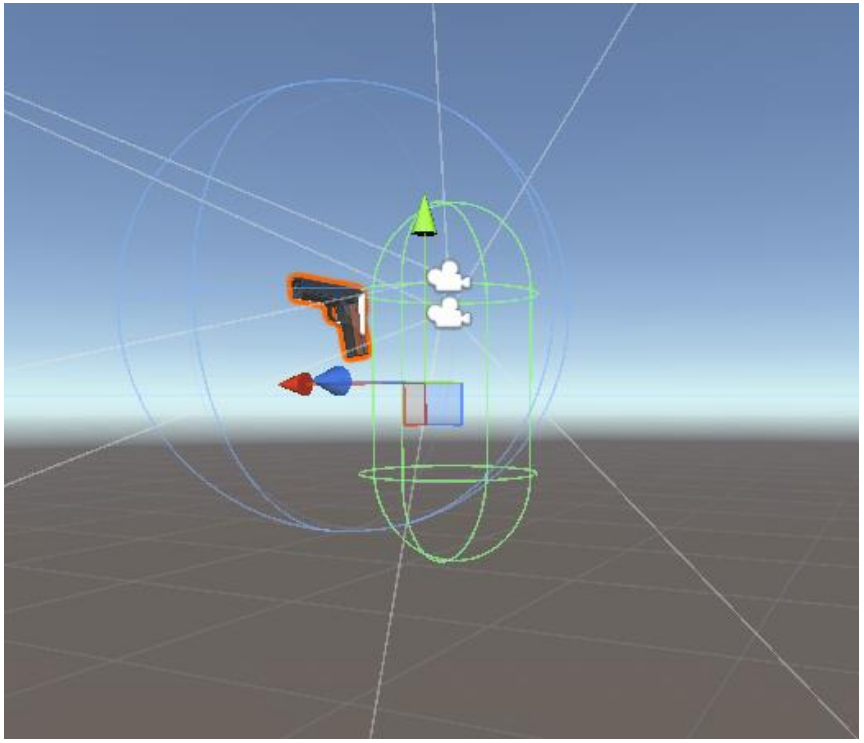
TEMA 5: El jugador.

Javier Alegre Landáburu

Resumen del tema

- ¿Cómo se montan los gameobjects que forman al jugador?
- ¿Qué atributos requiere un arma?
- ¿Cómo se desarrollan las armas a nivel de código?

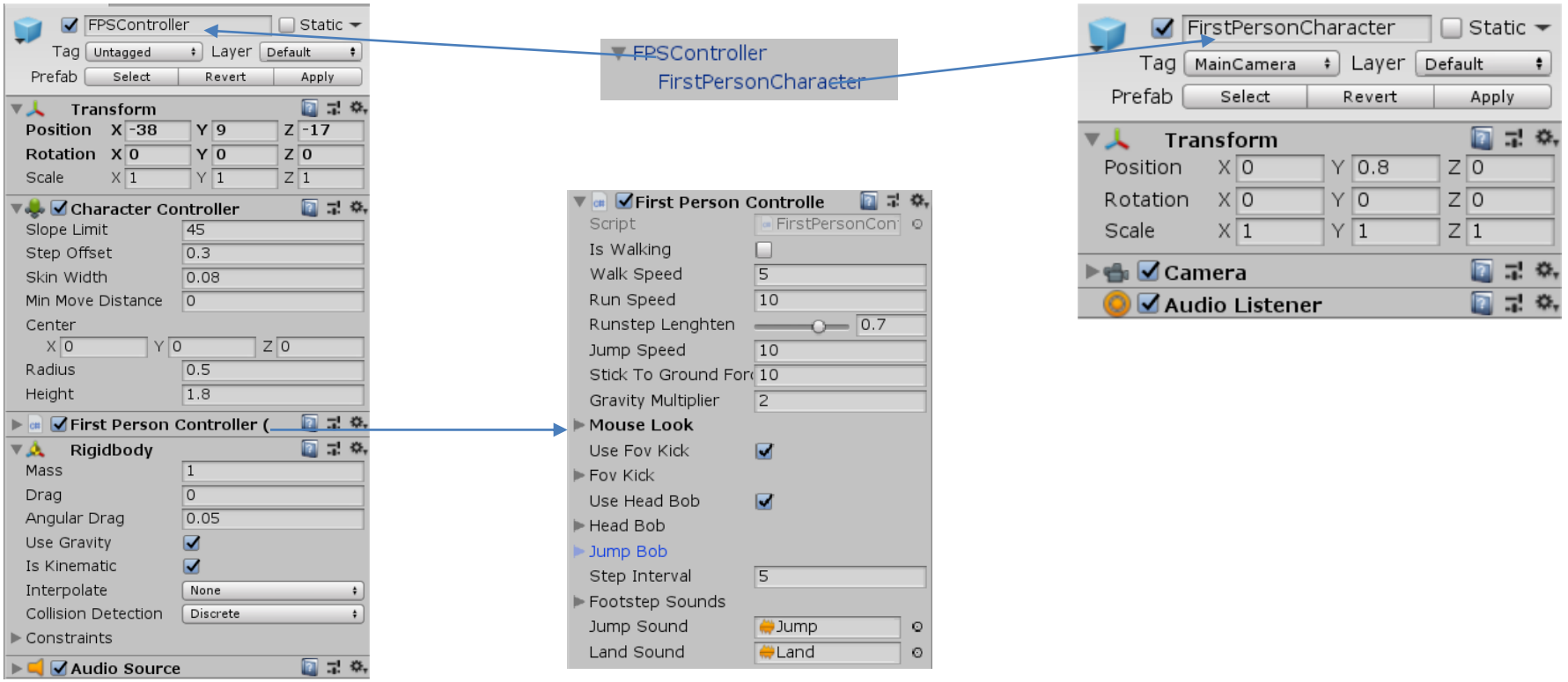
Crear al jugador



- Hay muchas maneras de crear al jugador de un fps con sus movimientos y acciones.
- Si el personaje se va a mover de forma no realista, no nos conviene usar el sistemas de físicas de unity y usaremos un *charactercontroller* para crear al personaje.
- Si queremos que se mueva de forma más realista, podemos usar el componente *rigidbody* y el sistema de físicas de Unity.
- En los StandarAssets de unity tenemos un ejemplo de cada uno.

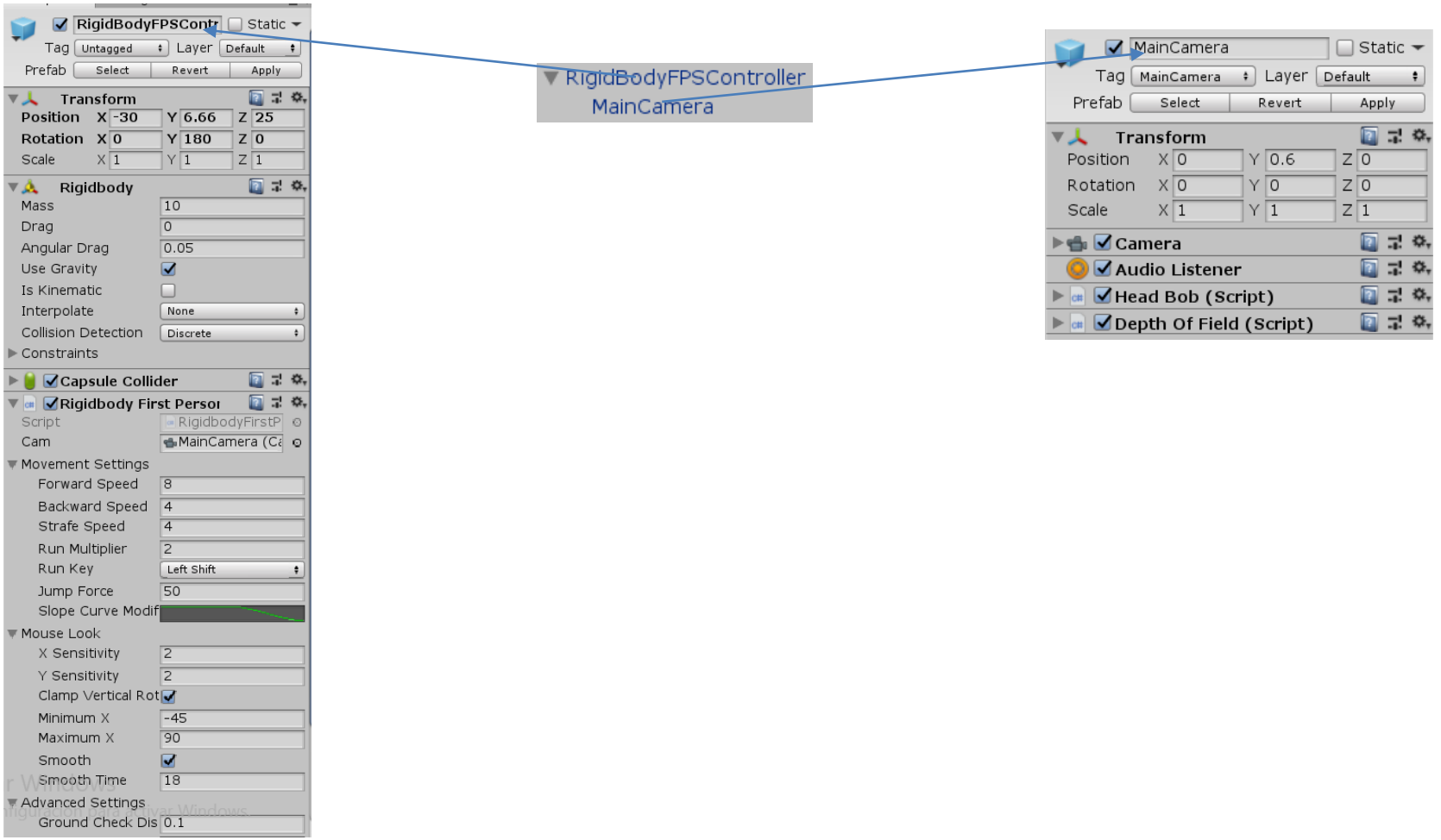
Crear al jugador

- Jugador con *charactercontroller* (Standar Assets):



Crear al jugador

- Jugador con *rigidbody* (Standar Assets):



Crear al jugador

- Vamos a usar el jugador con componente *charactercontroller* para crear nuestro propio jugador.

- Lo primero es añadir una segunda cámara como hija de la cámara principal. Tiene que tener la misma posición y rotación que la original.

- La llamamos *CameraWeapons*.

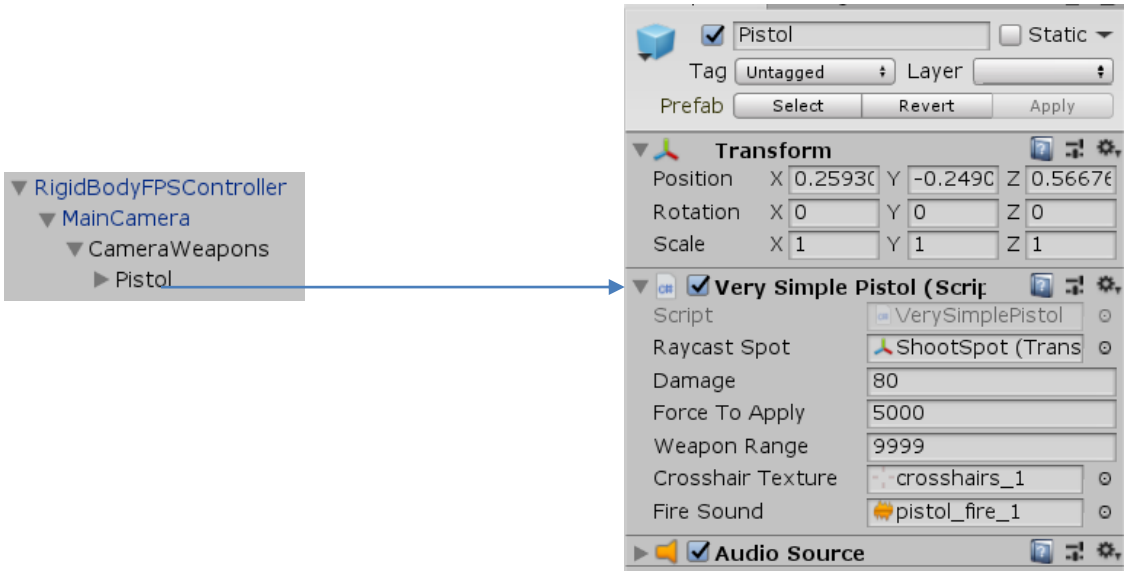
- La cámara principal va a pintar todo el escenario, los enemigos, etc. Tiene tag MainCamera.

- La nueva cámara se va a encargar de pintar el arma. No le ponemos tag.

Crear al jugador

Crear al jugador

- Descargamos algún arma gratuita del asset store y los *standarassets* de unity.
- Por ejemplo: Artsate - PBR Weapon Pack
- Añadimos un *gameobject* hijo llamado *Pistol* a *CameraWeapon*.
- Le añadimos los componentes *audiosource* y *VerySimplePistol* (descargado del blackboard).

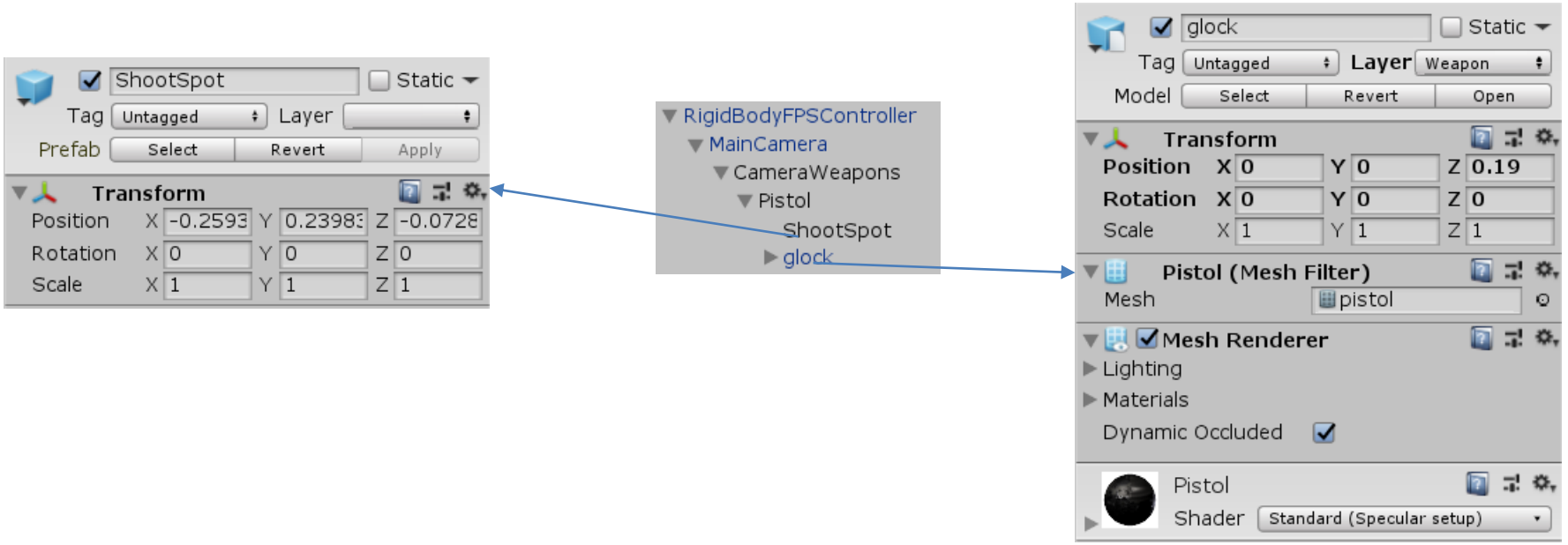


Crear al jugador

- Por último, le añadimos dos hijos a *Pistol*.

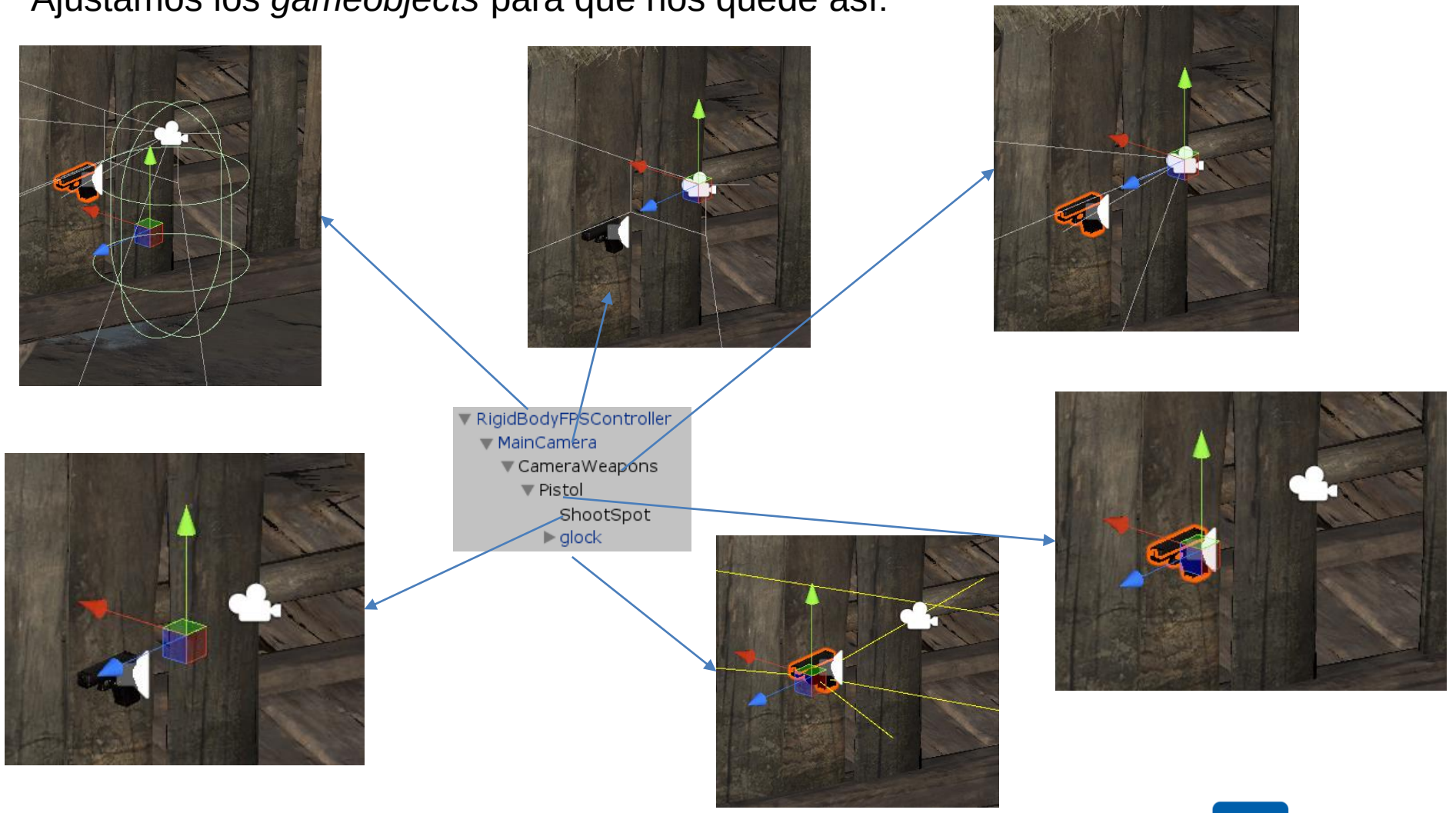
- El primero es un *gameobject* vacío que llamamos *ShootSpot*. Lo usaremos para calcular la dirección y posición del disparo.

- El segundo es el prefab/modelo del arma. Al arma le ponemos *Layer Weapon*.



Crear al jugador

- Ajustamos los *gameobjects* para que nos quede así:

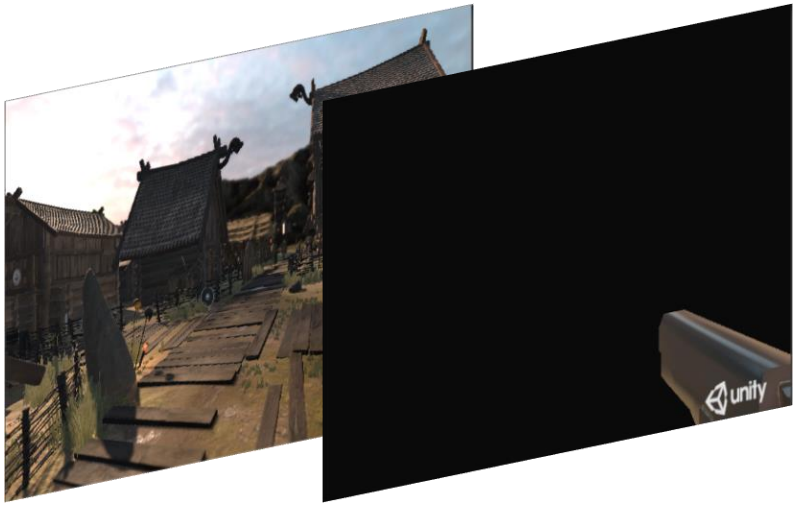


Crear al jugador

- Damos propiedades a las cámaras.

- La *MainCamera*, tendrá mayor profundidad (Depth == 0), pintará todas las capas menos *weapon* (Culling Mask) y de fondo pintará el skybox (Clear Flags).

- La cámara del arma la situamos más cerca del espectador (Depth == 1), pintará sólo la capa *weapon* (Culling mask) y de fondo no queremos que pinte nada para que no tape a la *MainCamera* (Clear Flags == Depth only).



Crear al jugador

- Probamos la escena para comprobar que todo está correctamente montado.
- Si añadimos cubos con un *rigidbody*, podemos dispararles y ver cómo vuelan.

Crear el arma (tipo pistola)

- Vamos a estudiar a fondo el script del arma.
- Lo primero es declarar los atributos necesarios para simular el comportamiento del arma.

```
public  Transform m_raycastSpot;
public  float     m_damage         = 80.0f;
public  float     m_forceToApply   = 20.0f;
public  float     m_weaponRange    = 9999.0f;
public  Texture2D m_crosshairTexture;
public  AudioClip m_fireSound;
private bool      m_canShot;
```

Crear el arma (tipo pistola)

- En el método *Update* se incluye la lógica para que si el jugador presiona el botón de disparar, el arma simule un disparo.

- Cómo es una pistola, hasta que el jugador no suelta el botón de disparo y lo vuelve a presionar, el arma no dispara.

```
private void Update()
{
    if (m_canShot)
    {
        if (Input.GetButton("Fire1"))
        {
            Shot();
        }
    }
    else if (Input.GetButtonUp("Fire1"))
    {
        m_canShot = true;
    }
}
```

Crear el arma (tipo pistola)

- Las balas son muy rápidas y no se ven.
- Por eso, el comportamiento de disparo lo simulamos usando un rayo y viendo con que colisiona.
- En este caso, si colisiona con algo, le añadimos una fuerza al objeto con el que ha colisionado si tiene *rigidbody*. En este punto es donde añadiríamos el generar daño a los enemigos.

```
private void Shot()
{
    m_canShot = false;

    Ray ray = new Ray(m_raycastSpot.position, m_raycastSpot.forward);

    RaycastHit hit;

    if (Physics.Raycast(ray, out hit, m_weaponRange))
    {
        Debug.Log("Hit " + hit.transform.name);
        if (hit.rigidbody)
        {
            hit.rigidbody.AddForce(ray.direction * m_forceToApply);
            Debug.Log("Hit");
        }
    }

    GetComponent<AudioSource>().PlayOneShot(m_fireSound);
}
```

Crear el arma (tipo pistola)

- El arma que estamos usando es muy sencilla, vamos a añadirle más propiedades:
  - La primera propiedad es la munición.
  - Creamos varios atributos en la clase para manejar este comportamiento.
  - El primero es la munición total del arma.
  - El segundo es la munición actual.
  - También, añadimos la acción de recargar el arma en el método Update.
  - Opcionalmente, podemos añadir un sonido de recarga.

Crear el arma (tipo pistola)

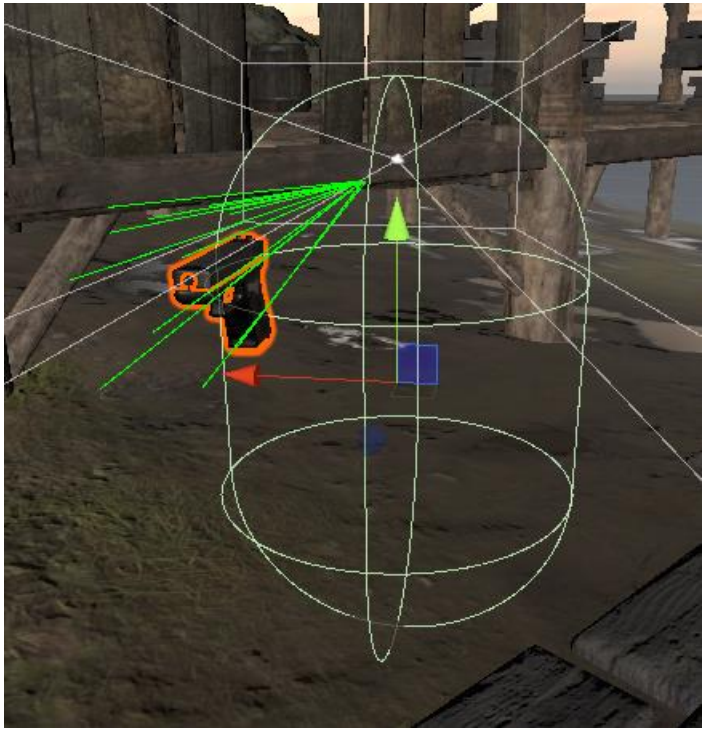
- También, vamos a añadirle un ratio de disparo máximo por segundo:
    - Necesitamos un atributo que nos indique el máximo número de balas por segundo (tendremos que hacer el cálculo de cuanto tiempo tiene que pasar entre disparos -> 1 / ratio de disparo).
    - Necesitamos un contador de tiempo para controlar el tiempo entre disparos.
    - El contador de tiempo se incrementa en cada *Update* en una cantidad *Time.deltaTime*
    - Si ha pasado el tiempo suficiente, al pulsar el botón de disparo el arma dispara.
    - Al disparar ponemos el contador de tiempo a cero.

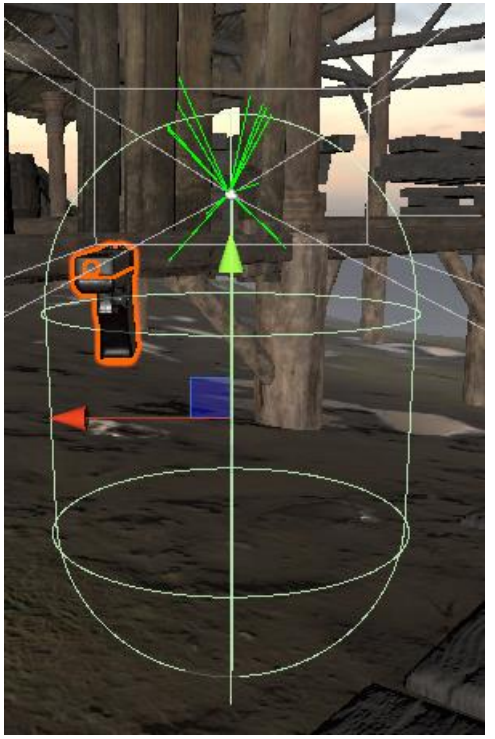
Crear el arma (tipo pistola)

- Cómo última característica añadimos la propiedad de puntería al arma:
    - Por cada disparo que realicemos está puntería se ira perdiendo haciendo el arma más imprecisa.
    - Recuperamos puntería con el tiempo.
    - Así, si el jugador dispara muchos tiros seguidos el arma ira fallando cada vez más.
    - Cómo no tenemos forma de comprobar en que trayectoria salen las balas, vamos a pintar su trayectoria usando el método *DrawRay*.
        - Debug.DrawRay(m_raycastSpot.position, direction, Color.green, 4);

Crear el arma (tipo pistola)

- Puntería:





Crear el arma (tipo pistola)

- Para poder implementar este comportamiento, tenemos que variar el vector de dirección que se usa para calcular el rayo del Raycast.

- Antes hacíamos:

Ray ray = new Ray(m_raycastSpot.position, m_raycastSpot.forward);

- Ahora haremos:

Ray ray = new Ray(m_raycastSpot.position, directionForward);

Siendo directionFordward el vector m_raycastSpot.forward con pequeñas variaciones.

```
float accuracyModifier = (100 - m_currentAccuracy) / 1000; // Entre 0 y 0.1f
Vector3 directionForward = m_raycastSpot.forward;
directionForward.x += UnityEngine.Random.Range(-accuracyModifier, accuracyModifier);
directionForward.y += UnityEngine.Random.Range(-accuracyModifier, accuracyModifier);
directionForward.z += UnityEngine.Random.Range(-accuracyModifier, accuracyModifier);
m_currentAccuracy -= m_accuracyDropPerShot;
m_currentAccuracy = Mathf.Clamp(m_currentAccuracy, 0, 100);

Ray ray = new Ray(m_raycastSpot.position, directionForward);
```

Crear el arma (tipo pistola)

- Se añade en el método Update la propiedad de recuperar puntería por segundo.

```
m_currentAccuracy = Mathf.Lerp(m_currentAccuracy, m_accuracy, m_accuracyRecoverPerSecond * Time.deltaTime);
```

Crear el arma (tipo pistola)

- Para que sea más visual, vamos a añadir el retroceso al arma. Así, cada vez que realicemos un disparo, el arma se moverá dando mejores sensaciones al usuario.

- Necesitamos mover el modelo en coordenadas locales una cierta cantidad si se realiza un disparo. Añadimos al método “Shot”:

m_weapon.transform.Translate(new Vector3(0, 0, -m_recoilBack), Space.Self);

-También necesitamos que se recupere la posición del objeto. En el Update añadimos:

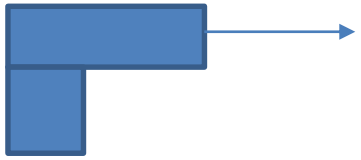
*m_weapon.transform.position = Vector3.Lerp(m_weapon.transform.position, transform.position, m_recoilRecovery * Time.deltaTime);*

Crear el arma (tipo pistola)

- Posibles mejoras:
    - Añadir una máquina de estados de animaciones a la pistola. Podemos crear el estado disparando, recargando, corriendo con la pistola, etc y asignar una animación a cada uno.
    - Añadir un sistema de partículas para simular el “fogonazo” que sale de la punta de la pistola.
    - Añadir un gameobject que sea un casquillo de bala y que caiga desde el cañon del arma al disparar.
    - Añadir los impactos que generan las balas en los objetos del escenario.
    - Añadir sistemas de partículas al impactar las balas dependiendo del material.
    - Añadir sonido al impactar las balas dependiendo del material.

Crear el arma (tipo escopeta)

- A partir del arma que tenemos, es muy fácil desarrollar un script para simular el comportamiento de una escopeta.
- La escopeta funciona igual que la pistola pero dispara varias balas a la vez.
- Cada bala tiene una pequeña desviación en su dirección. Usamos la precisión para generar esto (la precisión ya le mete una pequeña desviación aleatoria).
- Añadimos una propiedad que sea el número de balas que se disparan a la vez. Si estamos configurando una pistola será uno y en una escopeta, dos o más.



Crear el arma (tipo ametralladora)

- La ametralladora también es muy fácil de conseguir a partir del script de la pistola.

- Tenemos una variable llamada canShot, que impide que el usuario sea capaz de disparar una segunda vez si no ha vuelto a pulsar el botón. En el caso de que sea una ametralladora, ignoramos esta variable y disparamos si el usuario mantiene pulsado el botón.

```
m_canShot = (m_isAMachineGun) ? true : m_canShot;

if (m_shotTimer >= m_timeBetweenShots && m_canShot)
```

- La propiedad de puntería se aprovecha mucho en este tipo de armas, porque cuanto más disparo de seguido una ametralladora en un videojuego peor puntería tiene el arma.

Crear el arma (tipo lanza cohetes)

- El lanzacohetes se diferencia de la pistola en que a la hora de realizar el disparo, instancia un cohete en vez de lanzar un rayo.

- Podemos añadir en el código una variable boleana para distinguir entre los dos tipos de disparo.

```
if (Input.GetButton("Fire1"))
{
    if (m_isRocketLauncher)
    {
        ShotRocket ();
    }
    else
    {
        Shot();
    }
}
```

Crear el arma (tipo lanza cohetes)

- El método ShotRocket es prácticamente igual al método Shot, salvo en el momento de disparar.

```
public void ShotRocket ()
{
    m_shotTimer = 0.0f;

    if (m_currentAmmo <= 0)
    {
        return;
    }

    m_currentAmmo--;

    m_canShot = false;

    for (int i = 0; i < m_simultaneousShots; i++)
    {
        GameObject proj = Instantiate(m_rocket, m_rocketSpot.position, m_rocketSpot.rotation) as GameObject;
    }

    m_weapon.transform.Translate(new Vector3(0, 0, -m_recoilBack), Space.Self);

    GetComponent<AudioSource>().PlayOneShot(m_fireSound);
}
```

Crear el arma (tipo lanza cohetes)

- En este tipo de arma, es el cohete el que tiene la responsabilidad de moverse y de hacer daño al impactar. Necesita un script que sea capaz de realizar estas tareas.

- *GetComponent<Rigidbody>().velocity = transform.forward * speed;*

- *Void OnCollsionEnter ()*
  {
    - *Destruir cohete*
    - *Hacer daño.*
    - *Generar sistema de partículas de explosión*

  }

Crear el arma (tipo lanza granadas)

- El lanza granadas funciona igual que el lanzacohetes.
- Es el script del proyectil el que va ser diferente.
- En el lanzacohetes el proyectil sigue una trayectoria recta, en el lanzagranadas la trayectoria es curva.
- La granadas a diferencia de los cohetes, no explotan al impactar contra algo, explotan pasado un tiempo.

Ejercicio

- Crea un script que nos permita cambiar de arma tanto con los botones numéricos, como con la rueda del ratón.

- Añade un par de pistolas, la escopeta, la ametralladora, el lanzador de misiles y el lanzagranadas.

- Añade enemigos con IA que se muevan por el escenario.

- Añade la propiedad de vida tanto al player como a los enemigos. Si son atacados, resta daño a la vida hasta que se quede a cero. Si es el player, se acaba la partida, si es el enemigo, te suma puntos.

- Crea una UI donde se vea el número de balas que tiene el arma, la vida del jugador, la puntuación, la mirilla del arma, etc.

- Añade sistemas de partículas para representar los fogonazos y los impactos de bala en el enemigo.