Máster en Desarrollo de Videojuegos

(Especialidad de Programación)

Motores: Unity

Construyendo un FPS. II Parte.

[10 - 10 - 2019]

Ismael Sagredo: isagredo@ucm.es

Universidad Complutense de Madrid

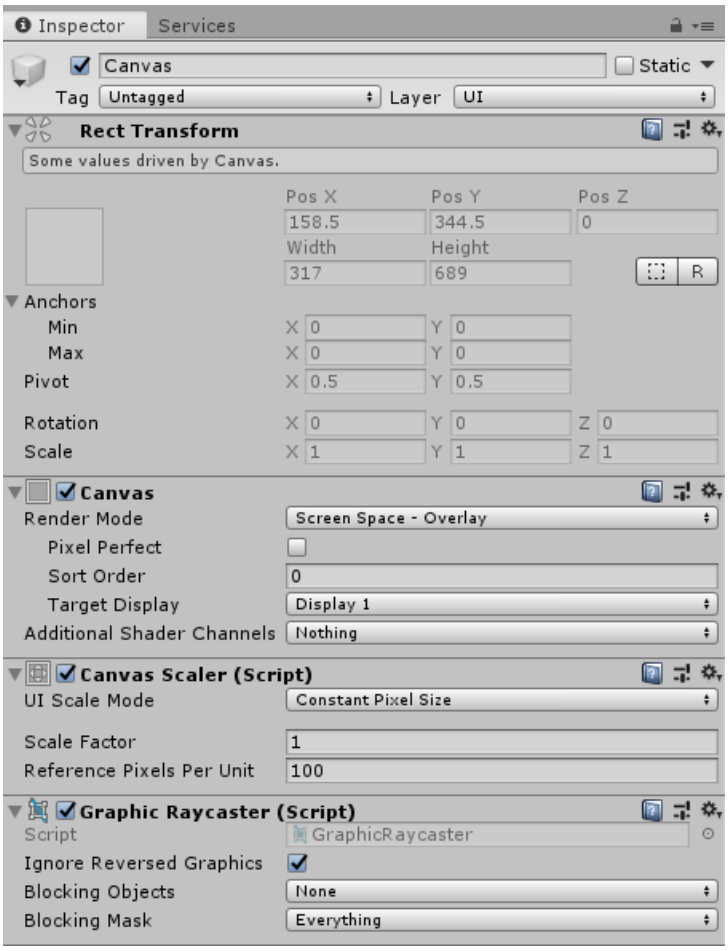
Contenido

Punto de mira y GUI de juego

Vamos a dibujar una mirilla y una barra de vida. La vida va a disminuir si nos caemos desde una gran altura o si pasamos por zonas peligrosas. Para ello vamos a crear el componente Health que gestiona la salud de una entidad. Para ello completamos el TO DO 1 del componente Health.

```
        m_CurrentHealth -= amount;
        if (m_CurrentHealth <= 0)
        {
            this.gameObject.SendMessage("OnDeath",
SendMessageOptions.DontRequireReceiver);
        }
```

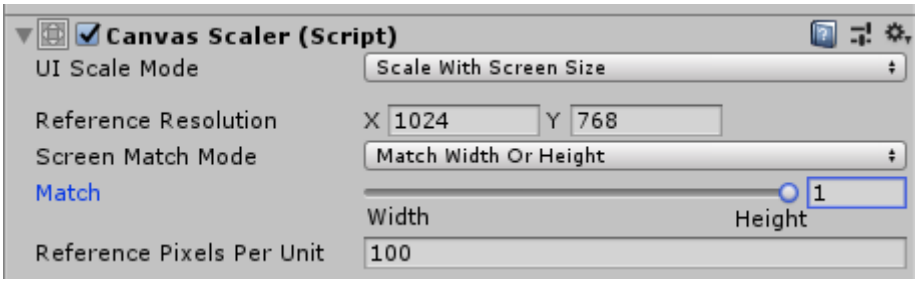
Una vez que tenemos el comportamiento para la vida, vamos a introducir la información en la UI. Para ello creamos un **Canvas en GameObject >> UI >> Canvas**.



El Canvas es el área donde se va a crear la GUI del juego. Todos los objetos del GUI deben ser hijos del Canvas. El orden de pintado de los objetos depende del orden de la jerarquía. El orden se puede cambiar por código usando Transform: SetAsFirstSibling, SetAsLastSibling, and SetSiblingIndex. Ya incorpora una serie de componentes como:

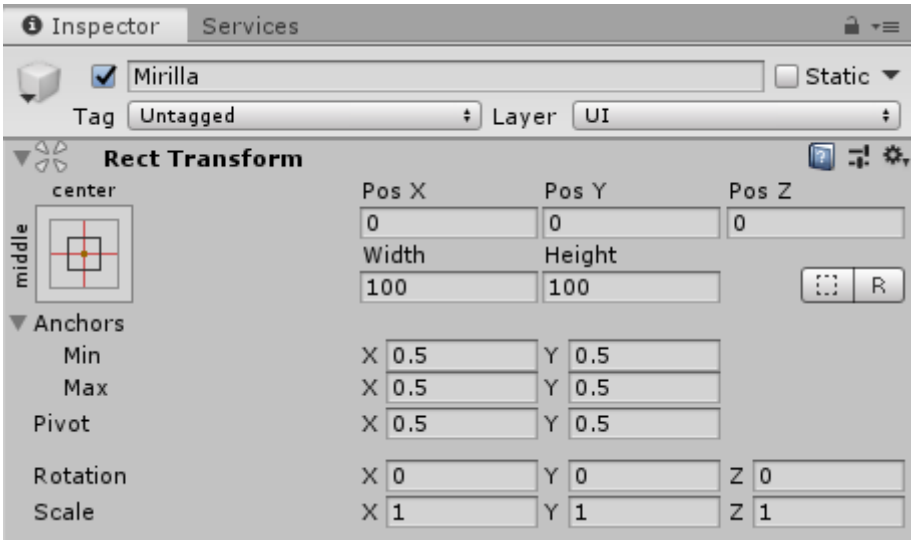
- **Rect Transform**: Transformación del rectángulo que determina la posición y el tamaño del objeto. Se puede modificar:
  - Anchor: Ancla del canvas que sirve como referencia de la posición del objeto y el escalado.
  - Pivot: Posición con respecto al padre.
- **Canvas**:
  - Render Mode: Hay varias formas de renderizar
    - Screen Space overlay: se coloca en la pantalla en el top 2D.
    - Screen Space camera: Se coloca en la pantalla por encima de todo, pero respeta los settings de la cámara, como por ejemplo la profundidad.
    - World Space: El canvas es un objeto más de la escena. El tamaño puede ser establecido usando el Rect Transform y no se calcula automáticamente.
  - Pixel perfect: evita que un pixel de la imagen caiga entre dos pixeles de la pantalla. Par ello, recoloca los gráficos dependiendo de la resolución de la pantalla y el tamaño de la imagen.
  - Sort Order: Puede haber más de un canvas, este campo los ordena para saber cuál es el orden de pintado y el orden de captura de eventos.
- **Canvas Scaler**: Controla como queremos escalar la imagen del canvas para diferentes dispositivos.
  - Content Pixel Size: Conserva el mismo tamaño en pixeles sin importar el tamaño de la pantalla.
  - Scale with Screen Size: Hace los elementos más grandes si la pantalla es más grande.
  - Constant Physical Size: Conserva el mismo tamaño físico sin importar el tamaño de la pantalla y la resolución.
- **GraphicRaycaster**: Control del Raycaster del input para saber a qué objeto se está clicando. Inicialmente tiene en cuenta el Rect del objeto y no las transparencias. Para eso hay que implementar un componente que implemente la interfaz **ICanvasRaycasterFilter**.

Para añadir la mirilla, tenemos que asegurarnos que esta está siempre en el centro de la pantalla. Vamos a configurar el escalado seleccionando **Scale With Screeen Size**, asumiendo que la pantalla es 4:3. Para ellos vamos a utilizar la definición HD estándar 4:3 que es 1024x768 y el match en height (1) como aparece en la imagen:



Como idea general (dependerá en cada caso concreto) es preferible diseñar la GUI del juego a 4:3 que a 16:9 y ajustar en alto. Esto es porque convertir 4:3 en 16:9 implica, si la GUI está centrada, dejar una parte de GUI de los extremos sin rellenar. Al contrario, o se deformará o no cabrá si lo ajustamos en ancho. En general, suele ser mejor que la GUI falte por los lados que por arriba y por abajo ya que tapará más parte del juego.

Creamos una imagen desde **GameObject >> UI >> image**, la renombramos a "Mirilla" y la colocamos en el centro del Canvas con el pivot en el 0.5 y los anchores con 0.5. Para seleccionar cómo va a escalar y dónde se va a anclar al padre, hay que presionar las teclas **shift** y **alt**. Cada una de ellas permite modificar ambas cosas.

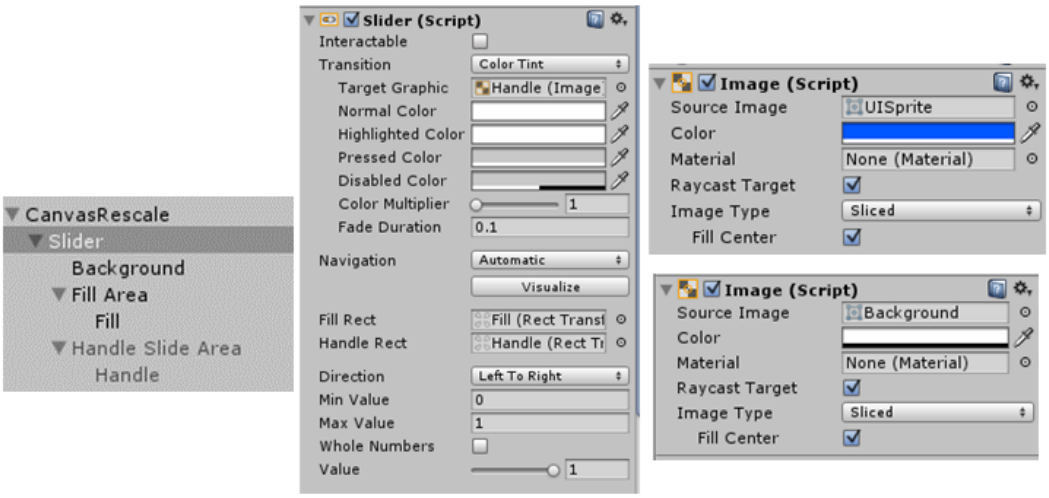


Si probamos el juego, vemos, la mirilla varía dependiendo de la resolución. Esto en este caso no lo queremos así que vamos a configurar el canvas scaler con la opción **Constant Pixel Size.**

Ahora bien, queremos que el resto del canvas sí que reescale, por lo que tenemos que crear otro canvas con las propiedades que habíamos establecido inicialmente al anterior. Vamos a renombrar el Canvas a **CanvasConstantPixel** y el nuevo a **CanvasRescale** y establecemos el short order a -1 en el nuevo canvas.

Ahora en el CanvasRescale vamos a meter la barra de vida. Para ello vamos a utilizar el control **Slider** de la UI de Unity, pulsando el botón derecho sobre **CanvasRescale >> UI >> slider**. Cambiamos el Anchor y el pivot para que estén centrados arriba de la pantalla.

Ahora vamos a configurar la slider. Nos vamos al ***background*** y le ponemos la imagen como transparente. Después vamos a **Fill Area >> fill** y le ponemos el color Azul. En slider desactivamos la opción interactable y desactivamos el gameobject **Handle Slide Area** para que no se vea el manejador.



Ahora tenemos que cambiar el Slider interactivamente con la vida del player. Para eso abrimos el componente **HealthSlider.cs.**

Completamos los TODOs de HealthSlider para conseguir modificar los valores de la slider y adaptarlos a la vida del personaje.

TO DO 1:

```
        m_slider = GetComponent<UnityEngine.UI.Slider>();
        if (m_slider != null)
        {
            m_slider.minValue = 0f;
            m_slider.maxValue = m_health.m_health;
            m_slider.value = m_health.m_health;
        }
```

TO DO 2

```
m_slider.value = m_health.CurrentHelth;
```

Y ahora montamos la escena colocando Health en el player y dándole de vida inicial 10 y HealthSlider en el Slider y asignamos la variable Health del HealthSlider el GameObject del player.

Una Puerta que se abre

Vamos a implementar una puerta que se abre cuando un objeto está a punto de entrar y se cierra cuando no hay objetos debajo de ella. Para ello vamos a abrir la clase Door y ver cómo está implementada:

- La puerta puede estar en 4 estados. Abierta, cerrada, abriéndose y cerrándose.
- Para detectar la presencia de un objeto vamos a usar el conceto de **trigger**.
    - Un **trigger** consiste en asociar a un volumen (collider en Unity) una serie de acciones. Normalmente en un trigger podemos detectar tres eventos. Cuando alguien entra en el trigger (OnTriggerEnter), cuando alguien que estaba en el trigger sale (OnTriggerExit) y cuando alguien está en el trigger (OnTriggerStay).
- Llevaremos la cuenta de los objetos que estan en el trigger para asegurarnos de que no cerramos una puerta si hay alguien dentro del trigger (`m_numElementsInTrigger`)
- Asumimos por simplicidad debido a que usamos animaciones para mover la puerta que estas no se cancelan hasta que no terminan. Por ese motivo debemos conocer la duración de la animación para cambiar de estado internamente.

TO-DO 1: Abrimos la puerta.

```
if (m_state == State.CLOSED && m_numElementsInTrigger > 0)
{
    m_state = State.OPENING;
    m_animation.Play(m_open);
    m_audio.Play();
    m_remainingTime = animationDuration;
}
```

TO-DO 2: Cerramos la puerta

```
if (m_state == State.OPEN && m_numElementsInTrigger <= 0)
{
    m_state = State.CLOSING;
    m_animation.Play(m_close);
    m_audio.Play();
    m_remainingTime = animationDuration;
}
```

Si ejecutamos vemos que puede haber errores si disparamos a la puerta sucesivamente, la puerta se queda abierta. Esto es debido a que puede ocurrir que haya objetos que entren en el trigger y que no salgan nunca (los destruimos) Para evitar el problema

TO-DO 3: Mantenemos un tiempo de espera adicional para cerrar la puerta independientemente de la lógica de los triggers. Cuando hay algo en el trigger, actualizamos el temporizador de emergencia de cierre.

```
    private void OnTriggerStay(Collider other)
    {
        triggerErrorTime = animationDuration;
    }
```

TO-DO 4: Y cuando ese tiempo expira, cerramos la puerta y reestablecemos a 0 el contador de elementos en el trigger.

```
            if (m_state == State.OPEN && triggerErrorTime <= 0)
            {
                m_numElementsInTrigger = 0;
                Close();
            }
```

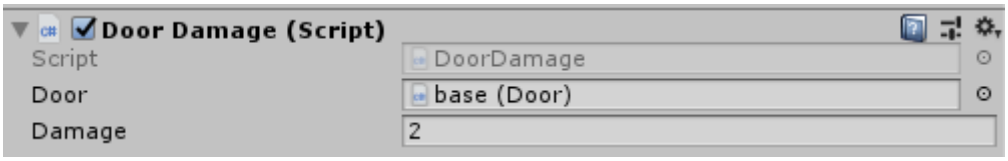
Ahora vamos a hacer daño al jugador si la puerta le golpea. Par ello vamos a usar el tag SF_**door** con el que esta etiquetada la puerta movil. Si existe una colisión con la puerta movil que golpea por encima al player, este recibe daño. Para ello:

- Creamos un gameobject debajo de door que denominaremos DoorDamageTrigger
- Le añadiremos un box collider, marcándolo como trigger y configurándolo para que quede justo debajo de la puerta sin que sobresalga por los lados para evitar colisiones con el player no deseadas. Aún así para evitar problemas, enlazaremos el script Door para comprobar el estado de la puerta y no hacer daño al player si la puerta no se está cerrando.
- Etiquetar FPSPrefab con el tag Player
- Creamos DoorDamage y lo asignamos en el DoorDamageTrigger
- Escribimos el siguiente código:

```
public class DoorDamage : MonoBehaviour
{
    public Door m_door;
    public float m_damage;

    private GameObject m_player;
    // Start is called before the first frame update

    private void Start()
    {
        m_player = GameObject.FindGameObjectWithTag("Player");
    }

    private void OnTriggerEnter(Collider other)
    {
```


```
        if(other.gameObject == m_player && m_door.DoorState ==
Door.State.CLOSING)
        {
            m_player.SendMessage("Damage", m_damage);
        }
    }
}
```

- Arrastrar el script Death al player y configurar los valores públicos de DoorDamage



Ejercicios

En esta segunda parte de la clase vamos a implementar los siguientes ejercicios. Estos se entregarán al final de la clase. Los ejercicios los podéis subir a vuestro drive y enviarme un enlace a él a isagredo@ucm.es

La teoría necesaria para resolver cada ejercicio se explica en cada uno de ellos.

Ejercicio 1: Metralleta

Colocar una segunda arma con disparo automático. Ya dejamos preparada la clase Shoot para gestionar esto. Debemos extenderla para que funcione usando un disparo automático. El disparo automático consiste en lanzar un rayo en la dirección de apuntado y comprobar donde colisiona para colocar un efecto de partículas de impacto y un sonido en loop de disparo.

Para lanzar un rallo podemos usar el método Raycast de Unity

https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

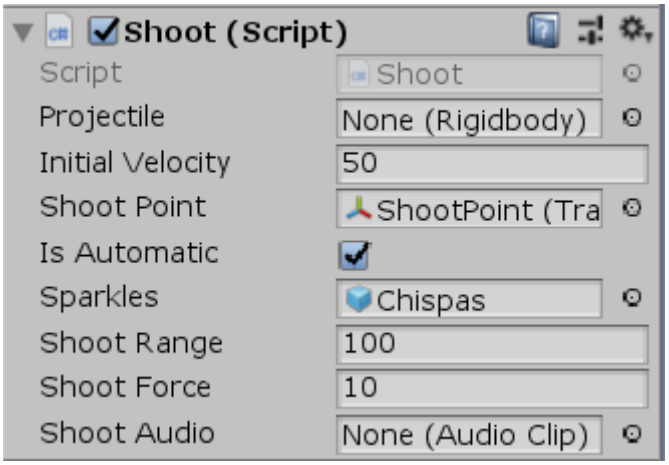
Que nos permite consultar si un rayo interseca un collider. Si es así, nos devuelve información del collider y de donde impacto en la estructura RaycastHit

https://docs.unity3d.com/ScriptReference/RaycastHit.html

Un rayo no es más que una abstracción de un segmento de cierta longitud.

Pasos:

- Completar los TO-Dos del 4 al 9 de shoot
- Desactivar el arma anterior
- Instanciar el prefab Machine como hijo de la cámara.
- Añadirle el audio source, activar **loop** y desactivar **playOnAwake**.
- Añadirle shoot a Machine y configurarlo con las partículas, que sea automático.



- Añadir el Audio Clip de disparo loopable_shoot al audio source
- Si se quiere modificar la cadencia de tiro del arma, hacer que la variable **m_TimeBetweenShots** se serialice.
- No olvidar quitarlo de la Main Camera.

Recursos necesarios:

- loopable_shoot (Audio)
- Machine (Malla y prefab configurado)
- FireMobile (Particulas) Hay que configurarlas, desactivando de los tres ParticleSystems el atributo Looping y estableciendo la duración en 0.5. Con el creáis un prefab nuevo que usareis para instanciar en el TO-DO 9

Ejercicio 2: Cambiar de arma.

Sólo una de las dos armas debe estar activa y visible en cada momento. Para cambiar de arma usaremos los números 1 y 2.

Para leer del input de teclado usaremos el método **Input.GetKeyDown**

Las teclas 1 y 2 son **KeyCode.Alpha1** y **KeyCode.Alpha2**

Pasos:

- Completar los TO-DOs de WeaponManager del 1 al 4

- Agrupar ambas armas bajo un gameobject común que podemos denominar Weapons
- Arrastrar a este gameobject el WeaponManager
- Configurar WeaponManager.

Ejercicio 3: Reespawn del enemigo al morir.

Al morir el player (Mensaje Death) deberá reaparecer en el punto inicial con su salud restaurada. Para ello podemos hacer uso del GameManager.

NOTA: Cuando teletransportamos un CharacterController es recomendable desactivarlo antes, debido a que este guarda una posición interna diferente. Al volverlo a activar, reinicia dicha posición.

Pasos**:**

- Creamos un objeto GameManager
- Le atachamos el script GameManager
- Completamos los TO-Dos 1 y 2.
- Arrastrar Death si no lo tenias ya arrastrado al Player.
- Completar los TODOS 1 y 2 de Death haciendo uso del GameManager
- Probad a morir con la puerta (subirle el daño :P)

Ejercicios opcionales

Cosas opcionales que podéis intentar por vuestra cuenta.

- Retroceso de las armas al disparar
- Añadir más armas
- Zonas de muerte adicionales, plano de muerte cuando caes del escenario
- Crear un escenario diferente con el Pro Builder.