

Inverse Kinematics y animación procedural

Luis Díaz

Definiendo poses

[illegible]

Poses: Forward Kinematics

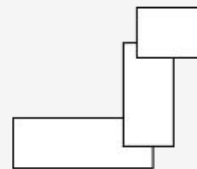
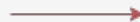
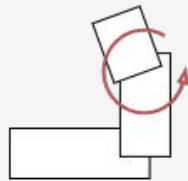
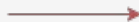
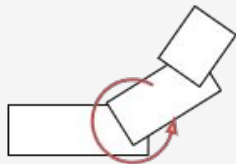
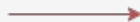
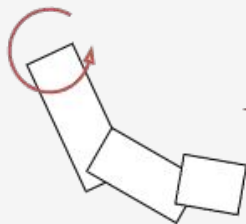
- Supongamos que tenemos un modelo en 3D con Rig
 - Y queremos cambiar su pose



Poses: Forward Kinematics

- La forma más **simple** de cambiar la pose es cambiando la rotación y posición de los huesos
- Cambiar la posición de un hueso **cambia la posición de todos sus hijos**
 - Similar a la jerarquía de Unity
 - Este proceso se denomina **Forward Kinematics**

Posición
Inicial

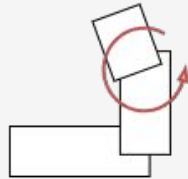
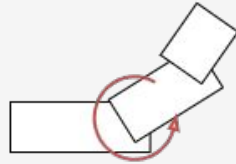
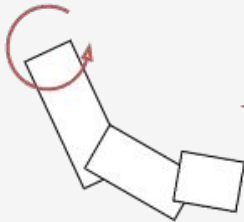


Posición
Final

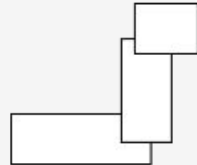
Poses: Forward Kinematics

Forward Kinematics (FK): Calcular la pose de los huesos hijos basado en la pose de sus padres

Posición
Inicial



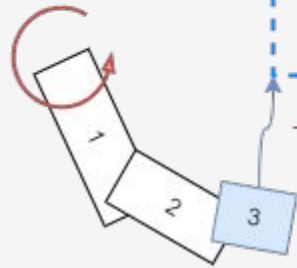
Posición
Final



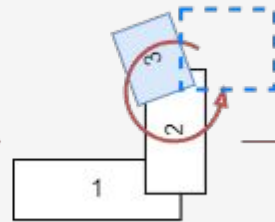
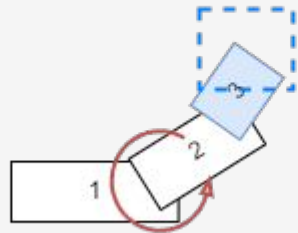
Poses: Forward Kinematics

- ¿Qué pasa si queremos ubicar una parte del cuerpo en una **posición objetivo**?
 - Identificamos el **primer hueso** en la cadena afectada
 - Partiendo de este, vamos cambiando posición y rotación hasta que el último hueso está en su posición deseada

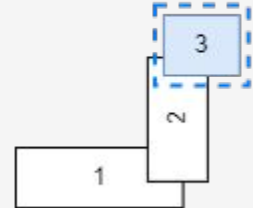
Posición
Inicial



Objetivo



Posición
Final



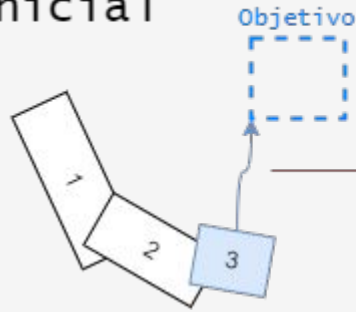
Poses: Forward Kinematics

- ¿Podemos saber la posición final del primer hueso por adelantado?
 - No es posible, normalmente movemos todos los huesos un poco hasta que conseguimos la pose deseada
- ¿Y qué tal el último?
 - Sí, es el objetivo, así que su posición es trivial
 - Con el último hueso de la cadena bien ubicado, el penúltimo también es fácil de ubicar
 - Y así sucesivamente hasta el primer hueso de la cadena

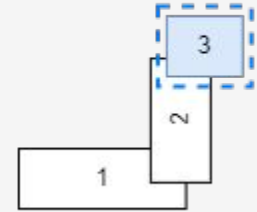
Poses: Inverse Kinematics

- ¿Entonces qué tal si **invertimos** el proceso?
 - Ponemos el último hueso de la cadena en su posición deseada
 - Ajustamos los demás para conseguir una pose coherente
 - Este proceso se denomina **Inverse Kinematics**

Posición
Inicial



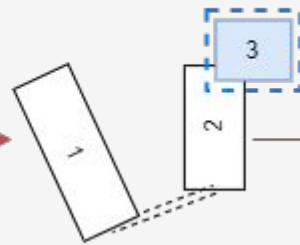
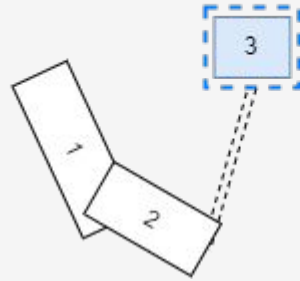
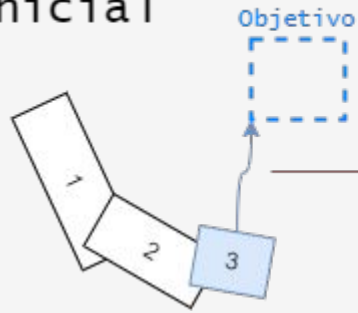
Posición
Final



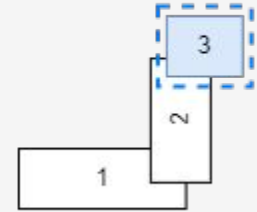
Poses: Inverse Kinematics

Inverse Kinematics (IK): Calcular la pose de los huesos basado en la posición del último hueso de la cadena

Posición
Inicial

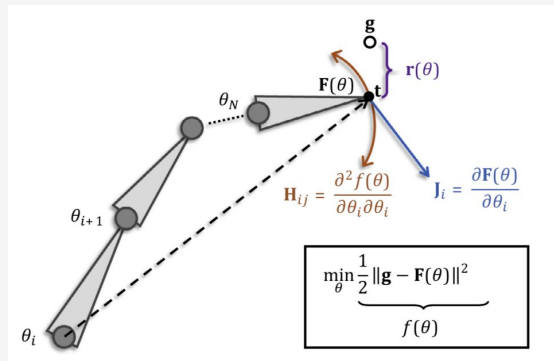


Posición
Final



Poses: Inverse Kinematics

- Resulta que **podemos automatizar este proceso**
 - Blender (y cualquier software de modelado 3D) te permite añadir **restricciones** entre los huesos
 - El ordenador puede calcular la pose que mejor satisface esas restricciones **automáticamente**
 - El artista solo provee los **objetivos** y las **restricciones**



Poses: Inverse Kinematics

Para trabajar con inverse Kinematics, se parte de un modelo con **Rig** y se le añaden **restricciones** y **controladores de IK**.

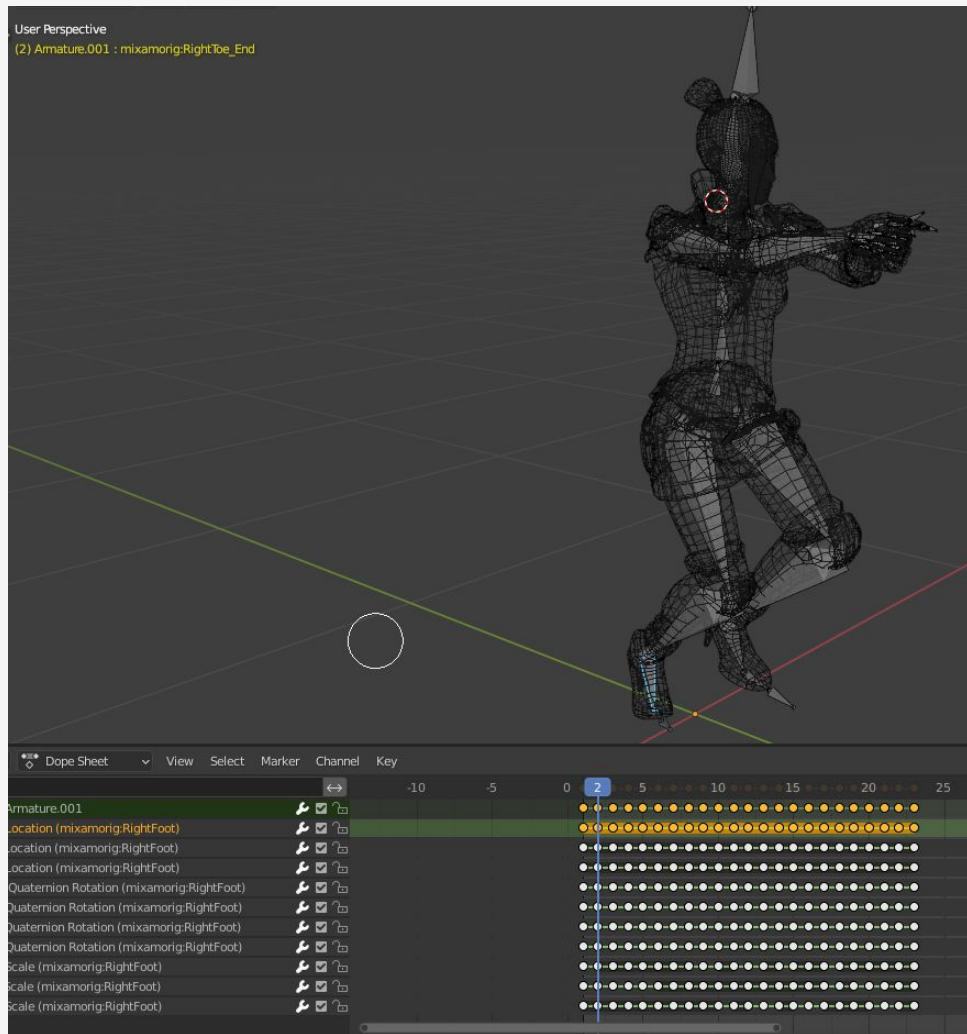
Rig: La cadena de huesos asociada al modelo

Restricciones: Son límites que se le indican a propiedades de los huesos: “Este ángulo no puede pasar de 90 grados”

Controladores IK: Son huesos especiales que se usan para definir **objetivos** para el Rig: “La mano debe estar en esta posición y rotación”

Animaciones

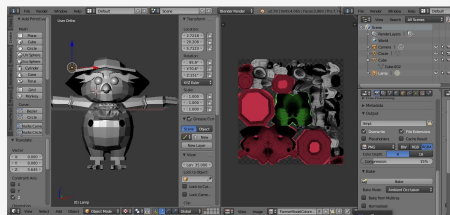
Podemos hacer animaciones animando las propiedades de los controladores de IK y huesos. Blender calculará la posición del resto del Rig automáticamente



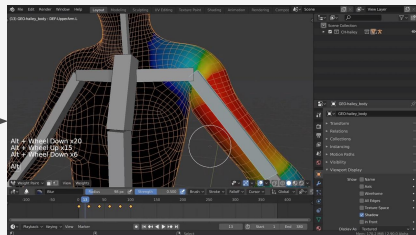
[illegible]

Animaciones estáticas

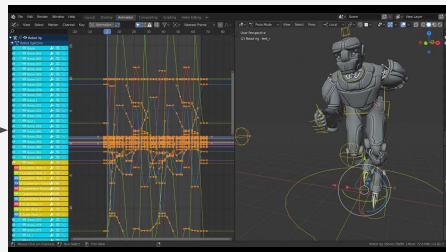
- Las animaciones se crean en un software como Blender
 - Usando cualquier combinación de FK o IK
- Ya exportadas, se usan en Unity con el *Animator*
- Se reproducen en Unity tal como se hicieron, salvo cambios por blending
 - Este estilo se conoce como **animaciones estáticas**



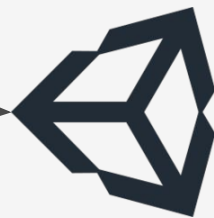
Modelado y
texturizado



Rigging y weight
painting



Animación



Animaciones estáticas

Animaciones estáticas: Animaciones que se reproducen tal como se produjeron en el software de animación

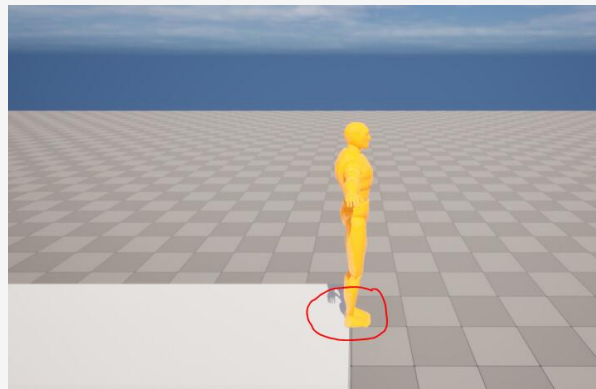
- Como Hemos estado trabajando hasta ahora



Animaciones estáticas

Las animaciones estáticas tienen algunos problemas:

1. No se adaptan bien a entornos dinámicos o con superficies complicadas
2. No se adaptan bien a todas las jugabilidades
 - a. Ejemplo: Shadow of the colossus, Gang Beasts
3. Dificultan crear sensaciones orgánicas



Animación procedural

Si recordamos unas páginas atrás...

Poses: Inverse Kinematics

- Resulta que podemos automatizar este proceso
 - Blender (y cualquier software de modelado 3D) te permite añadir restricciones entre los huesos
 - El ordenador puede calcular la pose que mejor satisface esas restricciones automáticamente
 - El artista solo provee los objetivos y las restricciones

¿Y si hacemos esto en tiempo real?



Animación procedural

¿Podemos usar IK para hacer animaciones en tiempo real... En Unity?

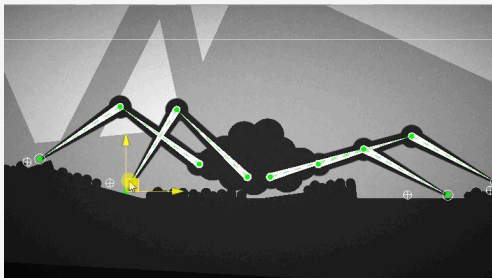
Sí, con un modelo con Rig, Unity permite aprovechar el esqueleto para hacer animaciones en tiempo real:

- Contextuales
- Dinámicas
- Personalizables
- **Programables**

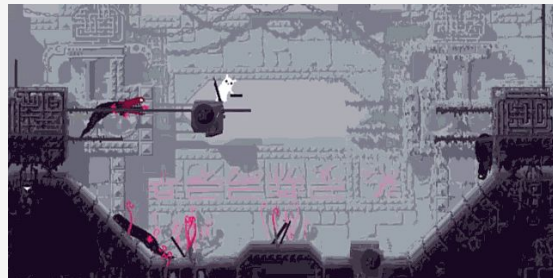
Animación procedural

Animación procedural: Animaciones que se calculan en tiempo de ejecución, normalmente dirigidas por un **programa**

- Permiten adaptar personajes al terreno
- Se integran con animaciones estáticas: ¡No son exclusivas!
- Nuevas opciones de jugabilidad
- Experiencias más orgánicas e inmersivas



Tom Stephenson,
[The Dark Cave](#)



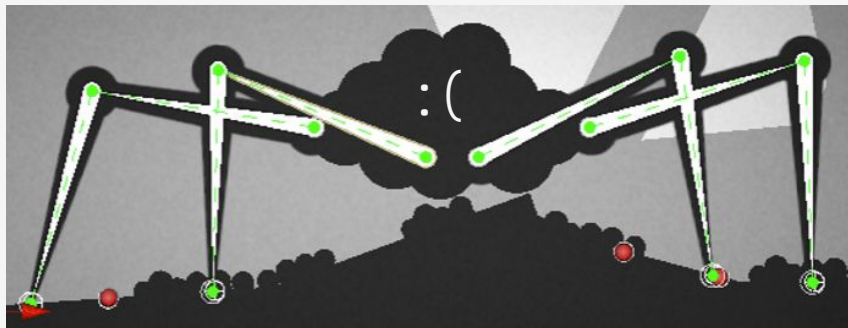
[Rain World](#)

Animación procedural

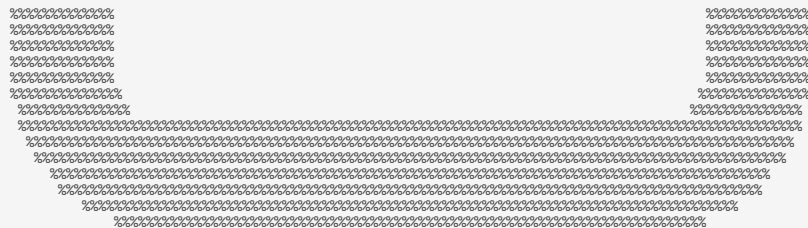
¿Cómo usar animación procedural?

Si ya estamos usando personajes con Rig y personajes humanoides, **no necesitamos nada más!**

Si queremos darle un Rig peculiar a una creatura, se complica un poco, así que nos enfocaremos en **personajes humanoides** con un **Rig ya definido**



Workflow de importación



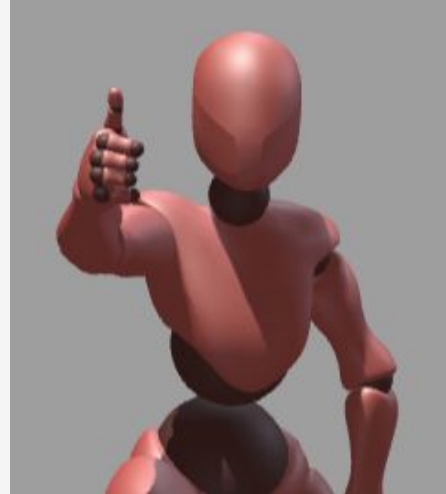
Importación

Importar modelos para ser usados con IK en Unity no es trivial

A partir de ahora asumimos tenemos un modelo:

1. Humanoide
2. Con Rig, restricciones y Weight Painting
3. Con animaciones

Es fácil conseguir un modelo así de [Mixamo](#)



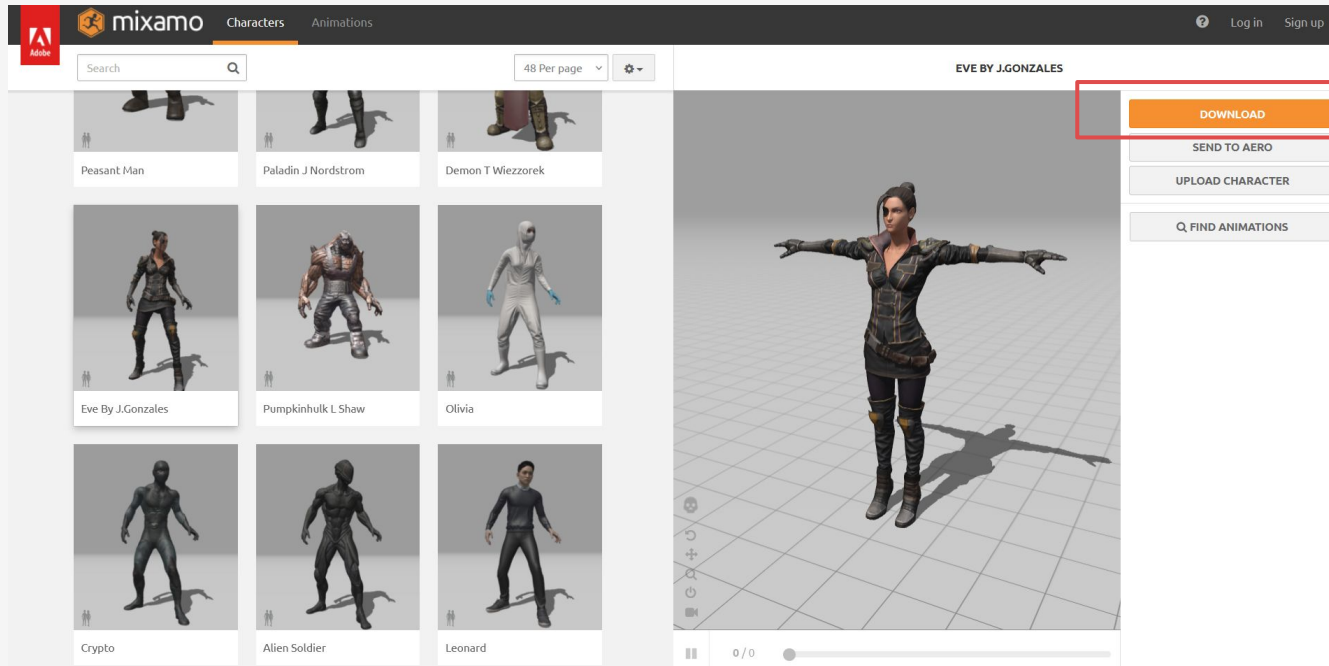
Importación

Empezaremos importando un personaje



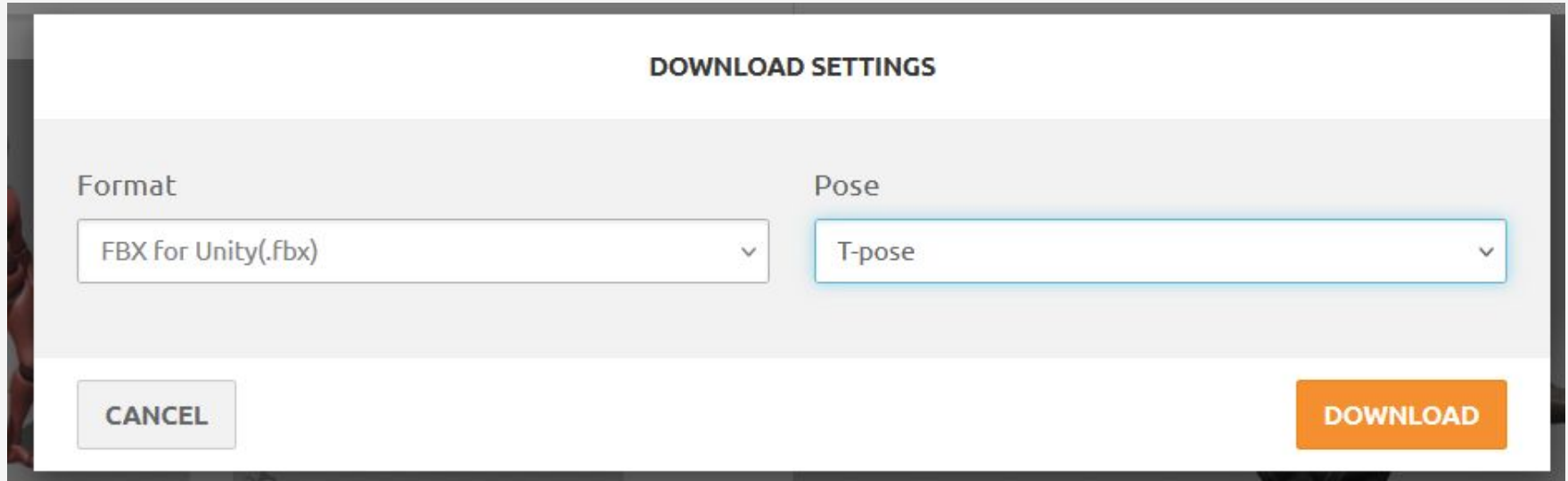
Mixamo: Modelo del personaje

1. Seleccionar el modelo que deseas y click en “descargar”



Mixamo: Modelo del personaje

2. Seleccionar: Formato = Unity FBX y Pose = T-Pose



DOWNLOAD SETTINGS

Format

FBX for Unity(.fbx) ▼

Pose

T-pose ▼

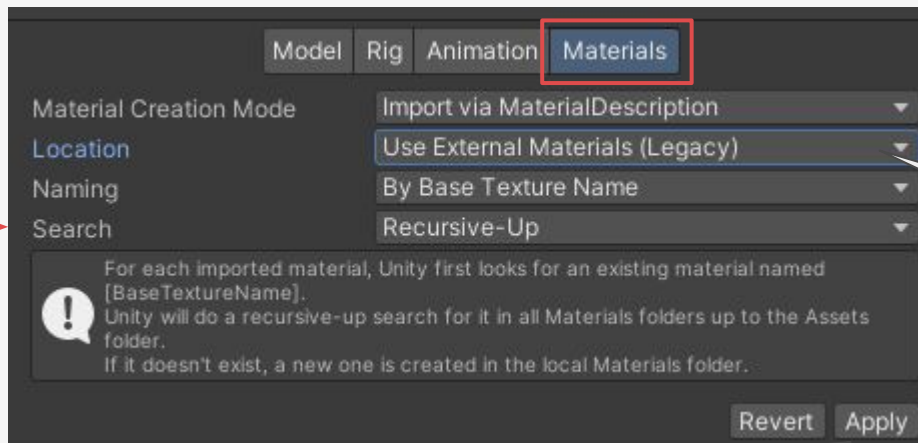
CANCEL

DOWNLOAD

Mixamo: Modelo del personaje

3. Importar en Unity

⚠ Si tienes problemas con el renderizado de las texturas del personaje, selecciona el asset, pestaña de **materiales...**



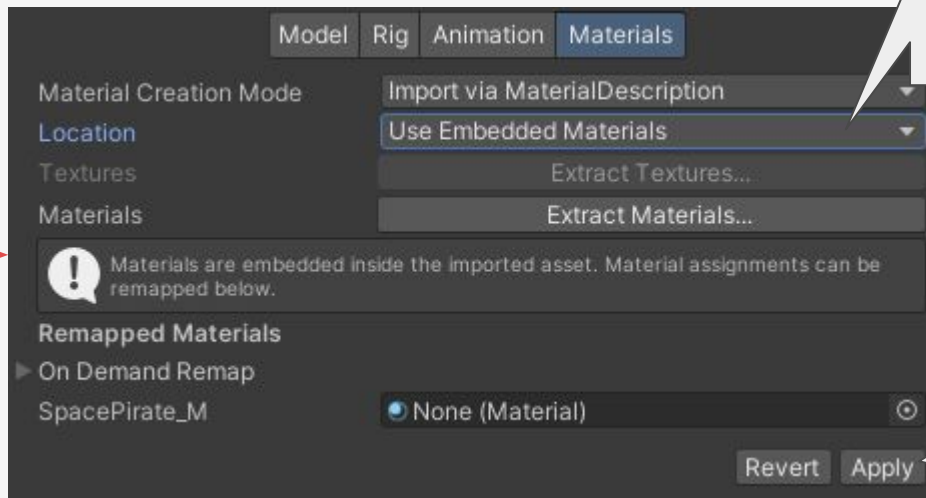
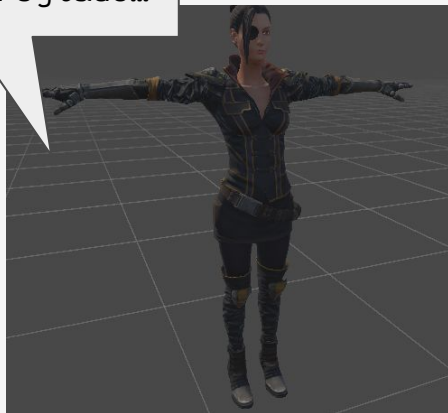
... y cambia *location* a "use external materials (legacy)"

Y aplicar

Mixamo: Modelo del personaje

⚠ Si tienes problemas con el renderizado de las texturas del personaje...

Ahora con
el modelo
arreglado...



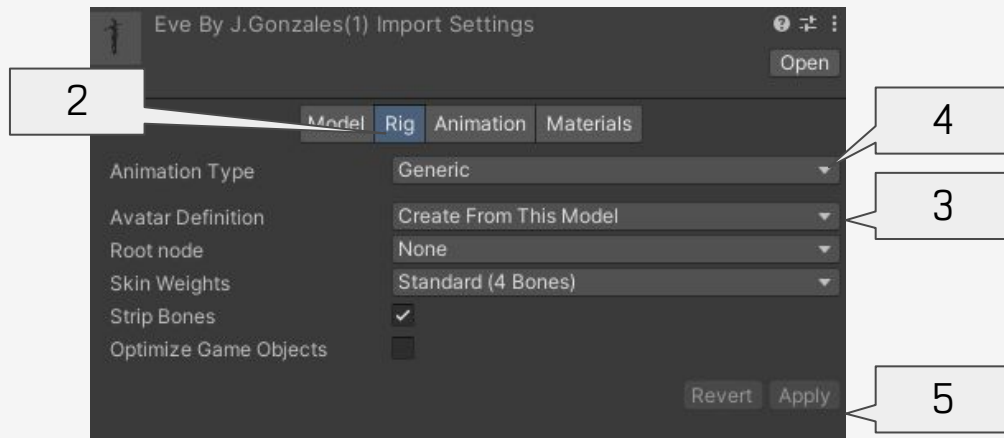
Volvemos a
cambiar
location de
vuelta a “use
embedded
materials”

Y volvemos
a aplicar

Mixamo: Modelo del personaje

4. Crear el “Avatar”:

1. Click en el Asset del personaje
2. Ir a “RIG”
3. Cambiar: Avatar Definition = Create from this model
4. Cambiar: Animation type = Humanoid
5. Apply



Mixamo: Modelo del personaje

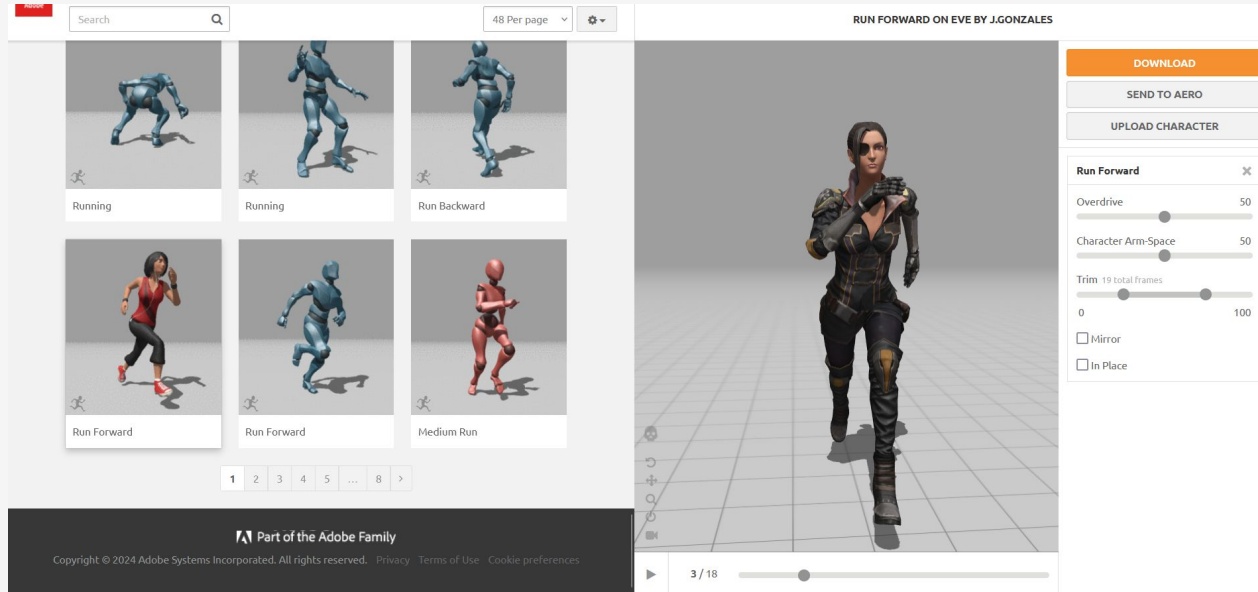
Esto nos genera un asset tipo avatar en el asset del modelo



Ahora podemos importar las animaciones

Mixamo: Animaciones

1. Escoger las animaciones y click en descargar



Mixamo: Animaciones

2. Configurar: Format = FBX for Unity, Skin = Without Skin

DOWNLOAD SETTINGS

Format

FBX for Unity(.fbx) ▾

Skin

Without Skin ▾

Frames per Second

30 ▾

Keyframe Reduction

none ▾

CANCEL

DOWNLOAD

Mixamo: Animaciones

2. Configurar: Format = FBX for Unity, Skin = Without Skin

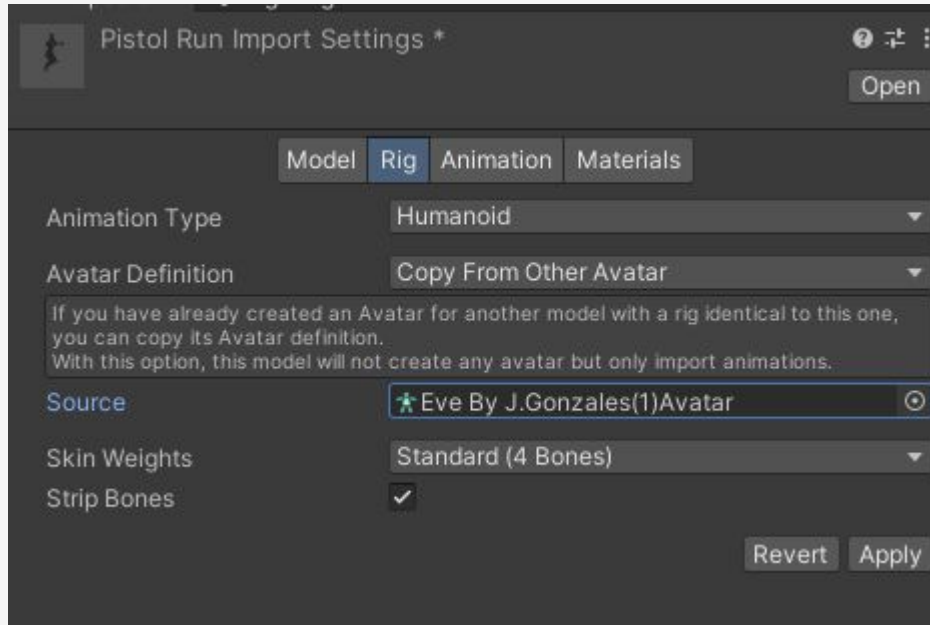
No incluimos el skin porque el modelo que importamos ya lo tiene

Skin: El *weight painting*. Son datos adicionales en los vértices de la mesh que indican qué huesos y con cuanto peso afectan a ese vértice.

Mixamo: Animaciones

2. Importar las animaciones a Unity. Por cada asset de animación:

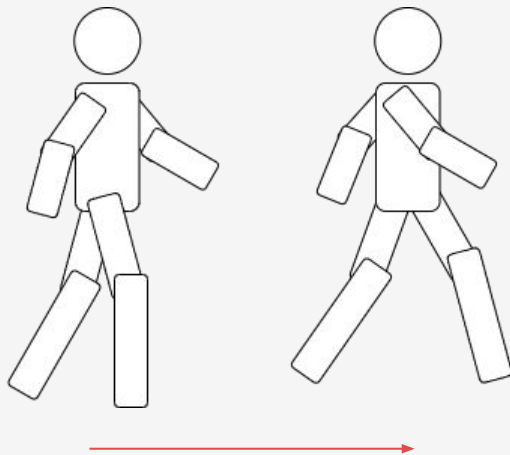
1. Ir a **Rig**
2. Animation type =
Humanoid
3. Avatar definition =
Copy from Other Avatar
4. Source = El avatar que creamos anteriormente
5. Aplicar



Unity: Bake into Pose y Root Motion

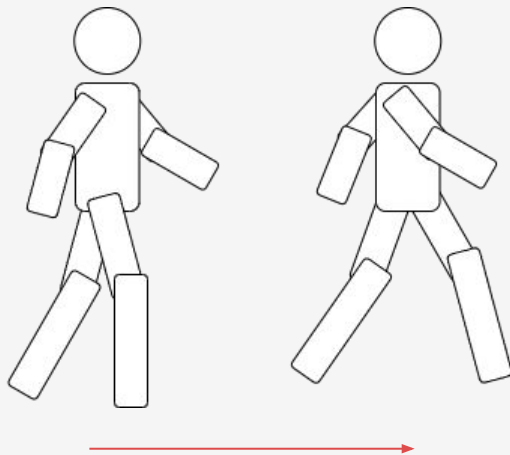
Las animaciones no solo modifican la posición de los huesos, también pueden modificar la posición de todo el modelo

- Es como la diferencia entre correr en una cinta y correr sobre el suelo



Unity: Bake into Pose y Root Motion

Esto se debe al **root motion**. Los Rigs siempre tienen un hueso raíz que es padre de todos los huesos, y transformarlo también transforma el resto del modelo



Unity: Bake into Pose y Root Motion

Root Motion: Animación del hueso raíz, que afecta la rotación y posición global del modelo

Cuando **NO** es conveniente:

1. Controlar el movimiento del personaje por código (ej. correr, caminar, saltar)
2. Interacción con ambiente
3. Juegos en línea con sincronización

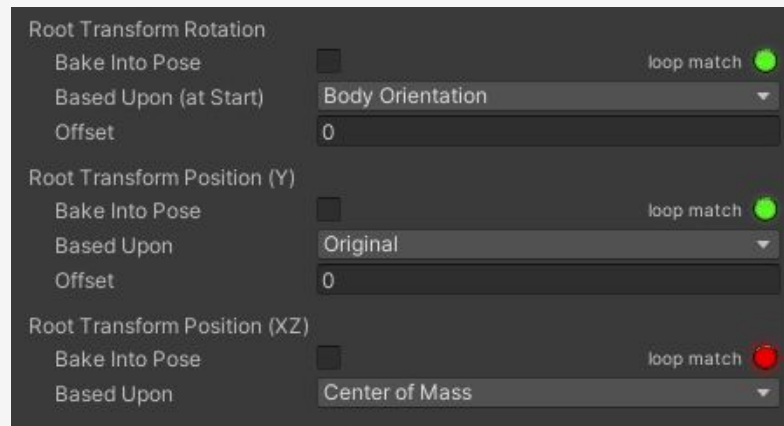
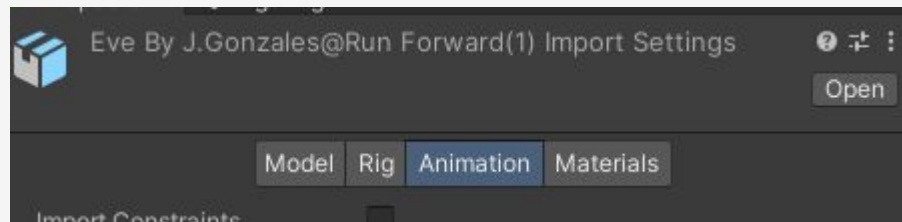
Cuando **SÍ** es conveniente:

1. Traslaciones complicadas. Ejemplo: animaciones de ataque en Dark Souls
2. Cinemáticas
3. Comportamiento de los NPC

Unity: Bake into Pose y Root Motion

Unity permite escoger si importar el root motion o no.

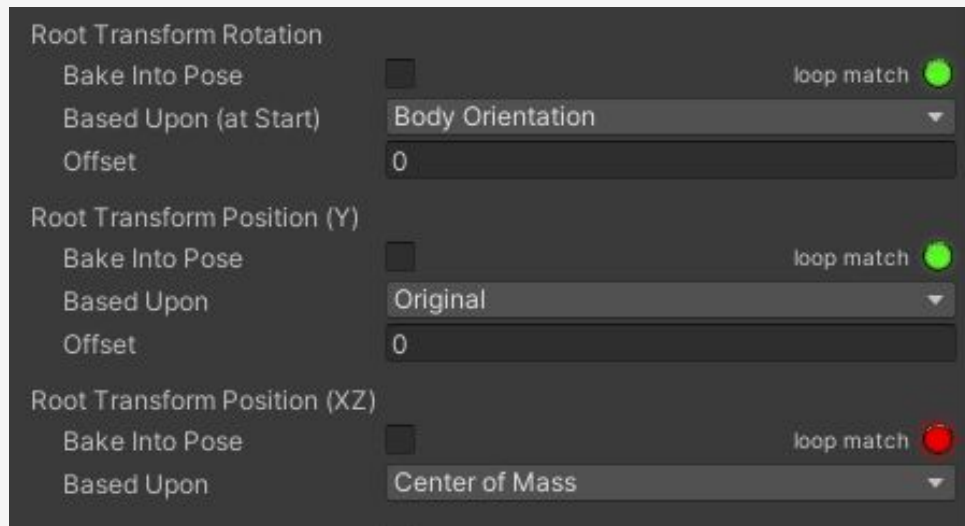
1. En el asset de animación ir a la pestaña “Animation”
2. En la parte inferior de ese menú, tenemos varias opciones de root transform



Unity: Bake into Pose y Root Motion

Seleccionar “Bake Into Pose” anula el root motion de ese parámetro.

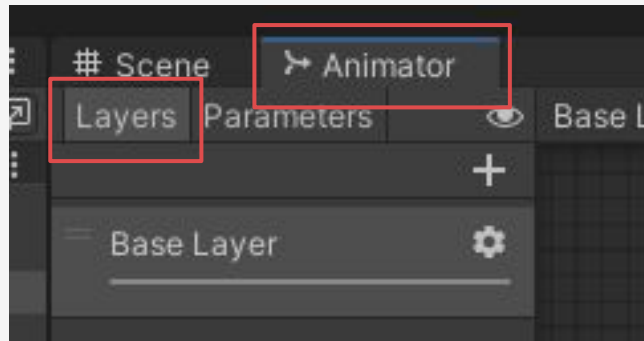
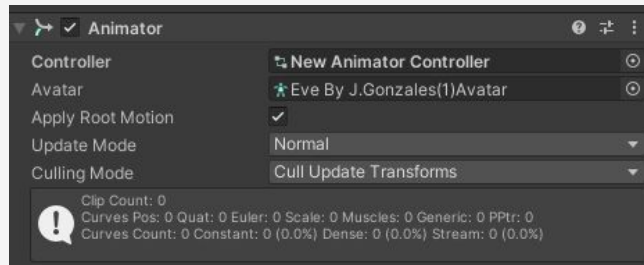
1. Root transform rotation:
Anula la rotación
2. Root Transform Position (Y): Anula traslación en el eje Y (altura)
3. Root Transform Position (XZ): Anula traslación en el plano XZ (horizontal)



Unity: Animator

Finalmente, el personaje en escena debe tener configurado un animator

1. Crea un GameObject con el modelo del personaje en escena (o arrastra el modelo a escena directamente)
2. Añade un animator con un controlador de animación
3. En la ventana de animator, ir a la pestaña **layers**

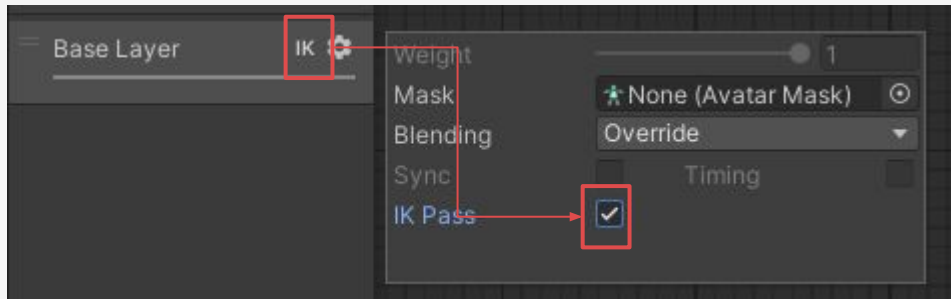


Unity: Animator

Finalmente, el personaje en escena debe tener configurado un animator

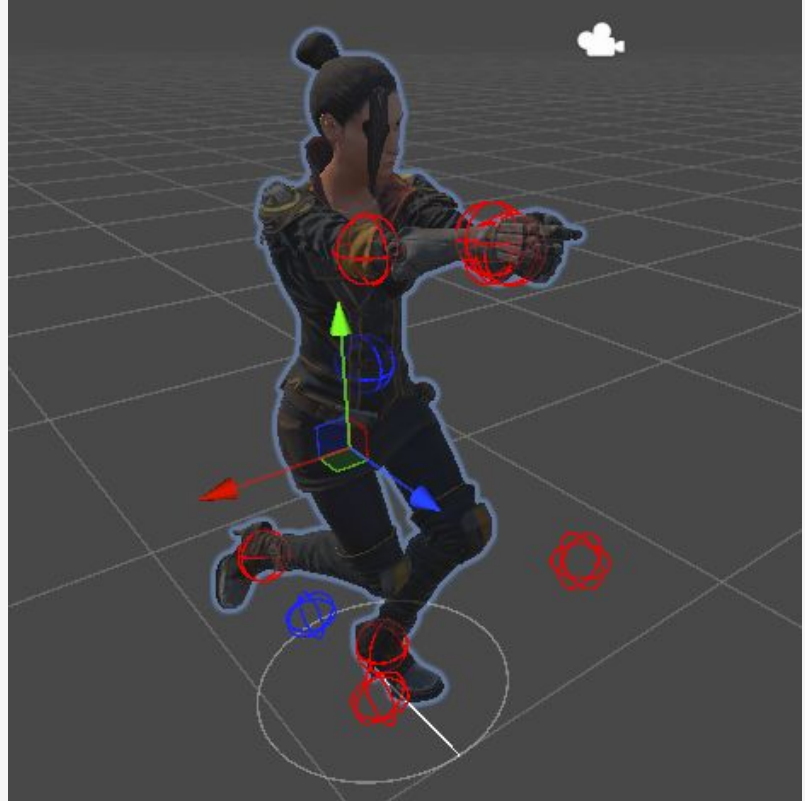
4. En la primera capa, ve al ícono de engranaje, que despliega un menú adicional

5. En este menú, activa “IK Pass”



Unity: Animator

Podemos añadir una animación para verificar que todo funcionó correctamente



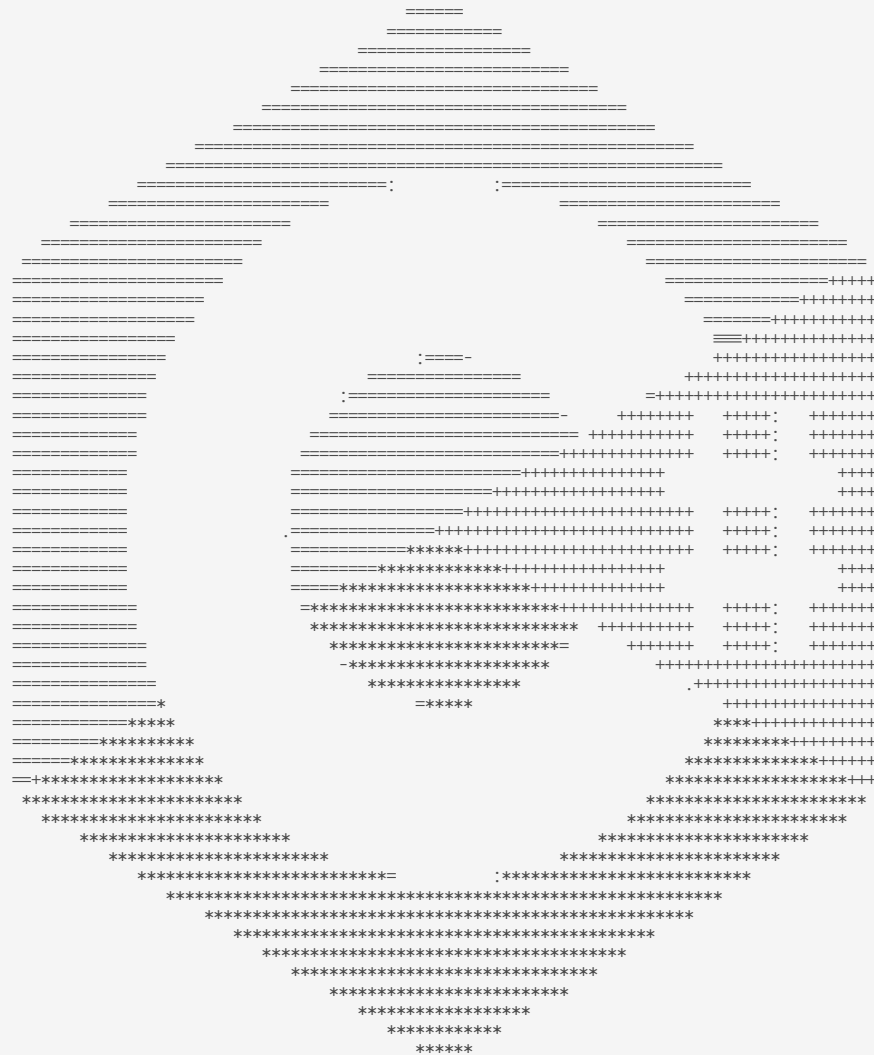
Unity: Animator

¡Con esto estamos listos para empezar a usar las funcionalidades de animación procedural de Unity!

HERE WE GO!

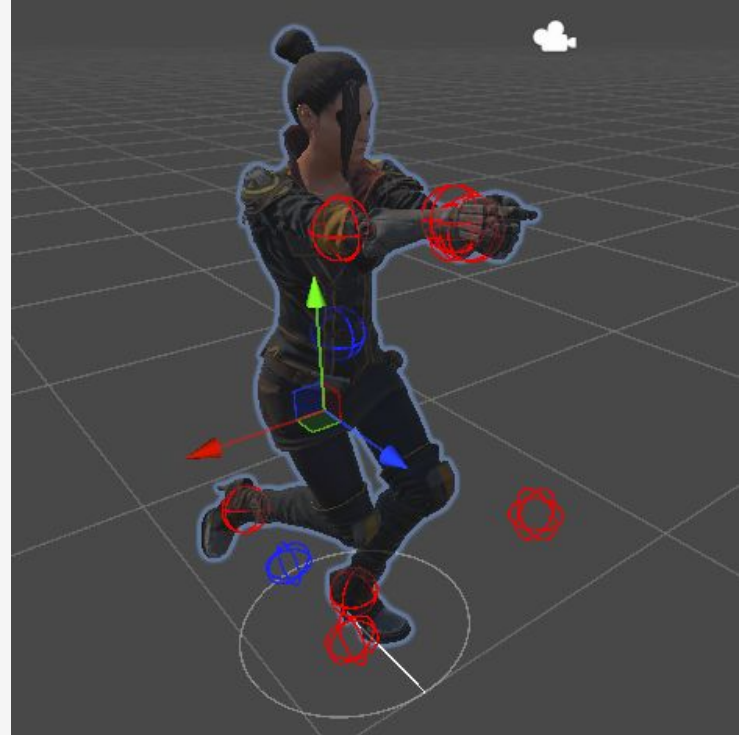


Animando desde C#



IK Goal y IK Target

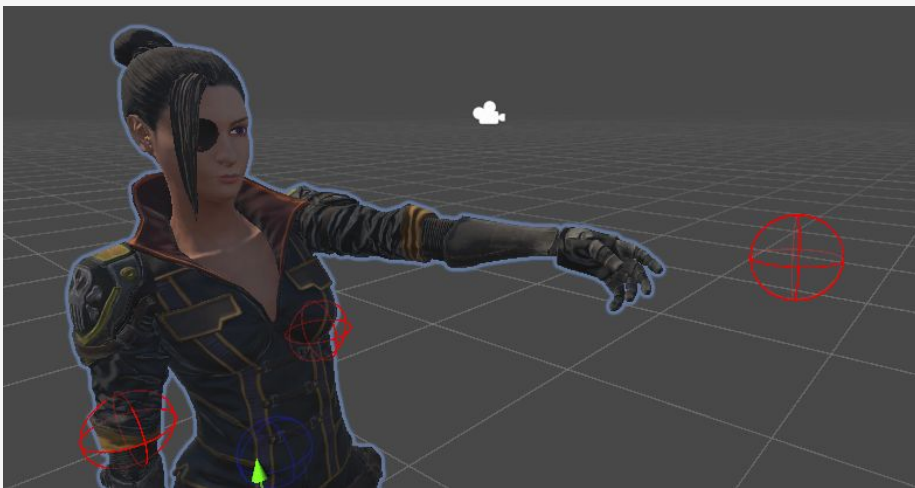
- Los **IK Goal** y **IK Hint** son objetos imaginarios que el animator usa para calcular la pose final del modelo
- En unity los podemos visualizar con estos Gizmos al seleccionar el personaje:



IK Goals y IK Target

IK Goal: Objetos imaginarios que le indican al animator donde *debería* ubicar una parte del cuerpo

- La parte del cuerpo no necesariamente terminará ahí
- Unity hará su mejor esfuerzo para satisfacer el objetivo

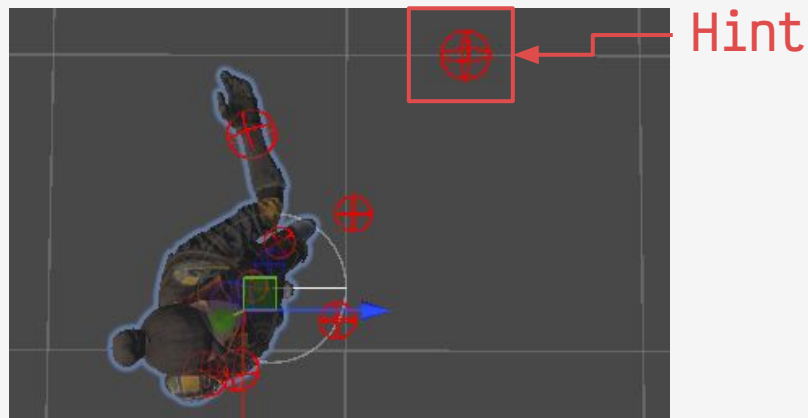


IK Goals y IK Target

- **IK Hint:** Objetos imaginarios que **proveen información adicional** para calcular la pose
 - Se usan para configurar la orientación de los miembros



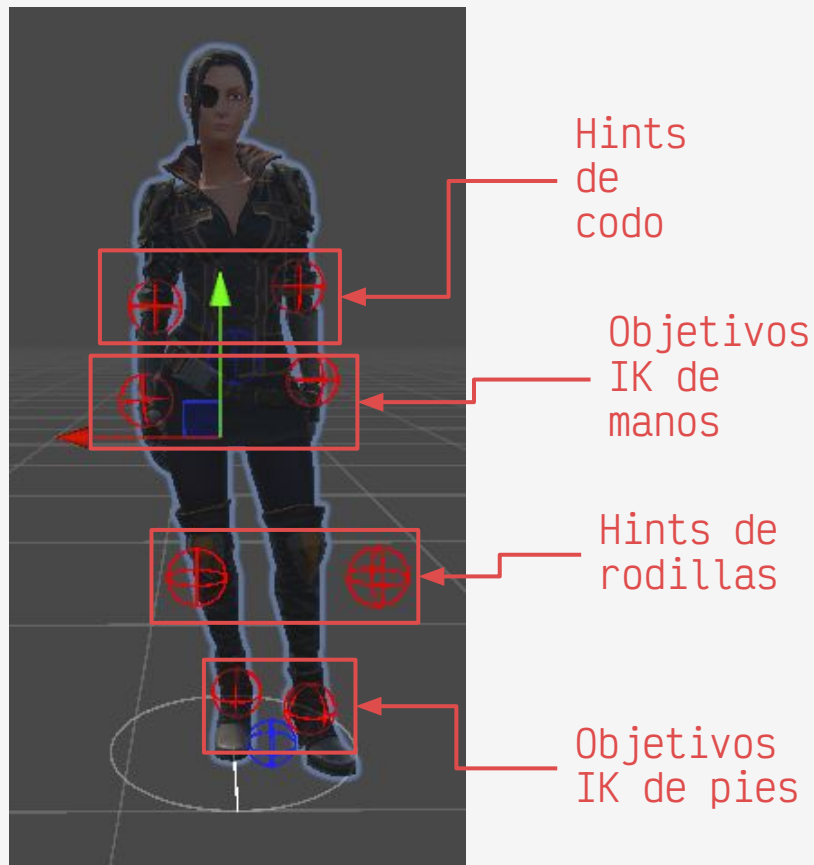
Brazo en posición
correcta



Brazo “Dislocado”

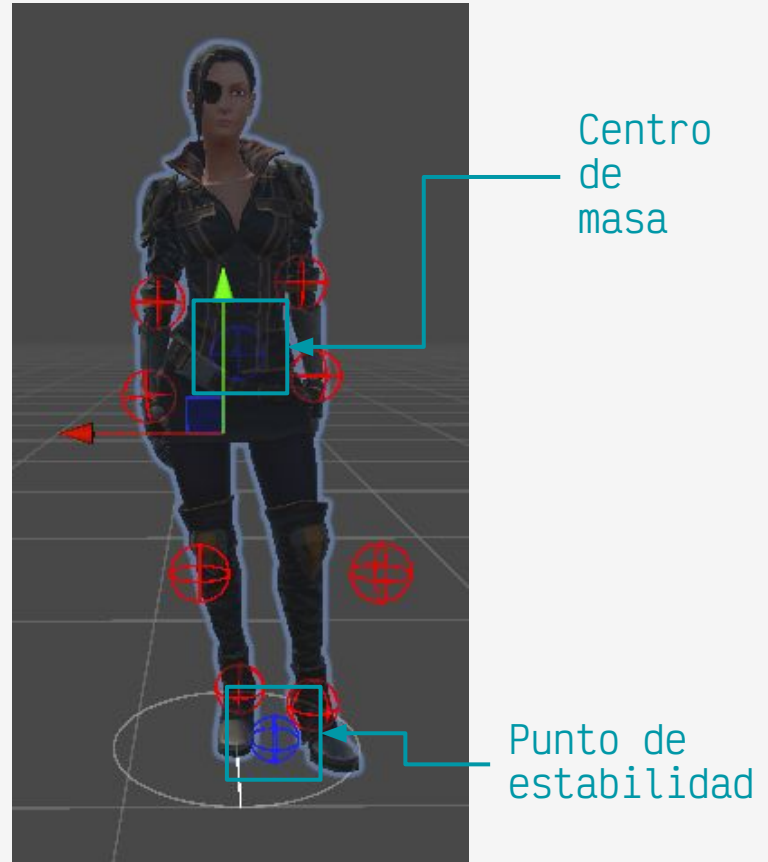
IK Goals y IK Target

- Los IK Goal de las extremidades están en los extremos de cada extremidad (manos, pies)
- Los hints están en los puntos de articulación principales (codos y rodillas)



IK Goals y IK Target

- Unity también nos indica el centro de masa y el punto de equilibrio entre los pies
- No los usaremos, pero son útiles como punto de referencia para hacer cálculos



IK Goals y IK Target

IK Target: Es la posición final que tendrá una parte del cuerpo luego del cálculo de IK (No confundir con IK Goal)

- Primero se calcula la pose de según la animación estática
- Luego se calcula la pose por IK considerando los IK hints y IK goals
- Eso produce la rotación y posición final de los huesos



Pose base

Animación estática



IK Pass



IK Target

OnAnimatorIK

Vamos a añadir un script a nuestro personaje **al mismo nivel del animator**.

En ese Script, podemos añadir la siguiente función:

```
private void OnAnimatorIK(int layerIndex)
{
    // ...
}
```

OnAnimatorIK

Esta es una **función de actualización**, como el update.

- Permite añadir código que se ejecuta cuando el animator hace la pasada de IK, que activamos anteriormente.

 Si no activamos IK Pass en el animator, esta función se ignora

```
private void OnAnimatorIK(int layerIndex)
{
    // ...
}
```

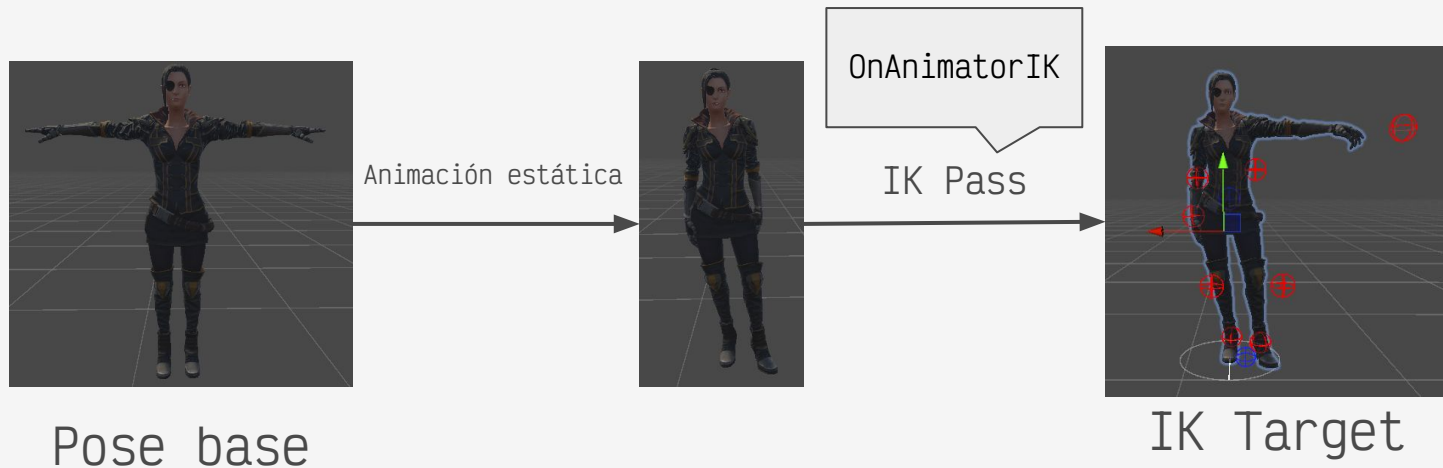
OnAnimatorIK

El argumento “layerIndex” indica el número de la capa de animación. Por ahora no lo usaremos para nada

```
private void OnAnimatorIK(int layerIndex)
{
    // ...
}
```

OnAnimatorIK

Se ejecuta luego del cálculo de la pose por la animación estática y antes del cálculo de IK



OnAnimatorIK

Permite llamar a las **funciones de controladores IK** del animator. En particular, nos interesan:

- SetIKPositionWeight
- SetIKRotationWeight
- SetIKPosition
- SetIKRotation
- SetLookAtPosition
- SetLookAtWeight



[Docs](#)

Posición y rotación

```
void SetIKPosition(  
    AvatarIKGoal goal,  
    Vector3 goalPosition  
)
```

SetIKPosition: Cambia la posición de un objetivo IK

- **goal:** El objetivo IK cuya posición cambiará, especificado por el enum “AvatarIKGoal”
- **goalPosition:** La posición final del objetivo IK

Posición y rotación

```
void SetIKRotation(  
    AvatarIKGoal goal,  
    Quaternion goalRotation  
)
```

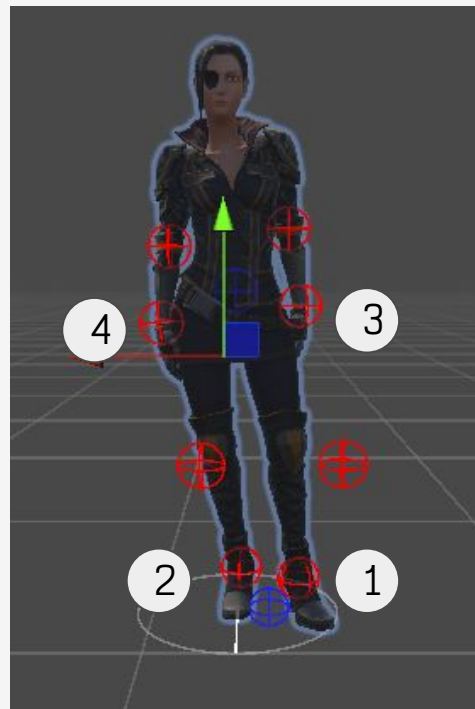
SetIKRotation: Cambia la rotación de un objetivo IK

- **goal:** El objetivo IK cuya rotación cambiará, especificado por el enum “AvatarIKGoal”
- **goalRotation:** La rotación final del objetivo IK, especificada como quaternion

Posición y Rotación

Las posibles variantes del enum `AnimatorIKGoal` son:

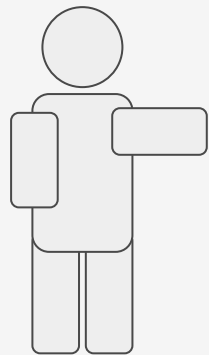
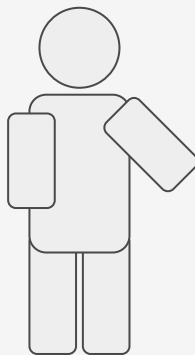
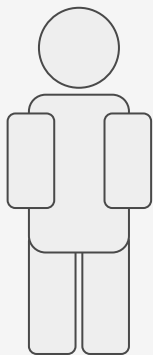
1. `AnimatorIKGoal.LeftFoot`
2. `AnimatorIKGoal.RightFoot`
3. `AnimatorIKGoal.LeftHand`
4. `AnimatorIKGoal.RightHand`



Peso

Las funciones `SetIKPositionWeight` y `SetIKRotationWeight` se usan para asignar peso al IK Goal respectivo.

El **peso** es un valor entre 0 y 1 que indica **qué tan cerca del IK Goal** debe estar el IK target resultante



Peso: 0

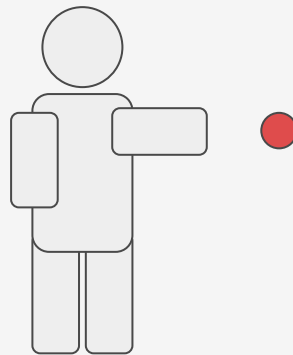
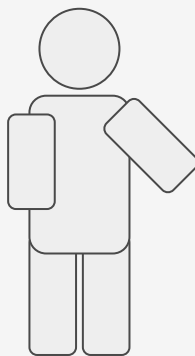
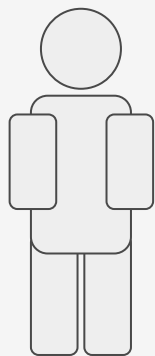
0.5

1

Peso

El peso te permite **escoger un IK Target intermedio** entre la pose de animación estática y la pose con IK Target en el IK Goal

Es útil para hacer animaciones y lograr transiciones de estado fluidas



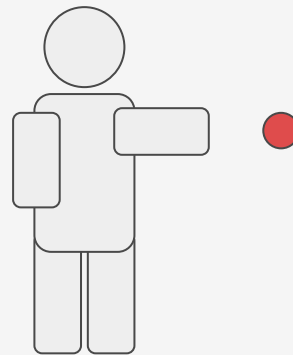
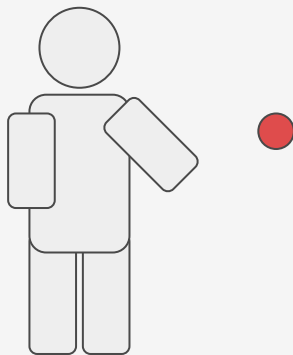
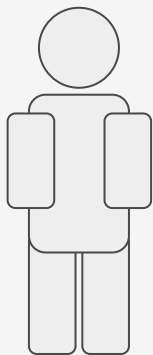
Peso: 0

0.5

1

Peso

El peso puede ser **de posición o rotación**, te permite indicar qué tanto se considera la posición o la rotación por separado



Peso: 0

0.5

1

Peso

```
void SetIKPositionWeight(  
    AvatarIKGoal goal,  
    float value  
)
```

SetIKPositionWeight: El peso de la posición de un objetivo IK

- **goal:** El objetivo IK, especificado por el enum “AvatarIKGoal”
- **value:** Nuevo valor de peso

Peso

```
void SetIKRotationWeight(  
    AvatarIKGoal goal,  
    float value  
)
```

SetIKRotationWeight: El peso de la rotación de un objetivo IK

- **goal:** El objetivo IK, especificado por el enum “AvatarIKGoal”
- **value:** Nuevo valor de peso

Ejemplo

- Vamos a guardar el componente de animator en el componente que acabamos de crear
- También vamos a añadir un game object en escena para usarlo como objetivo

```
public GameObject target; // lo configuramos en Unity
Animator _animator;
private void Awake()
{
    _animator = GetComponent<Animator>();
}
```

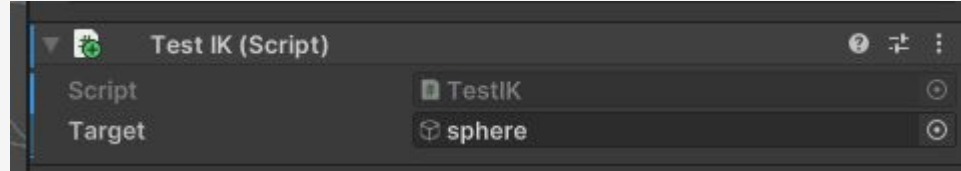
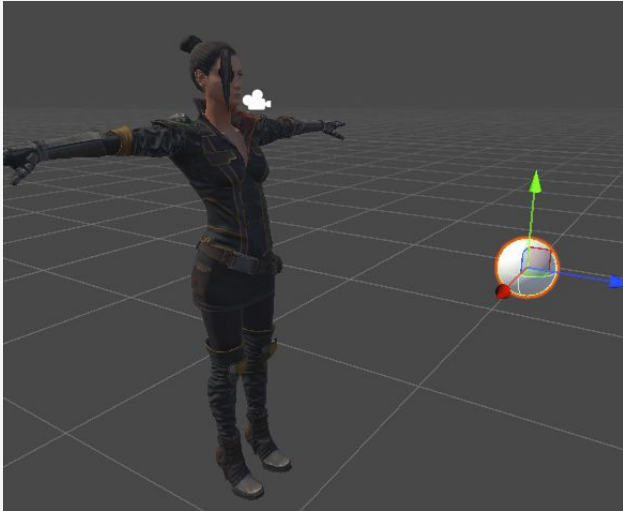
Ejemplo

- Vamos a añadir el siguiente código a la función de OnAnimatorIK

```
private void OnAnimatorIK(int layerIndex)
{
    _animator.SetIKPositionWeight(AvatarIKGoal.LeftHand, 1);
    _animator.SetIKPosition(AvatarIKGoal.LeftHand, target.transform.position);
}
```


Ejemplo

Ahora en Unity, ponemos un objeto cualquiera en escena y se lo asignamos al parámetro “target”



Ejemplo

- El personaje intentará alcanzar el objeto con la mano izquierda
- Mover el objeto en la escena también moverá la mano del personaje



Ejemplo

Una propiedad importante de OnAnimatorIK es que **los cambios hechos en un frame no avanzan al siguiente**

Esto quiere decir que no debemos preocuparnos por restaurar el estado de los IK Goal

Vamos a añadir la siguiente variable al componente:

```
public bool ikActive = true;
```

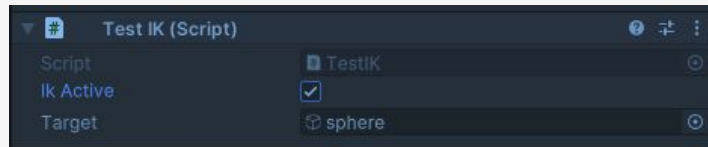
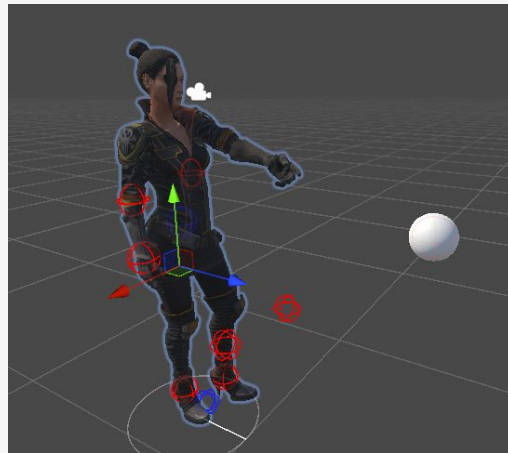
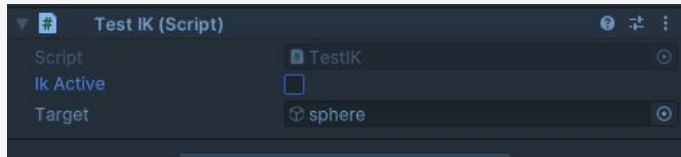
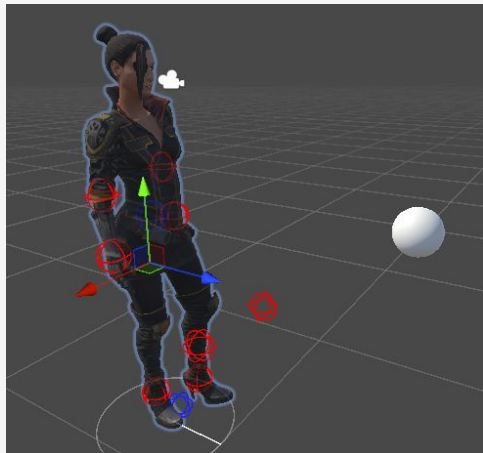
Ejemplo

Y cambiamos OnAnimatorIK

```
private void OnAnimatorIK(int layerIndex)
{
    if (!ikActive)
        return;
    _animator.SetIKPositionWeight(AvatarIKGoal.LeftHand, 1);
    _animator.SetIKPosition(AvatarIKGoal.LeftHand, target.transform.position);
}
```

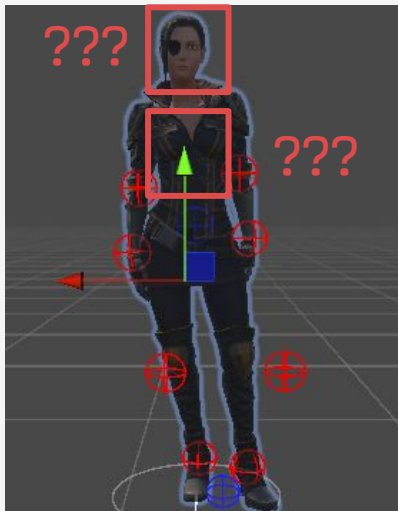
Ejemplo

Y ahora podemos activar y desactivar el IK sin trabajo adicional



Moviendo la cabeza y el cuerpo

- Hasta ahora solo podemos mover los brazos y piernas del personaje
- ¿Qué pasa con la cabeza y el cuerpo?
 - ¡No tienen IK Goal!



```
public enum AvatarIKGoal
{
    | The left foot.
    LeftFoot,
    | The right foot.
    RightFoot,
    | The left hand.
    LeftHand,
    | The right hand.
    RightHand,
}
```

Moviendo la cabeza y el cuerpo

La cabeza y el cuerpo no funcionan igual que piernas y brazos

Se mueven usando la función **SetLookAtPosition**, que funciona similar a todas las funciones *LookAt* que hemos usado hasta ahora



Look At Position

```
void SetLookAtPosition(Vector3 lookAtPosition)
```

lookAtPosition: Posición hacia donde el personaje debe mirar, en coordenadas del mundo



Look At Position

```
public void SetLookAtWeight(  
    float weight,  
    float bodyWeight,  
    float headWeight,  
    float eyesWeight,  
    float clampWeight  
)
```

SetLookAtWeight: Configura el peso de “Mirar”

- Es un shortcut que provee Unity para manipular la posición de los huesos de la cabeza y el cuerpo
- A pesar de su nombre, no solo aplica para la cabeza

Look At Position

```
public void SetLookAtWeight(  
    float weight,  
  
    float bodyWeight,  
  
    float headWeight,  
  
    float eyesWeight,  
  
    float clampWeight  
)
```

weight: El peso global de la acción de Mirar, se multiplica al resto de pesos

body weight: Cuanto se afecta el cuerpo por la acción de mirar (0 por defecto)

head weight: Cuanto se afecta la cabeza por la acción de mirar (1 por defecto)

eyes weight: Cuanto se afectan los ojos por la acción de mirar (0 por defecto)

Look At Position

```
public void SetLookAtWeight(  
    float weight,  
    float bodyWeight,  
    float headWeight,  
    float eyesWeight,  
    float clampWeight  
)
```

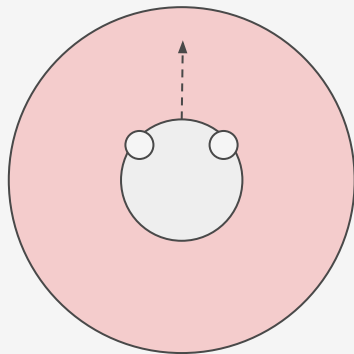
clampWeight: Un valor entre 0 y 1, indica el límite de movimiento para el LookAt.

- 0 significa que no tiene restricción, la cabeza puede girar en 360°
- 1 significa que está totalmente restringido y no puede girar (0° de libertad de giro)
- 0.5 significa que puede girar hasta 180°

Por defecto vale 0.5

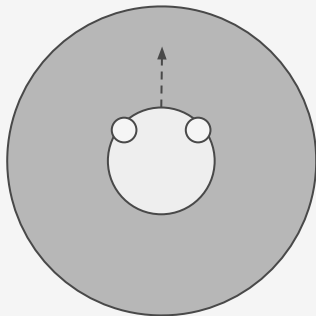
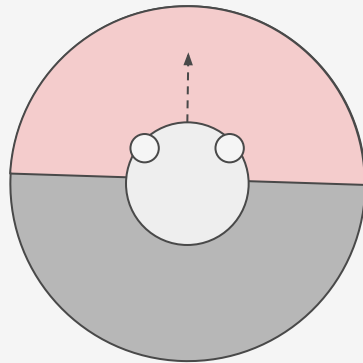
Look At Position

```
public void SetLookAtWeight(  
    float weight,  
    float bodyWeight,  
    float headWeight,  
    float eyesWeight,  
    float clampWeight  
)
```



clampWeight = 0,
360° de libertad de
giro

clampWeight = 0.5,
180° de libertad de
giro



clampWeight = 0, 0°
de libertad de giro

Ejemplo

Vamos a añadir otra variable de clase a nuestro componente de prueba:

```
public GameObject lookAtTarget;
```

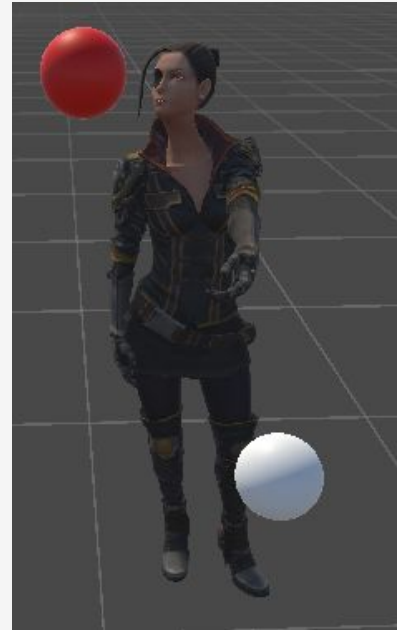
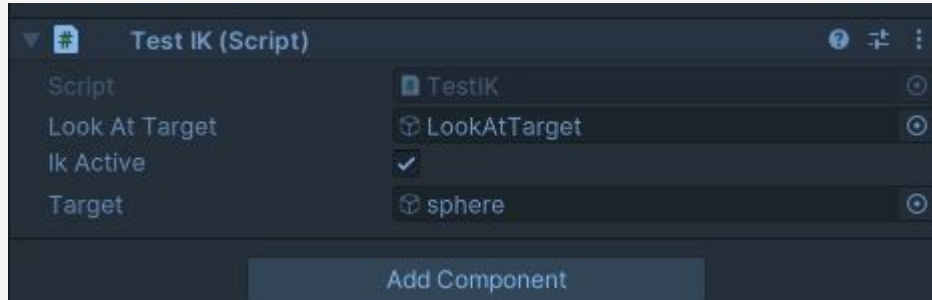
Y luego modificamos la función OnAnimatorIK:

```
private void OnAnimatorIK(int layerIndex)
{
    if (!ikActive)
        return;
    _animator.SetIKPositionWeight(AvatarIKGoal.LeftHand, 1);
    _animator.SetIKPosition(AvatarIKGoal.LeftHand, target.transform.position);

    // Nuevo!
    _animator.SetLookAtWeight(1);
    _animator.SetLookAtPosition(lookAtTarget.transform.position);
}
```

Ejemplo

En unity podemos arrastrar otro objeto a escena y ahora el personaje mirará el objeto



Ejemplo

Podemos usar el parámetro **bodyWeight** para que el cuerpo también forme parte del movimiento

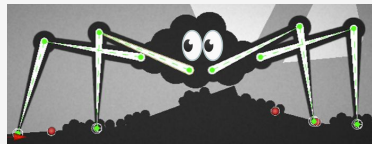
```
_animator.SetLookAtWeight(1, 1);
```



¿Cómo se usan estas funciones?

Las funciones de IK son simples para ser usadas en muchos escenarios. Ejemplo:

1. Usar la posición de otro objeto para seguirlo con la mirada o señalar
2. Usar Raycast desde la posición de los IK Goal para ajustar la posición de los miembros a superficies
 - a. Usando la función **GetIKPosition** para calcular el rayo
3. Usar objetos invisibles para dirigir los IK Goals
4. Usar el centro de masa y punto de estabilidad para hacer locomoción procedural



Consejos y recomendaciones

1. **Pesos de 1 generan resultados poco naturales**, trata de evitar pesos en 1 para efectos más suaves
 - a. ¡Y más si se trata del pecho!
2. **En lugar de asignar el peso, trata de animarlo** para generar animaciones procedurales más naturales
 - a. ¡Dotween es una gran ayuda!
3. Aunque la cabeza no tenga un IK Goal, puedes **simular una funcionalidad** similar usando un objeto invisible delante del personaje

Consejos y recomendaciones

4. **Centraliza el cálculo de IK:** Si tienes varios componentes generando efectos de IK sobre el personaje, algunos podrán sobrescribir otros sin ningún orden en particular

Debes coordinar este cálculo para evitar artefactos de animación

Calidad de vida

Cuando tenemos personajes con Rig es normal usar huesos para sujetar elementos del juego, como armas o equipamiento

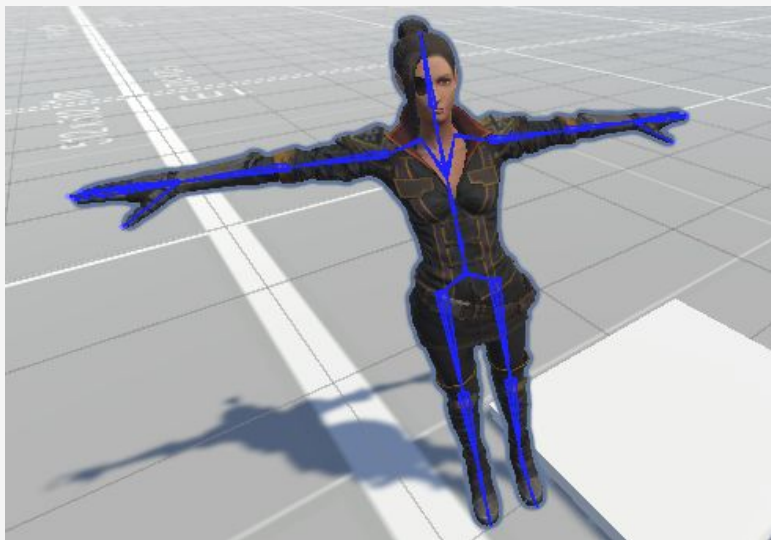
Con animaciones procedurales también es útil, para añadir interacción a ciertos huesos

Pero para **seleccionar el hueso correcto**, hay buscarlo en la jerarquía del modelo manualmente



Animation Rigging al rescate

Pero ya que estamos usando animación procedural y personajes con Rig, podemos aprovechar el paquete de Unity: **Animation Rigging**

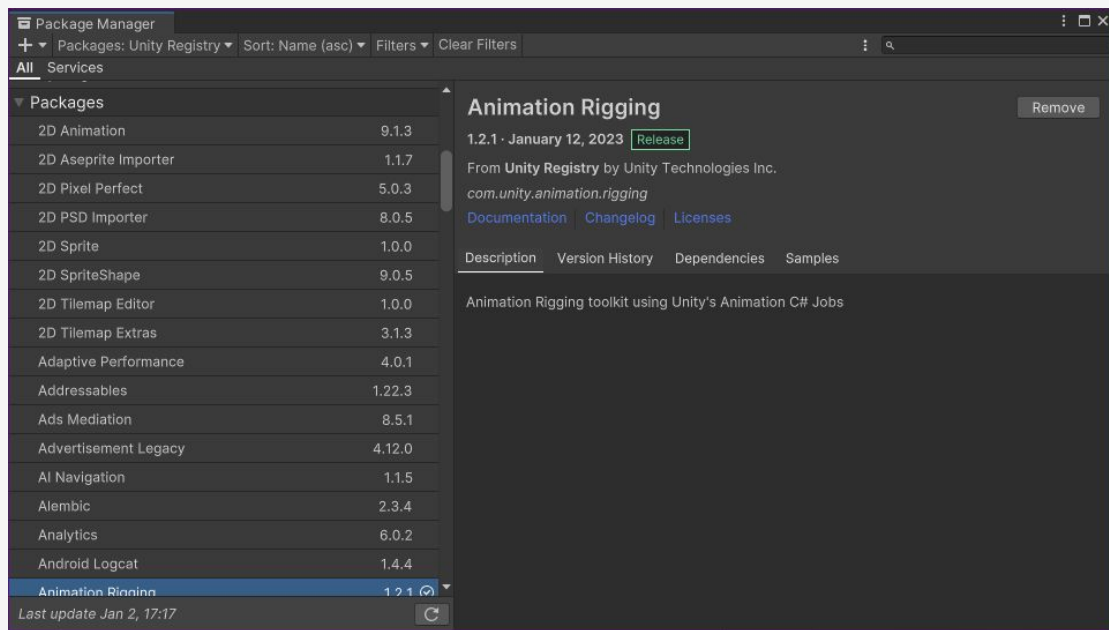


[Docs](#)

Animation Rigging

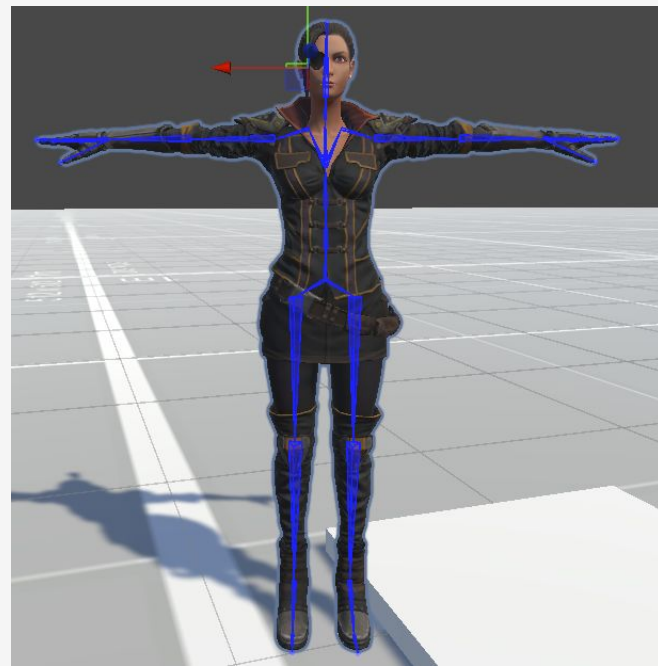
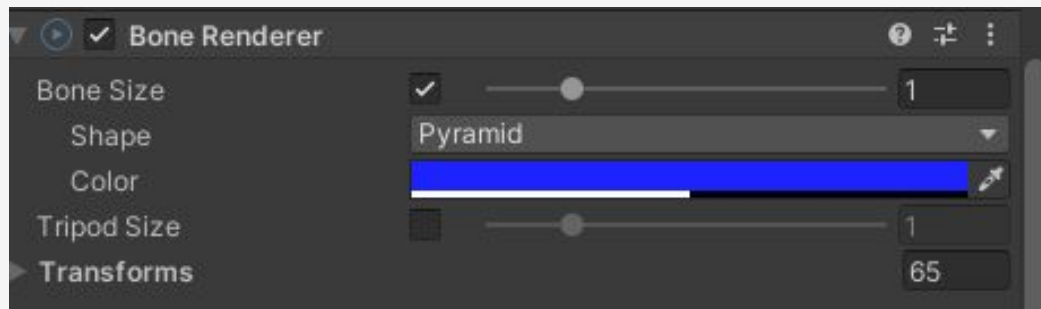
Es un paquete de Unity que ofrece muchas funcionalidades para **desarrollo de animaciones procedurales**

Puedes conseguirlo desde el *Package Manager* de Unity



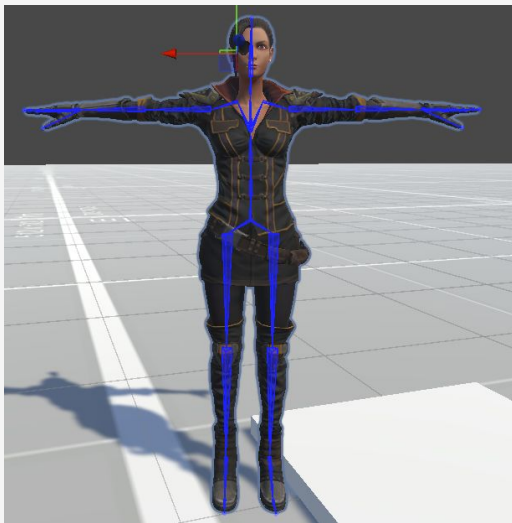
Animation Rigging

El paquete es basto y tiene muchas funciones y componentes, pero por ahora solo usaremos el componente BoneRenderer



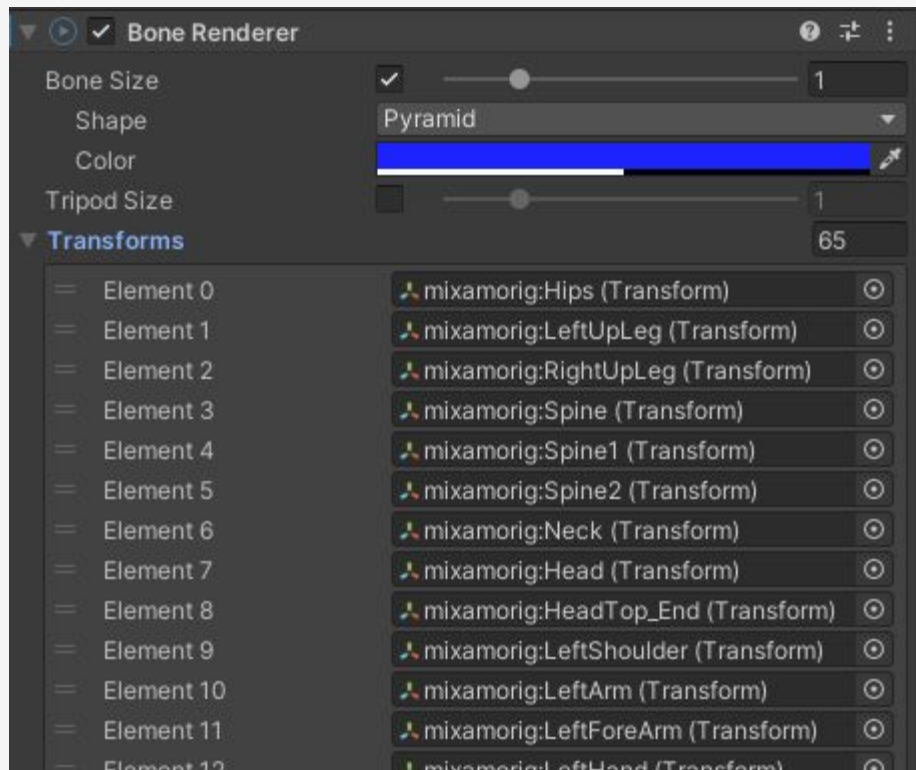
Animation Rigging

Si añadimos un *bone renderer* a un personaje (al mismo nivel que el animator) tendremos acceso a un gizmo que nos permite visualizar el esqueleto como lo hacíamos desde Blender



Animation Rigging

⚠ Si añades el componente pero no puedes ver el Gizmo del esqueleto, revisa que los huesos estén añadidos a la lista de transform en el componente

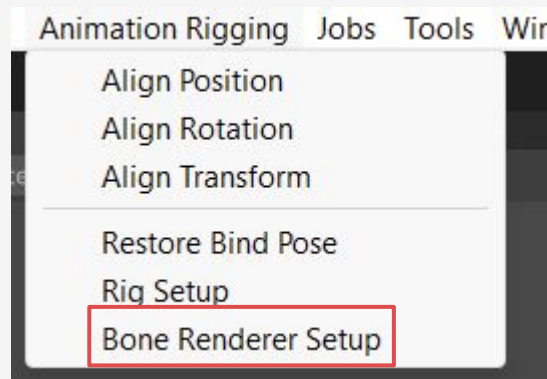
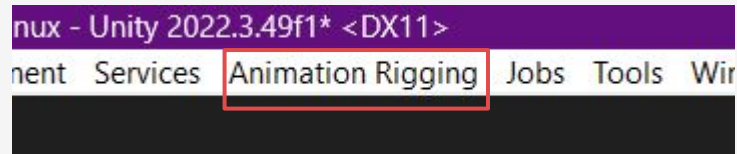
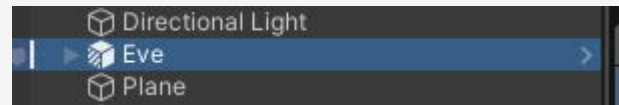


Animation Rigging

⚠ Si no lo estan:

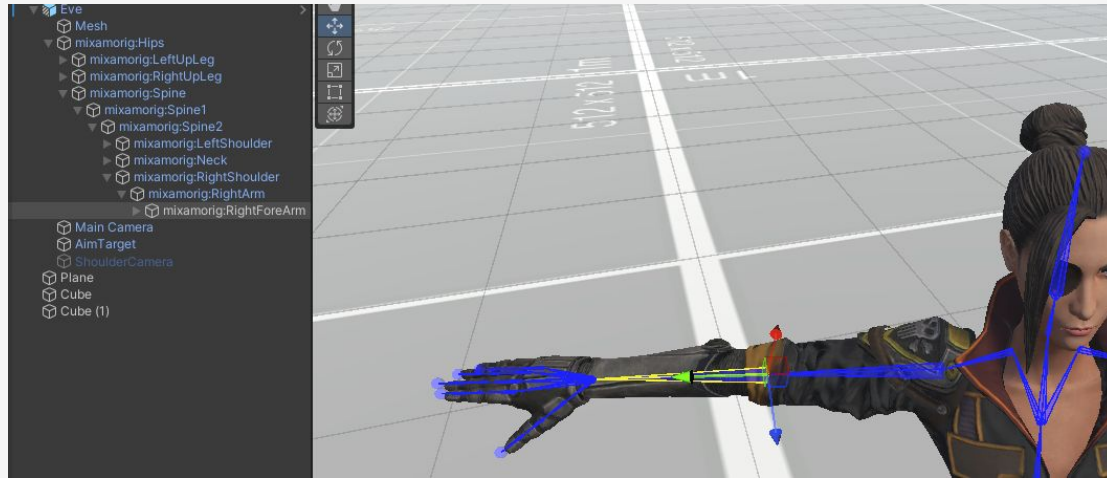
1. Selecciona al personaje en el editor
2. Ve a “Animation Rigging” en la barra de opciones superior
3. Selecciona “Bone Renderer Setup”

También puedes añadirlos manualmente si prefieres



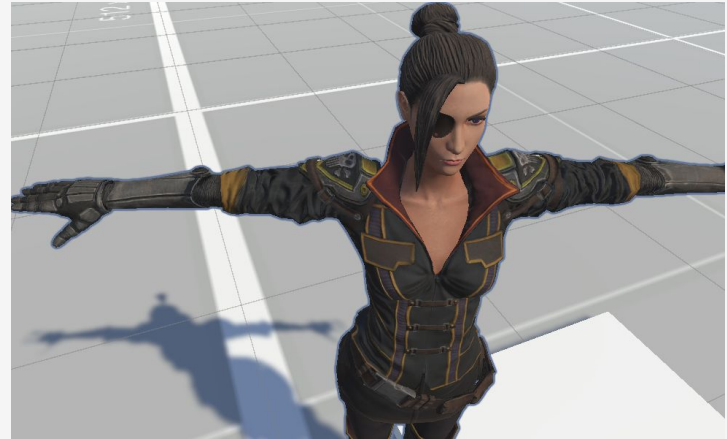
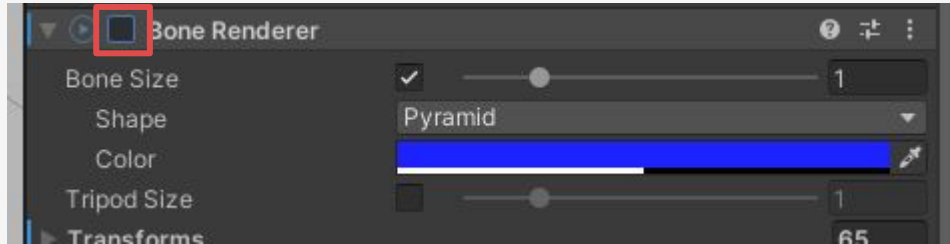
Animation Rigging

Con el *Bone Renderer* de Animation Rigging podrás visualizar y seleccionar los huesos directo en la vista de escena



Animation Rigging

Si terminaste de trabajar con el esqueleto y no quieres verlo más, puedes desactivar el componente para quitar sus gizmos, no afectará en nada al juego



Ejercicios

[illegible]

Ejercicio 0: Importar un personaje

Objetivo: Tener un personaje importado y configurado correctamente en tu proyecto

1. Busca un personaje de tu preferencia en Mixamo
2. Busca animaciones en Mixamo para ese personaje, incluyendo:
 - a. Caminar
 - b. Idle
3. Importa el personaje a Unity siguiendo el workflow de importación para personajes humanoides
4. Crea un controlador de tanque para este personaje que incluya las animaciones de caminar e Idle
5. Asegurate de que puedas ver los Gizmos de IK Goals y IK Hints y que las animaciones funcionan correctamente

Ejercicio 0: Importar un personaje

Resultado



Ejercicio 1: Mirada amenazante

Objetivo: El monstruo enemigo debe seguir al jugador con la mirada

1. Crea un script para el monstruo en la escena para que siga con la mirada al personaje que puedes controlar
2. El monstruo solo puede seguir con la mirada al personaje en una distancia definida desde el editor
 - a. Alternativamente, puedes usar un collider configurado como trigger
3. Las transiciones entre seguir y dejar de seguir con la mirada deben ser suaves
 - a. Consejo: ¡Usa DoTween!

Ejercicio 1: Mirada amenazante

Resultado



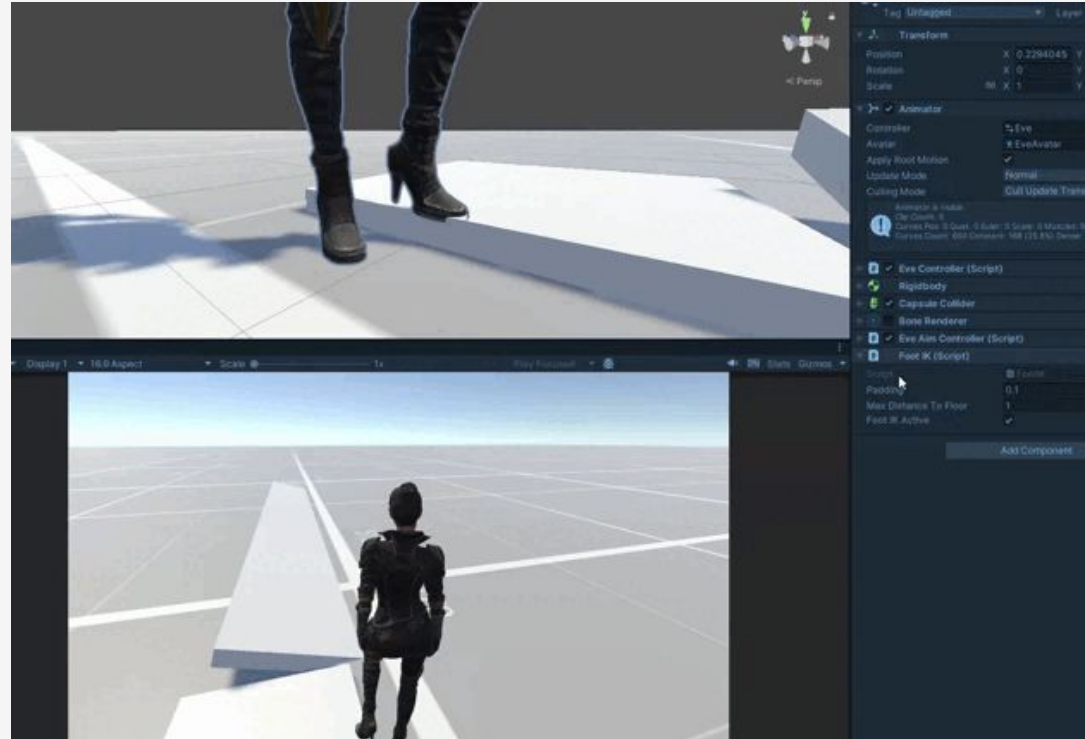
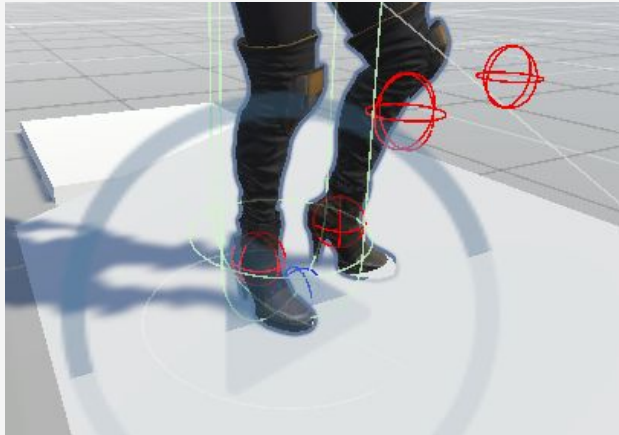
Ejercicio 2: Con los pies sobre la tierra

Objetivo: Nuestro personaje debe tener los pies ubicados en la tierra sin atravesar superficies irregulares

1. Añade un componente al personaje en escena que ajuste la altura de los IK Goal de los pies para que se mantengan en contacto con el suelo
2. El componente del personaje debe exponer una variable booleana en el editor que permita activar y desactivar el ajuste de pies
3. **Sugerencia:** Añade una variable float configurable desde el editor que sirva de tolerancia para este ajuste
 - a. ¿Tolerancia de qué? 🙄

Ejercicio 2: Con los pies sobre la tierra

Resultado



Ejercicio 3: Apuntar como Leon

Objetivo: El personaje debe poder apuntar con el click derecho, moviendo el arma con el mouse, como en Resident Evil

1. El personaje en escena puede apuntar con el arma usando click derecho, lo que le hace entrar en modo apuntado
 - a. No tienes que implementarlo!
2. Completa el componente “EveAimController” para que incluya la siguiente funcionalidad:
 - a. Cuando el jugador mueve el mouse, el personaje mueve la mira del arma en la misma dirección
 - b. Los brazos, el cuerpo y la cabeza deben seguir el movimiento de forma natural
3. El área de apuntado debe ser un cuadrado (o rectángulo) con tamaño de lado personalizable desde el inspector
 - a. Bonus: Añade un gizmo para visualizar el cuadrado
4. **Sugerencia:** Añade un objeto invisible que se use para guiar el movimiento del cuerpo

Ejercicio 3: Apuntar como Leon

Resultado



Ejercicio 4: Knockback dinámico

Objetivo: Cuando se le hace click (dispara) al monstruo en los brazos o la cabeza, mueve estas partes del cuerpo hacia atrás en la dirección del disparo

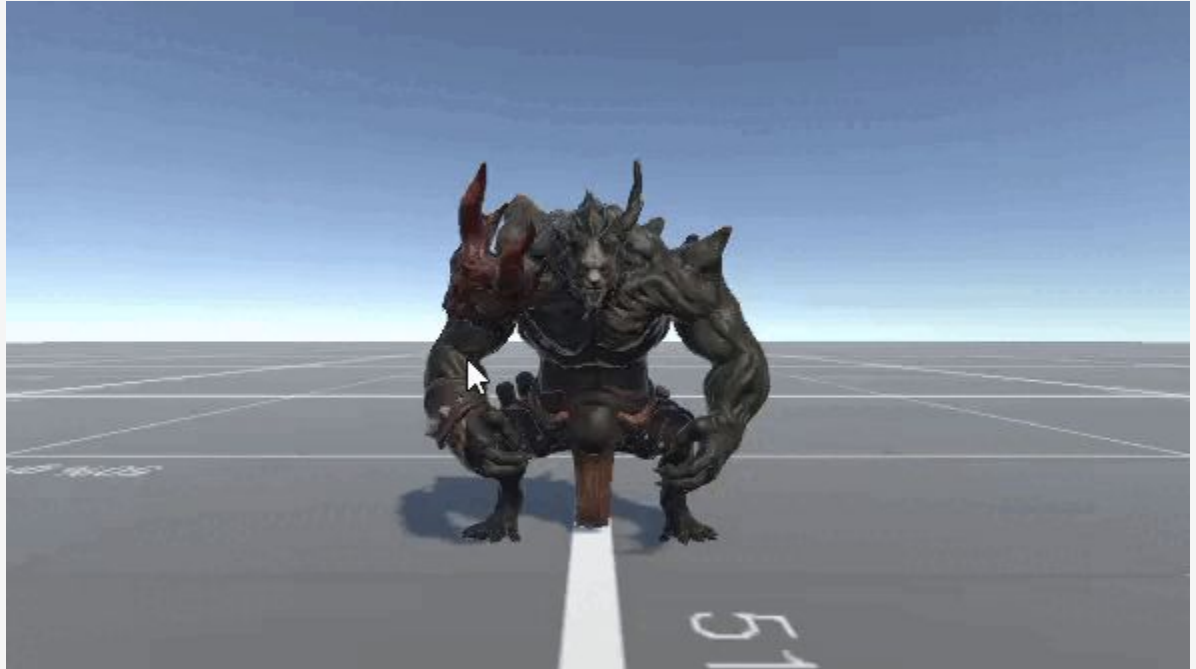
1. Añade un componente al monstruo para manejar el knockback
2. En ese componente, crea una función pública “Hit” que reciba la dirección del “disparo” en el punto de contacto (y cualquier otra cosa que consideres útil)
 - a. Esta función debe iniciar el knockback
 - b. El componente debe encargarse de determinar qué parte del cuerpo fue impactada
3. Crea un objeto vacío en escena y añade un componente que permita “disparar” haciendo click sobre el monstruo
 - a. Al hacer click sobre el monstruo, el objeto hace un *ray cast* en la dirección del click. Si alcanza al monstruo, debe llamar a la función “Hit” que mencionamos antes
4. Cuando se le dispara en **los brazos o la cabeza**, el monstruo debe mover la parte del cuerpo en la dirección del disparo por un momento
 - a. ¿Cómo distingues donde recibió el disparo? 🙄
 - b. ¿Podemos implementar el movimiento de la cabeza igual al de los brazos?
5. El knockback debe estar suavemente animado, las partes del cuerpo no pueden dar saltos

Ejercicio 4: Knockback dinámico

Resultado

Pista: El proyecto tiene una capa especial llamada “knockback”.

¿Por qué?



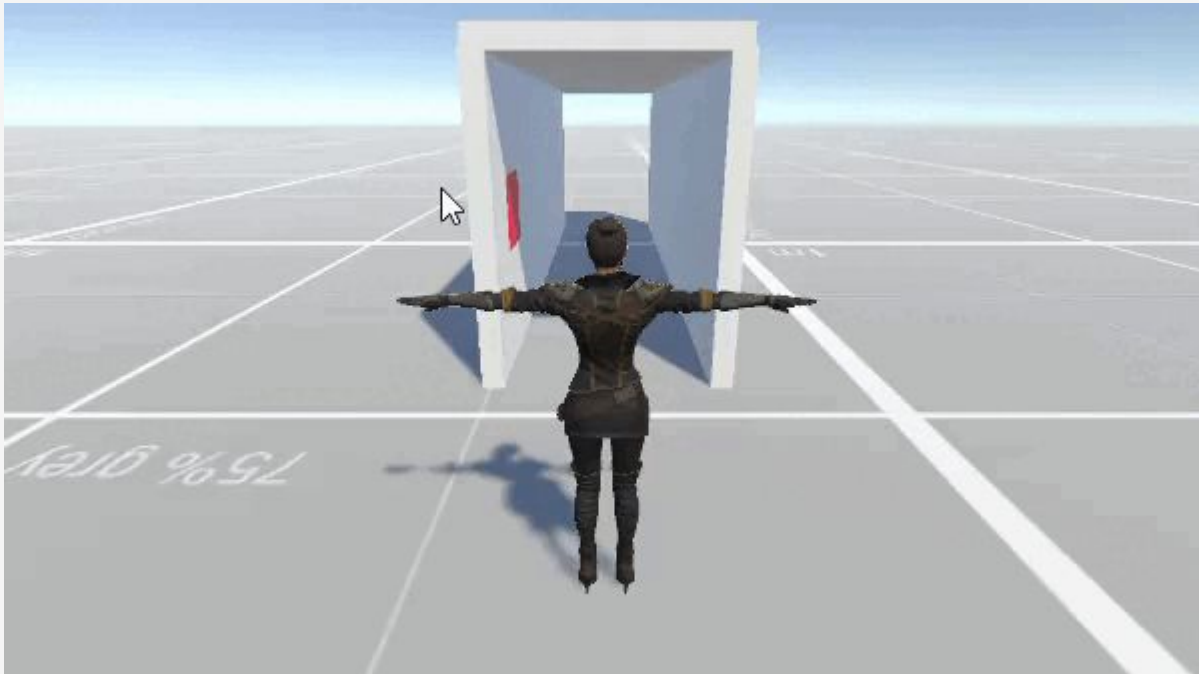
Ejercicio 5: Immersive Walking

Objetivo: El personaje debe poder “curiosear”: mirar y tocar elementos destacables del ambiente cuando pasa cerca de ellos

1. Añade un componente al objeto rojo en la pared que gestione la entrada en rango del personaje
 - a. El objeto tiene un collider como trigger, úsalo para determinar entrada y salida del personaje del área de efecto
2. Añade un campo de tipo **GameObject** al componente que sirva para indicar el punto donde el personaje ubicará su mano
3. Añade un componente al personaje para implementar el gesto de curiosear
4. El personaje debe mirar al objeto interesante cuando entra en rango
5. También debe estirar la mano para intentar tocarlo
 - a. Con la mano que tenga más cerca (si el objeto está a la izquierda del personaje, usa la mano izquierda. Si no, usa la derecha)
6. Debe tener animaciones suaves para entrar y salir del estado de curiosear

Ejercicio 5: Immersive Walking

Resultado



Ejercicio 6: Nivel Final

Objetivo: Usa lo que hiciste en los ejercicios anteriores para montar una escena

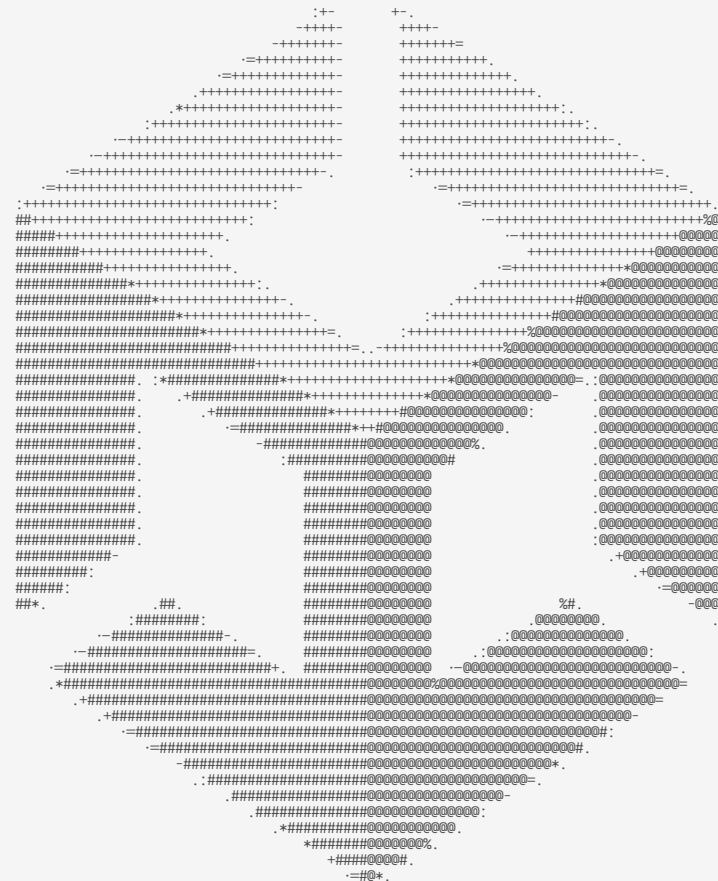
1. Añade los componentes que creaste al personaje jugable de la escena
 - a. componente para ajuste de pies
 - b. componente para curiosear
2. Lo mismo con el monstruo
 - a. componente de knockback
 - b. componente de mirada amenazante
3. Y con el panel rojo de la escena
 - a. Componente de cosa interesante
4. Completa la función “Shoot” el componente “EveShooterController” para que use la función “Hit” en el monstruo para dispararle

Ejercicio 6: Nivel Final

Resultado



Referencias y assets



Referencias

- Inverse kinematics, Unity
<https://docs.unity3d.com/Manual/InverseKinematics.html>
- Root motion, Unity
<https://docs.unity3d.com/Manual/RootMotion.html>
- Humanoid Animations, Unity
<https://www.youtube.com/watch?v=s5lRq6-BVcw>

Assets

- Sci Fi Gun, Unity Asset Store:
<https://assetstore.unity.com/packages/3d/props/guns/sci-fi-gun-162872>
- Eve By J. Gonzalez, Mixamo:
<https://www.mixamo.com/#/?page=1&query=eve&type=Character>
- Warrok (monster), Mixamo:
<https://www.mixamo.com/#/?page=1&query=warrok&type=Character>
- Animaciones de [Mixamo](#)
- DOTween:
<https://assetstore.unity.com/packages/tools/animation/dotween-hotween-v2-27676>
- Sci Fi Warehouse, Unity Asset Store:
<https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-construction-kit-modular-159280>
- Blaster Kit, Kenney:
<https://kenney.nl/assets/blaster-kit>
- Quick Effects, Unity Asset Store:
<https://assetstore.unity.com/packages/vfx/particles/free-quick-effects-volume-1-304424>