorio han borrado los proyectos no os preocupéis, en el **La Wiki** se irán colgando todas las **versiones intermedias**, de forma que, para cada práctica estará disponible, ya **resuelta**, la anterior.





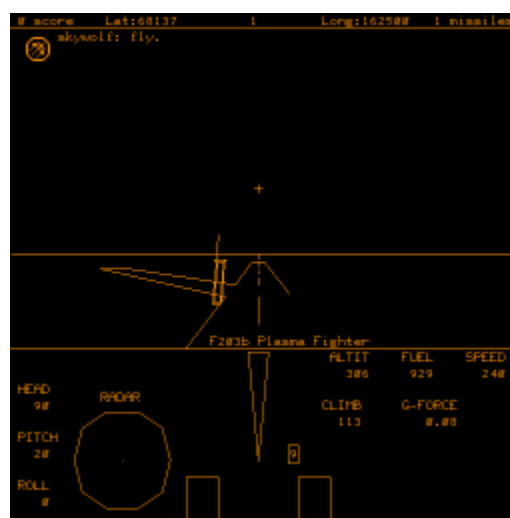
First-Person Shooter (FPS)

Los FPS (quien no los conoce) son un género de videojuegos y subgénero de juegos de acción en los que el jugador observa el mundo desde la perspectiva del personaje protagonista de este y que consisten, básicamente, en acabar con los enemigos disparando con un arma.

El primer FPS surge en 1974, Maze War, diseñado por **Steve Colley** para Mac y otras computadoras de la época.



Junto con Spasim:



Ambos permitían batallas multijugador por red local.





Pero los verdaderos padres del género aparecieron en la época de los 90 con juegos como Wolfenstein 3D (1992), Doom (1993) de la mano de ID Software.

Y la consagración total vino con Quake (1996) y Half life (1998)



A la lista de títulos podemos añadirle algunos de especial relevancia:

- Deus Ex: Añadió rol y sigilo.
- Quake 3 Arena o Counter Strike: Referentes en el multiplayer competitivo.
- Aliens Vs Predator: Distintos personajes con distintas habilidades.
- Metroid Prime: FPS más de exploración y puzzles que de combate.
- GoldenEye: Primer FPS en consola con éxito.
- Halo: Referente de FPS en consola, apuntado asistido y Behaviour Trees para la IA.
- Half Life 1 y 2: Considerado por muchos como el referente del FPS moderno. Físicas y animaciones por huesos en su primera entrega.
- FarCry / Crysis: por ser los primeros que tendían a ser Sandbox.
- System Shock / Bioshock: Mezclas de géneros. Añaden poderes a los FPS.
- Battlefield / Battlefront: Batallas multijugador-masivas en espacios amplios, donde mezclan vehículos terrestres y aéreos.
- Overwatch: Juego multijugador que intenta hacer un híbrido entre un fps y un MOBA.





Sus mecánicas básicas suelen ser:

- Ofrece un número más o menos alto de enemigos a los que tienes que matar.
- El personaje se maneja con un direccional o teclado y un dispositivo para mover la cámara y apuntar, que puede ser un ratón u otro direccional, en casos más exóticos se puede utilizar un sensor de movimiento (DooM en Switch o Metroid Primer 3 Corruption en Wii).
- Es una de las principales puntas de lanza de la VR. Inicialmente los FPS fueron muy disruptores y pretendían emular la sensación de inmersión de la VR, poniendo al jugar desde el punto de vista del protagonista.
- El juego básicamente consiste en apuntar a los enemigos y dispararlos, intentando que la vida del protagonista no caiga a mínimos mediante algún sistema de regeneración de vida. En caso contrario, el jugador pierde.
- Han derivado en otro tipo de juegos no estrictamente de disparos: Portal.
- A la mecánica básica, se le puede añadir componentes de todo tipo, desde sigilo a saltos, puzles, progresión del personaje, etc.
- En las versiones multijugador normalmente se busca o bien ser el último en morir o bien matar al mayor número de jugadores posibles. También existen modalidades donde el objetivo es mantener el control de una zona por más tiempo.



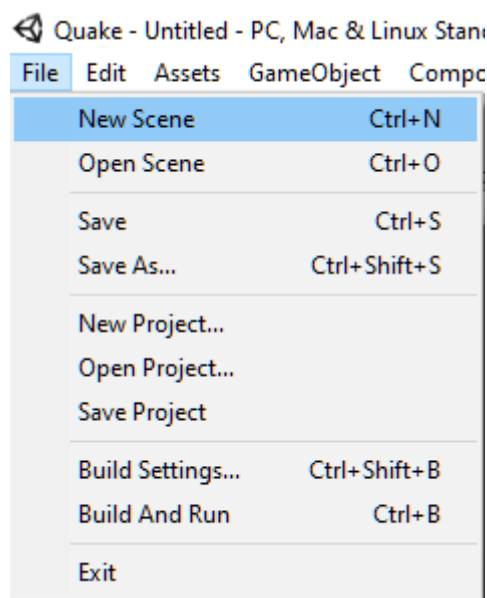


Prototipando un FPS: Crear nuevo proyecto

Vamos a ir construyendo poco a poco un FPS usando Unity. Para ello vamos a crear un proyecto.

Al crear un nuevo proyecto, Unity te facilita la posibilidad de importar una serie de paquetes que pueden simplificar mucho las cosas. Para nuestro proyecto vamos a importar el paquete de controladores que trae consigo una implementación básica de un gran número de comportamiento para el player con un control predeterminado.

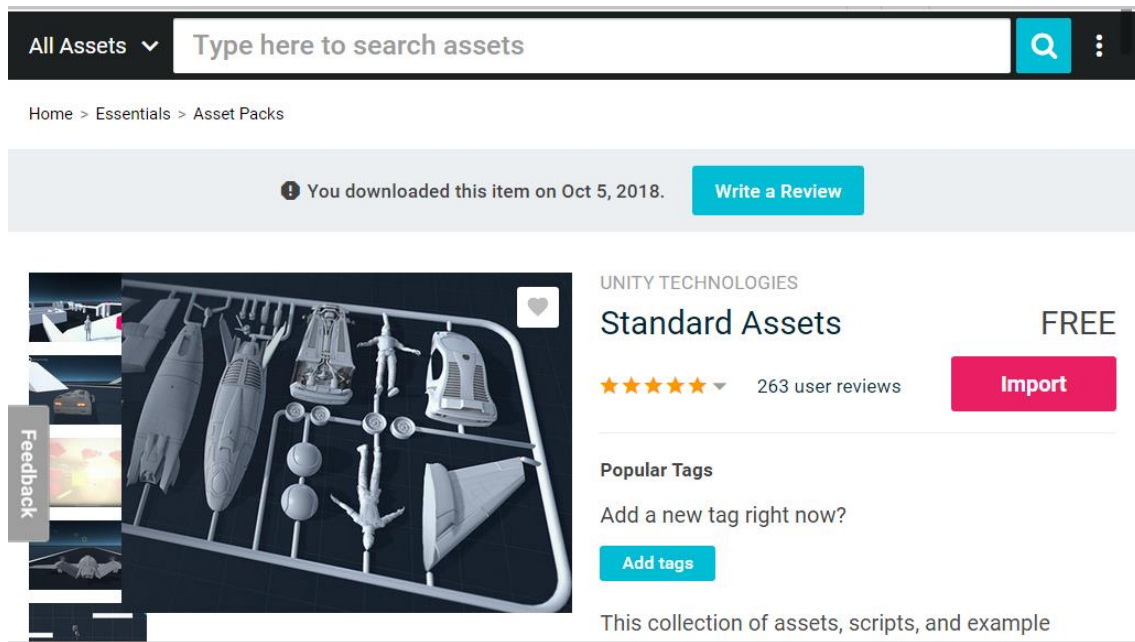
Para crear un proyecto vamos a **File >> New Project**.



Al crear un proyecto nuevo, Unity crea una escena vacía con la **Main Camera** como único GameObject.

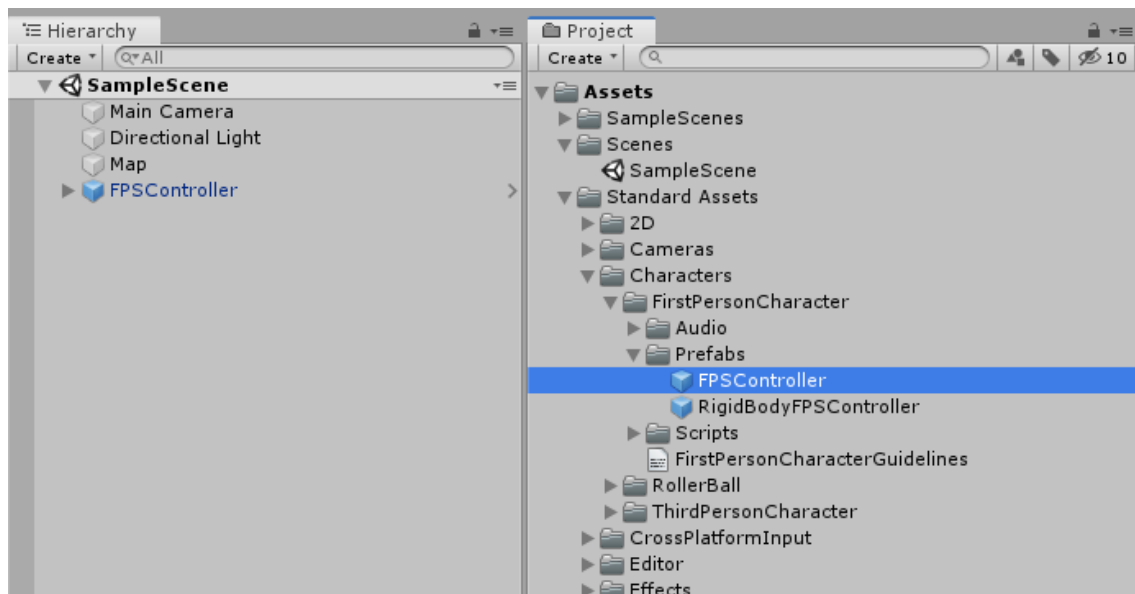
Vamos a utilizar los prefabs de Unity para los Characters. Para ello hay que importarlos de la asset store. Abrimos la asset store en **Windows >> General >> Asset Store** e importamos Standard Assets.





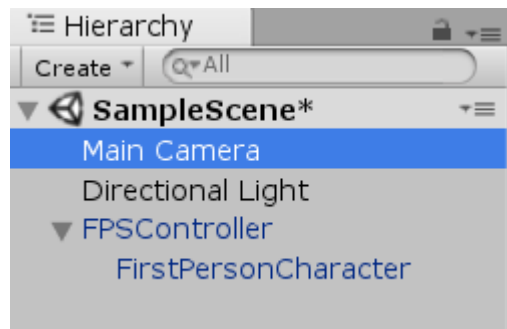
Pulsamos el botón importar. Sólo necesitamos el Asset Character First Person pero vamos a importar todo el paquete para evitar problemas de dependencias.

Vamos a crear el personaje principal. Vamos a arrastrar el prefab **First Person Controller** desde la ventana Project a la Hierarchy.

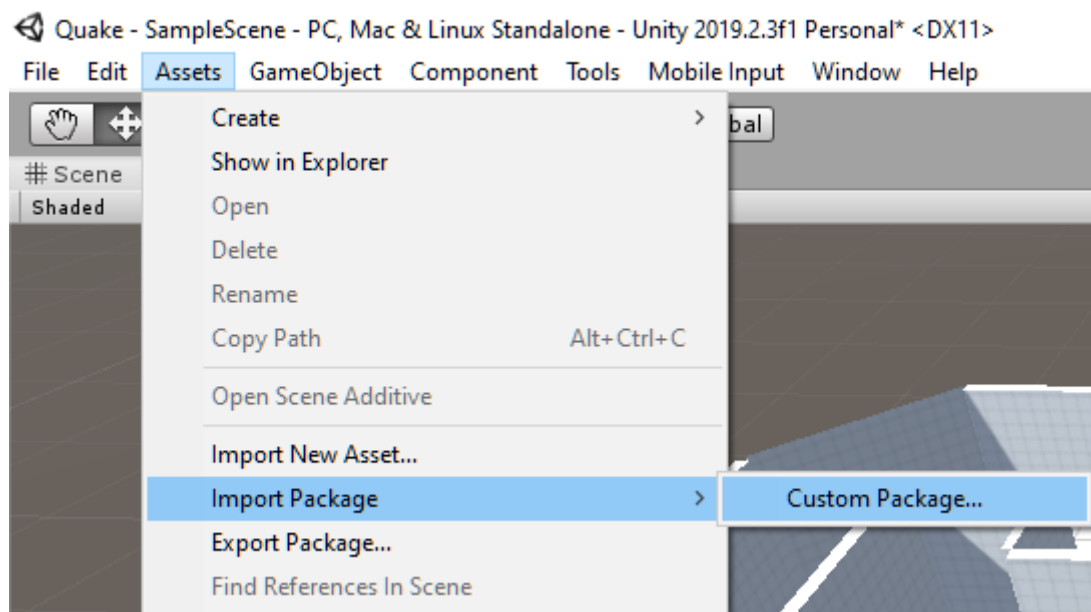


El **First Person Controller** ya dispone de una cámara incorporada que es gestionada por el propio controlador para dar la experiencia subjetiva que se busca. Así pues, debemos quitar la Main Camera que Unity incorpora por defecto.





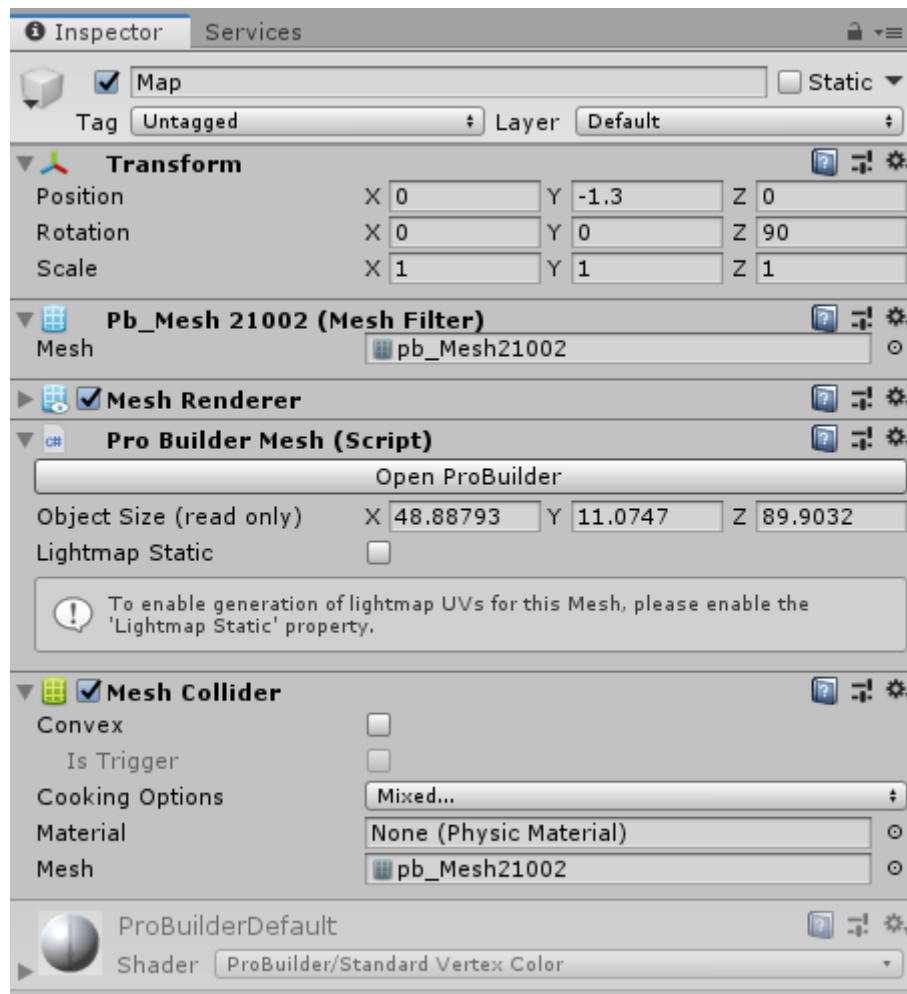
Ahora necesitamos un suelo. Lo que vamos a hacer es importar el paquete del mapa de juego. Para ello vamos a **Assets >> Import Package >> Custom Package** e importamos el paquete **Mapa.unitypackage**






Colocamos el mapa en el 0,0,0, y el player FPSController a cierta altura sobre él.

El mapa está creado usando Pro Builder, una herramienta fantástica para prototipar de la que hablaremos más adelante. Tiene una serie de componentes que describimos a continuación:





El Mesh renderer y Mesh Filter que simplemente sirven para dibujar la malla. De momento no nos interesan. El componente Pro Builder Mesh que nos permite volver a editar el componente con el Pro Builder y finalmente un MeshCollider que nos permite definir una malla de colisión para el mapa que sirve a Unity como colisionador

Probemos el juego pulsando   

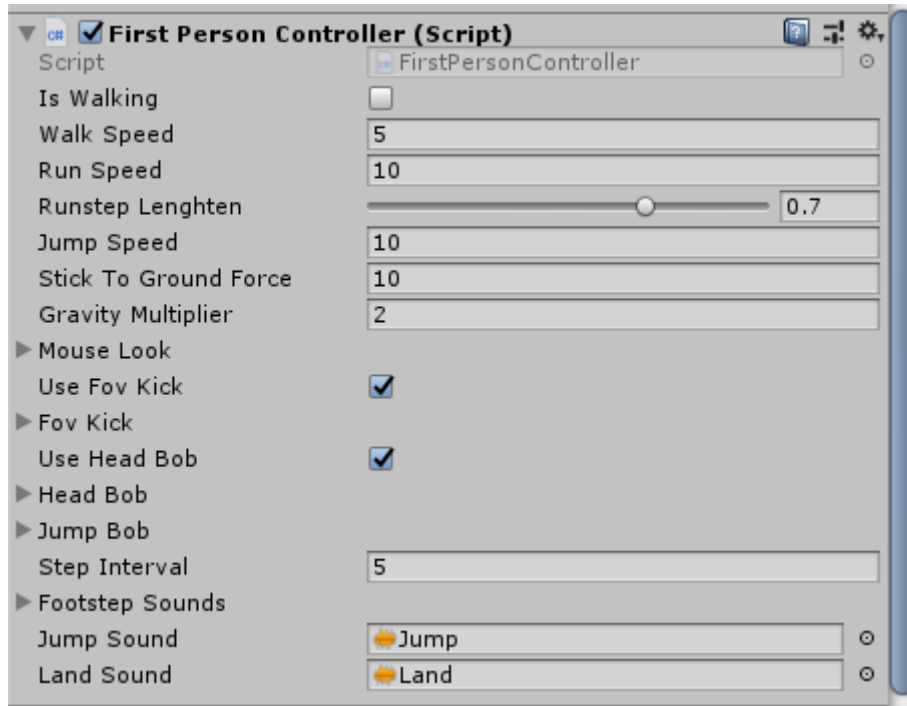
Podemos ver como el personaje cae por acción de la física y se coloca sobre el mapa. Para desplazarlo podemos usar las teclas **W**, **A**, **S**, **D** y para mover la cámara podemos usar el ratón.





MODIFICAMOS EL COMPORTAMIENTO DEL CHARACTER CONTROLLER

El componente **First Person Controller** nos permite configurar el control del player para adaptarlo a nuestras necesidades.



Puedes modificar los parámetros para ver cómo afectan al comportamiento del player.

- **Walk Speed:** Velocidad a la que anda el personaje.
- **Run Speed:** Velocidad a la que corre el personaje.
- **RunStepLengthen:** Velocidad de arranque del primer paso y de la cabeza.
- **Jump Speed:** Velocidad del salto.
- **Stick to ground force:** fuerza de adherencia a la tierra del personaje.
- **Gravity Multiplier:** Multiplicador de la física de la gravedad (simula caer más rápidamente)
- **Fov Kick:** Activa un cambio en el **FOV** (campo de visión) cuando corre el personaje. Esto consigue dar más sensación de movimiento y profundidad.
- **Head Bob:** Movimiento de la cabeza.
- **Jump Bob:** Configuración de cómo se salta.





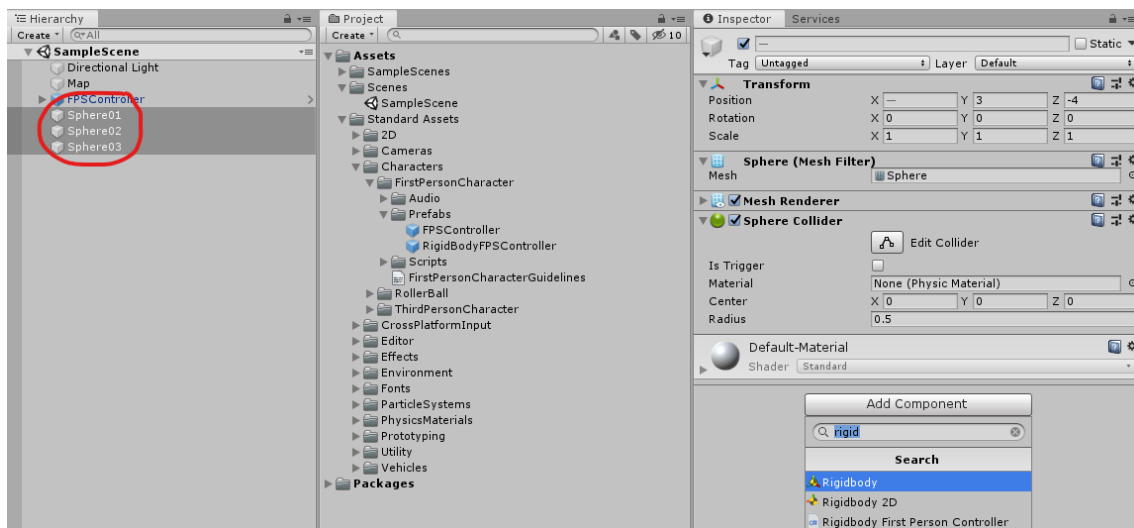
Objetos con Física

Los objetos por defecto no tienen un comportamiento físico preestablecido. Así que hay que añadirle el componente físico **Rigidbody** para que se sientan atraídos por la gravedad, sean colisionables, etc.

Vamos a ver un ejemplo:

Creemos tres esferas en **Game Object >> Create Other >> Sphere** y las nombramos como **Sphere01**, **Sphere02** y **Sphere03**. Si damos a play, veremos que las esferas no son interactivables, simplemente son objetos estáticos que no podemos atravesar y que flotan en el espacio.

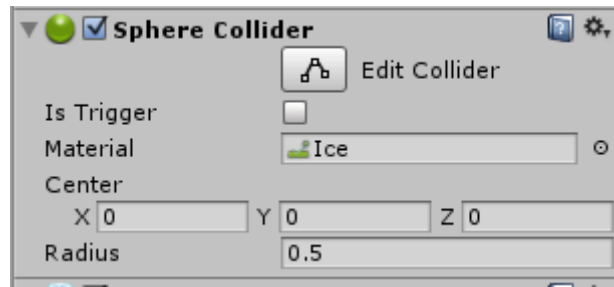
Añadamos el **Rigidbody** en **Components >> Physics >> Rigidbody** a cada una de las esferas. (Si seleccionáis las tres, control + click en cada una de ellas o shift + click para seleccionar un bloque, y agregáis el componente se agrega a las tres. Esto se puede hacer con casi cualquier acción del inspector)



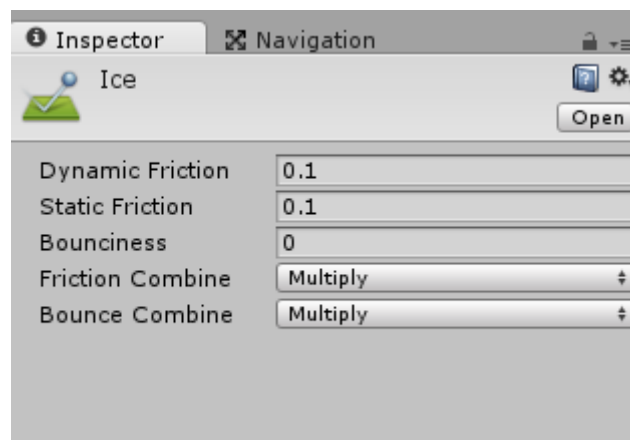
Si probamos la escena, los objetos caerán al suelo por efecto de la gravedad, pero no reaccionarán de una forma muy realista. Se quedarán parados en el sitio. Si queremos cambiarle las propiedades físicas al objeto, hay que darles un “Material Físico”

Unity proporciona una abstracción a las peculiaridades de un **collider** con el concepto de **Material Físico**. Los materiales físicos tienen una configuración almacenada de las características físicas que debe tener un **collider** para que se comporte de una determinada forma. De esta forma, nuestros objetos pueden comportarse como si fueran de goma, cristal, madera, etc.





Los materiales físicos tienen una serie de características o atributos tales como la fricción dinámica, estática o la elasticidad, que podemos configurar y después reutilizar para otros GameObjects. Aquí podemos ver el material físico Ice del paquete de ejemplo.



Podemos crear nuestros propios materiales físicos en **Assets >> Create >> Physics Material**.

Si asignamos estos materiales físicos a cada una de las esferas, podemos apreciar las diferencias entre ambas. Una vez terminado, podemos probar la escena y ver los resultados...

Como se puede ver, ahora los objetos caen y ruedan, y al colisionar entre ellos tienen diferentes comportamientos.

Si nos acercamos a las esferas podemos empujarlas, pero es un efecto algo “tramposo” del comportamiento general de PhysX y su interacción entre los objetos dinámicos y el **Character Controller** o el **Rigidbody Kinemático**. El Character Controller y el rigidbody al que le hemos activado el flag Kinematic son objetos físicos especiales en PhysX que se diferencian de las entidades físicas dinámicas en que la velocidad y masa del cuerpo no puede ser calculada por el motor de física, ya que depende completamente de la lógica del juego. Por lo tanto, si un cuerpo dinámico choca con el Character Controller o un rigidbody, este se desvía y el Character Controller detecta las colisiones con los objetos dinámicos, pero no puede empujarlos. ¿Por qué nuestro personaje puede empujar entonces las esferas? Pues porque en el First Person Controller implementado por Unity, se





ha programado este hecho como parte de la lógica. Si abrimos el script `FirtsPersonController` y buscamos el método **`OnControllerColliderHit`** podemos observar cómo se ha implementado.

```
private void OnControllerColliderHit(ControllerColliderHit hit)
{
    Rigidbody body = hit.collider.attachedRigidbody;
    //dont move the rigidbody if the character is on top of it
    if (m_CollisionFlags == CollisionFlags.Below)
    {
        return;
    }

    if (body == null || body.isKinematic)
    {
        return;
    }
    body.AddForceAtPosition(m_CharacterController.velocity*0.1f,
hit.point, ForceMode.Impulse);
}
```

NOTA: esa constante debería estar o en el inspector o con `const` pero no lo hemos programado nosotros ^_^

`OnControllerColliderHit` es llamado cuando existe un componente **`Character Controller`** y este entra en contacto con otra entidad colisionable. Al colisionar con un objeto colisionable que no sea kinemático, se aplica una fuerza (`AddForceAtPosition`) proporcional a la velocidad a la que se esté moviendo el `Character Controller`.

Ahora vamos a guardar la escena para no perder los cambios. Vamos a **`File >> save`** le damos un nombre y ya la tenemos salvada.





La Escena de Inicio

Ahora que tenemos el player más o menos configurado, lo vamos a guardar como un **prefab**. Para ello arrastramos el **GameObject** a la ventana **Project** y lo nombramos como “FPSPlayer”.

Vamos a importar un paquete con algunos recursos para facilitarnos la tarea.

En la ventana Project le damos **botón derecho >> Import package >> Custom package** y seleccionamos el paquete **fps_resources.unitypackage**.

La importación nos crea una serie de directorios y recursos en nuestro proyecto.

Colocamos nuestro player arrastrando el prefab “fps_player” a la jerarquía de una nueva escena o reutilizamos la anterior.

¿Qué sería de un fps sin armas?

Instanciamos el **Launcher** arrastrando el prefab a la escena que se llama **Launcher**. (Cualquiera de ellos)

Colocar el **Launcher**, hay que mirar en qué dirección está mirando la cámara y rotar el Launcher de forma que cuadre con la vista. Para ello aconsejamos colocar primero el objeto en el (0, 0, 0), rotarlo y luego desplazarlo para que parezca en el centro.

Como se puede apreciar, si en este punto ejecutamos el juego, el Launcher siempre mira hacia el frente, independientemente de donde esté mirando la cámara. Para resolver esto, tenemos que **poner el Launcher como hijo de la cámara**, para conseguir que la rotación y la translación vayan con la cámara.

Mecánica de disparo

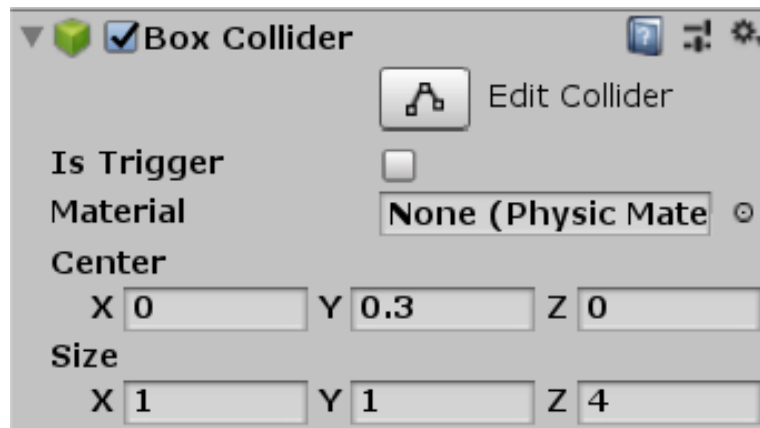
El disparo del lancher del juego se basa en una premisa muy sencilla: instanciar un proyectil con una velocidad inicial muy elevada. Dichos proyectiles serán GameObjects con física. Para conseguirlo:

1. Instanciamos el prefab ChickenProjectile.

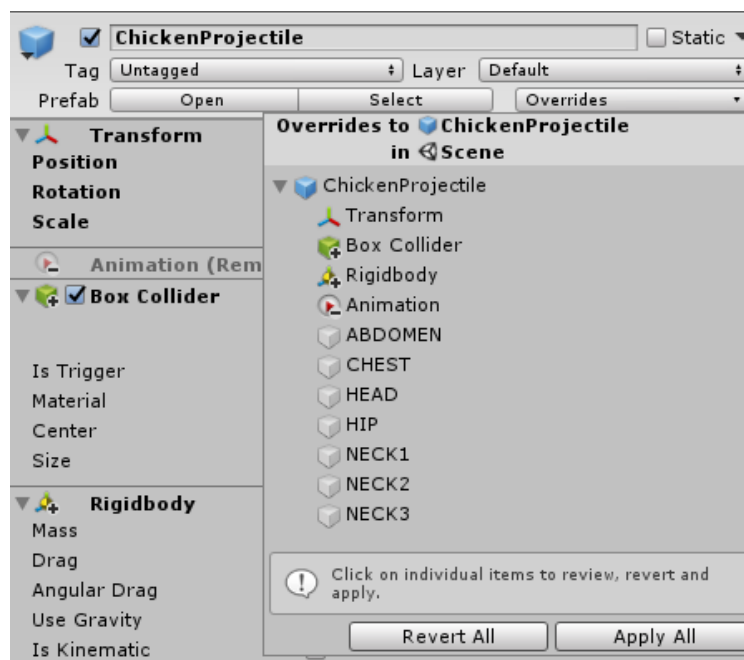




2. Eliminamos el componente Animation que crea por defecto.
3. Añadir un BoxCollider y configurarlo como se muestra en la imagen.



4. Configurar el Physic Material Bouncy para el box collider.
5. Añadirle un componente Rigidbody.
6. Actualizar su prefab arrastrando nuevamente el objeto configurado al mismo para que así podemos reutilizarlo o bien en la pestaña **override**.



Ahora vamos a utilizar este proyectil ya configurado para implementar la mecánica de disparo. El pseudocódigo de la mecánica de disparo es el siguiente:

1. Se espera a que el jugador pulse el botón de disparo.
2. Se instancia un proyectil en una posición determinada.
3. Se le otorga al proyectil una gran velocidad inicial en la dirección hacia la que está mirando el jugador.
4. Se controla un tiempo determinado, mediante el cual, aunque el jugador esté intentando disparar, no sucede nada.
5. Volver a 1.





Para implementarlo:

- Creamos un GameObject vacío (ShootPoint) y lo colocamos en el punto donde se disparan los pollos. Debería ser hijo del Lancher. Digamos que este GameObject object es el punto de respawn del proyectil.
- Arrastramos el script shoot.cs al project

Buscamos la función **GetFireButton()** e implementamos el TODO 1.

```
if (m_IsAutomatic)
    return Input.GetButton("Fire1");
else
    return Input.GetButtonDown("Fire1");
```

Buscamos la función ShootProjectile () y completamos el TODO 2

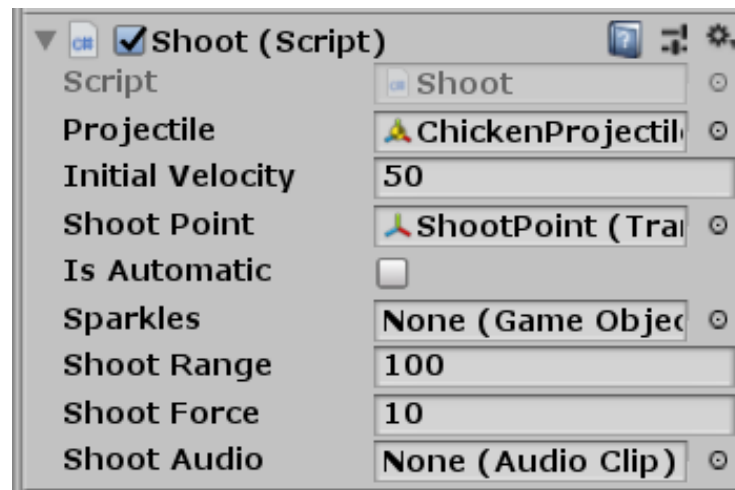
```
// 1.- Instanciar el proyectil pasado como variable pública de la clase,
// en la posición y rotación del punto de disparo "m_projectile"
// 1.2.- Guardarse el objeto devuelto en una variable de tipo Rigidbody
// 2.- Asignar una velocidad inicial en función de m_Velocity al campo
// velocity del rigidBody. La dirección será la del m_ShootPoint. Una vez que esté
// orientado el pollo simplemente hay que añadirle velocidad.
// 3.- Ignorar las colisiones entre nuestro proyectil y nosotros mismos
Rigidbody project = Instantiate(m_projectile,
m_ShootPoint.transform.position, m_ShootPoint.rotation) as Rigidbody;
project.velocity = project.transform.forward * m_InitialVelocity;
Collider projectileCollider = project.GetComponent<Collider>();
Collider mycollider = transform.root.GetComponent<Collider>();
Physics.IgnoreCollision(projectileCollider, mycollider);
```

Añadimos la función OnDrawGizmos() donde mostraremos un puntero laser que nos oriente a la dirección de disparo.

```
private void OnDrawGizmos()
{
    if (m_ShootPoint != null)
        Debug.DrawRay(m_ShootPoint.position, m_ShootPoint.forward * 2f,
Color.red);
}
```

- Una vez hecho esto, debemos añadir Shoot.cs como componente del GameObject Bazooka y probar su funcionamiento.
- Configurar los atributos de Projectile, Initial Velocity y Shoot Point de Shoot para que funcionen correctamente.





Al disparar podemos ver como la escena se va llenando de objetos. Sería interesante destruir los pollos para que no se nos acumulen en la pantalla.

Por ahora destruiremos los objetos, aunque hay mejores formas de gestionar esto, como por ejemplo un pool de objetos. Por ahora nos vamos a centrar en destruir los proyectiles pasado cierto tiempo.

Arrastramos el script de AutoDestroy a nuestro proyecto y de ahí al prefab de ChickenProjectile.

Implementar el TO-DO 1 del script para que cada cierto tiempo, los objetos desaparezcan.

```
void Start ()
{
    // ## TO-DO 1 - Llamada a la función Destroy, pasandole un puntero a mi
    mismo (gameObject) y un delay ##
    Destroy(gameObject, m_Delay);
}
```





Sonido 3D

Vamos a añadir sonidos para conseguir algo de feedback para el jugador. El objetivo es añadir sonido a nuestros proyectiles pollo. Para ello:

1. Añadimos la carpeta sounds al proyecto.
2. Completamos el TO-DO 1 de CollisionSound.cs
3. Añadimos el script al ChickenProjectile
4. Añadimos el sonido basketball_bounce en el Collision Sound del componente.
5. Añadiremos al prefab del proyectil el componente AudioSource.

```
// ## TO-DO 1.1 - Añadir la función de la API que se lanza cada vez que el
GameObject colisiona. Pista: OnCollision...
void OnCollisionEnter()
{
    // ## TO- DO 1.2 - En caso de que haya sonido, reproducirlo una única
    vez. Pista: audio.PlayOne...
    if (m_CollisionSound != null)
    {
        AudioSource audio = GetComponent<AudioSource>();
        audio.PlayOneShot(m_CollisionSound);
    }
}
```

¿Se te cuelan los pollos por la pared?

Vamos a jugar con la velocidad del fixed update para evitarlo. Si no se te cuelan y quieres forzarlo, aumenta el fixed update a 0.05 para forzarlo. Para solucionar que los objetos atravesasen la pared podéis poner los objetos más lentos o bien subir la velocidad de cálculo de la física.

Ojo, si la velocidad de cálculo de la física se incrementa, el coste de calcular toda la física es mayor, por lo que hay que tener cuidado.

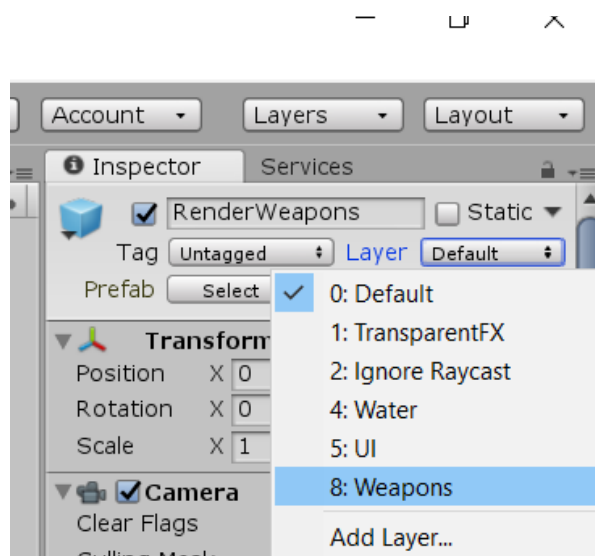




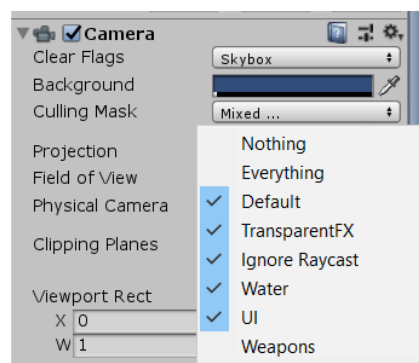
Doble Cámara de renderizado para un FPS

Si nos acercamos a una pared mucho, podemos ver que el arma se introduce en la pared. Esto no es lo que solían hacer los fps clásicos. El arma en todo momento era visible. Actualmente hay otras soluciones para evitar que el arma atraviese una pared. Por ejemplo, en el nuevo Doom el personaje cuando se acerca a una pared retira el arma, pero en la mayoría de los juegos lo que se suele hacer es pintar el arma siempre por encima del resto de la escena.

Para hacerlo en Unity vamos a crear una nueva layer que denominaremos Weapons



Y la asignamos al arma disponible. Ahora vamos a configurar la cámara del FPS y en **Culling Mask** desactivamos el tick de Weapons.



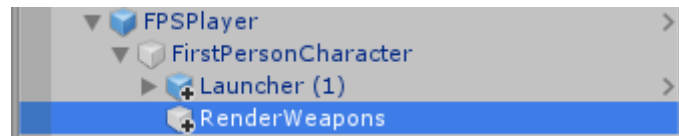
De esta forma la cámara principal no renderizará el arma.

Ahora creamos una nueva cámara y la llamamos **RenderWeapons** y la hacemos hija de FirstPersonCharacter y el gameobject Weapon hijo de esta nueva cámara. Aseguraos que esté en el 0,0,0.



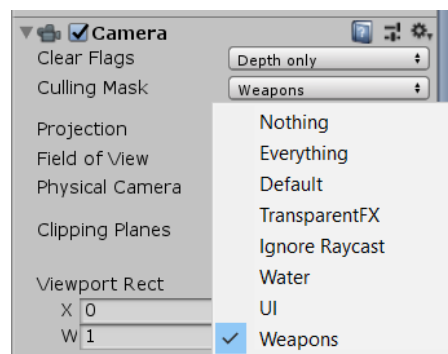


Aseguraos también que no tiene el tag **MainCamera**, ya que es el que utiliza el FPSController para cargar la cámara. Si lo tiene, cambiarlo. La escena debe quedar similar a esto:



Ahora configuramos la nueva cámara y lo configuramos de la siguiente forma:

- El **Culling Mask** sólo tiene el tick Weapons.
- En **clear flags** marcamos Depth only.
- En la variable **Depth** ponemos un 1.



Si probamos veremos que el arma no se introduce en la pared. ¿Cómo funciona?

Cuando se hace el render, la gpu antes de pintar una geometría nueva comprueba el **Depth Buffer** que es un buffer donde se guarda la profundidad de los pixeles dibujados. Si el nuevo píxel a dibujar está más cerca que el píxel que ya se dibujó en la misma coordenada previamente, se dibuja. Si el píxel a dibujar está más lejos no se dibuja. Si el atributo Clear Flags está a *depth only* lo que hace es borrar del depth buffer todo lo que hubiera. Como la cámara MainCamera ya ha terminado de dibujar (se dibujan en orden de profundidad, la MainCamara está en profundidad 0 y la nueva está en profundidad 1 por lo que primero pinta la MainCamera y luego la cámara de render de Weapons) y ha escrito el depth buffer, al borrarlo esta nueva cámara que sólo pinta el arma, la dibuja a pesar de que ésta está en realidad dentro de la geometría. Pero el rasterizador no lo sabe ya que cuando intenta dibujarla, supera el **depth** test y por tanto cree que puede dibujarla, aunque en realidad este detrás de geometría que ya dibujó la primera cámara.

