Máster en Desarrollo de Videojuegos

(Especialidad de Programación)

Motores: Unity

Construyendo un Thirds Person Platformer. 2 Parte.

[16 - 10 - 2019]

Ismael Sagredo: isagredo@ucm.es

Universidad Complutense de Madrid

Contenido

Preparativos

Al igual que en prácticas anteriores, se presupone que el punto de inicio de esta práctica es el punto de fin de la anterior.

Si por cualquier razón no tienes el proyecto preparado, puedes bajar una versión actualizada desde el Wiki.

En esta segunda parte de la práctica de nuestro plataformas 3D vamos a realizar un nuevo nivel, el representado por el color Azul.

Veremos brevemente en qué consiste la animación procedural y añadiremos diferentes tipos de plataformas que sirvan para diseñar niveles más completos (y difíciles) así como power-ups para modificar el comportamiento del personaje.

- Importamos **Plataformas – 2.unitypackage**
- Creamos una nueva escena denominada **blue_world** y la añadimos al **BuildSetting** para que no se nos olvide.
- Nos creamos un prefab del escenario si no lo hemos hecho ya par instanciarlo en la nueva escena. También podemos abrir la escena inicial y guardarla como blue_world y después borrarla. Lo más práctico es prefab.
- Añadimos el prefab **Levelblue** y colocarlo en el escenario en su sitio.
- Una vez colocado, borrar escenario.
- Añadir la llave instanciando su prefab y configurándolo en **yellow**.
- Eliminar MainCamera y la luz.
- Guardar la escena y crear el trigger que carga la escena en inicial como hicimos con el nivel verde.
- Configuramos **TriggerAzul**.
- Configurar el parámetro **Level To Load** del componente **TriggerLoadAdditive** a *Blue*.

Plataformas en movimiento

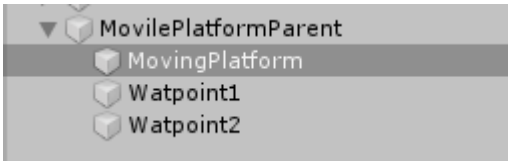
Vamos a configurar una plataforma clásica en la que poder saltar. Esta plataforma luego la reutilizaremos posteriormente en otros escenarios.

- Vamos a la escena blue_world y seleccionamos la plataforma **MovingPlatform**.
- Agregarle el componente **MovingPlatform** y completar los **TODOS**.

```
        void _DoMovement()
        {
          // TODO 1 - Obtener la dirección del desplazamiento y normalizarla
          Vector3 direction = m_CurrentWaypoint.position - transform.position;
          direction.Normalize();
          // TODO 2 - Mover la plataforma en función de la dirección obtenida, la
velocidad, y el deltaTime
        transform.position += m_MovementSpeed * direction * Time.deltaTime;
    }
```

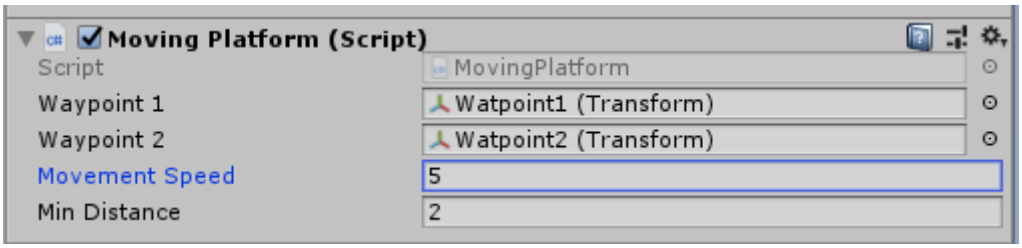
- Creamos dos **GameObjects** vacíos con los nombres **Waypoints1** y **Waypoints2** y le asignamos un **Gizmo** Genérico. Los Gizmos Genéricos.

El **guizmo** nos servirá para poder ver de dónde a donde se moverá la plataforma en diseño. Vamos a construir la plataforma para poderla integrar en cualquier parte. Par ello vamos a agrupar las partes físicas bajo un mismo **GameObject** en el centro del objeto que llamaremos **MovingPlatform**. Los **Waypoints** serán hijos del **MovingPlatformParent** y hermanos de la **MovingPlatform** como se puede ver en la captura.



Hay que desactivar la casilla **Static** de si la tuviese ya que vamos a moder la posición del gameobject. Si fuera estçatico estáticos, Unity no las movería en ningún caso.

Finalmente configuramos el componente **Moving Platformar** como muestra la captura siguiente, arrastrando los dos Waypoints a sus correspondientes campos.



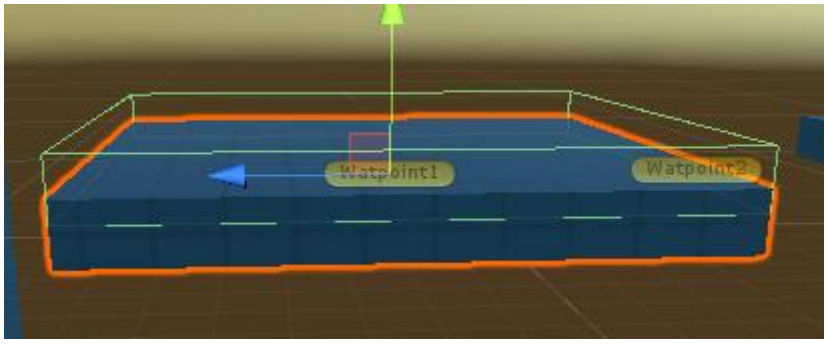
Y Finalmente guardamos **MovinPlatform** como **Prefab**.

Podemos ver que el personaje resbala, no se agarra a la plataforma. ¿Por qué? Pues porque los objetos Kinematicos no alteran la física y porque la plataforma no agarra al objeto ella se mueve independientemente del objeto que está sobre ella.

¿Cómo lo solucionamos?

Añadimos el componente **StickFloor** que nos va a permitir agarrarnos a la plataforma. El truco está en meter al personaje dentro de la estructura de la paltaforma.

- Convertir **MovingPlatform** en **Trigger** añadiéndole un **BoxCollider**. Cambiar el tamaño del BoxCollider para que se ajuste al tamaño de la superficie.



- Añadir **StickFloor** a la plataforma.
- Creamos el Componente **Attachable** con create > c# script borrar todo su contenido y escribimos el siguiente código.

```
public class Attachable : MonoBehaviour {
    [SerializeField]
    private bool _IsAttachable;
    private bool _IsAttached;

    public bool IsAttachable{
        get { return _IsAttachable; }
        set { _IsAttachable = value; }
    }

    public bool IsAttached{
        get { return _IsAttached; }
        set { _IsAttached = value; }
    }
}
```

- Completar los TODOS de StickFloor.
- Asignamos **Attacheble** al **Player** y marcarlo como attachable.

Solución de los TODOS de StickFloor.

```
    void OnTriggerEnter(Collider other)
    {
        //TODO 1: Cuando el objeto que caiga sea attachable, atachamos el objeto.
Ojo, la scala puede cambiar!!!
        Attachable atachable = other.GetComponent<Attachable>();
        if (atachable && atachable.IsAttachable)
        {
            m_EnterScale = other.transform.localScale;
            other.transform.parent = m_transformToAttach;
            atachable.IsAttached = true;
        }

    }

   void OnTriggerExit(Collider other)
    {
```

```
        //TODO 2: Cuando el objeto que caiga sea attachable, como estamos
saliendo, desatachamos el objeto. Ojo, la scala puede cambiar!!!
        Attachable atachable = other.GetComponent<Attachable>();
        if (atachable && atachable.IsAttached)
        {
            other.transform.parent = m_globalParent;
            other.transform.localScale = m_EnterScale;
            atachable.IsAttached = false;
        }
    }
```

Probamos el nivel para ver que podemos agarrarnos a la plataforma.

Rotación de plataformas.

Ahora vamos a hacer una plataforma rotatoria.

- Atachamos al gameobject RotatePlatform el componente RotatingPlatform y resolvemos los TODOS.
- Podemos duplicar la plataforma y poner una como rotación **EndlessRotatingPlatform** y otra como **PeriodicRotatingPlatform**.

```
void Update ()
      {
        // TODO 1 - En función del tipo de rotación, llamar a JustRotate() o a
RotateAndStop()
        // Pista: switch (m_RotateType) {...}
        if (m_RotateType == RotateType.PERIODIC)
        {
            RotateAndStop();
        }
        else
        {
            JustRotate();
        }
    }

      bool CheckIfHasRotated()
      {
        // TODO 2 - Retornar si alguna de las rotaciones de los ejes ha
sobrepasado o es igual al LoopLimit
        return (m_CurrentXRotation >= m_LoopLimit) ||
                (m_CurrentYRotation >= m_LoopLimit) ||
                (m_CurrentZRotation >= m_LoopLimit);

    }
```


Power ups.

Vamos a darle potenciadores a nuestro personaje.

- Crear un cilindro en la escena y llamarlo **SuperJumpPowerUp**. Hacer que sea **Trigger**.
- Completar los TODOs del script **SuperJumpPowerUp.cs**

```
IEnumerator OnTriggerEnter(Collider other)
      {
        // TODO 2 - Si el objeto que entra en mi trigger tiene el tag player

        TrailRenderer trailRenderer = null;
        if (other.tag == "Player")
        {
            // TODO 3 - Le envío un mensaje "SetJumpHeight" con la altura que
tengo configurada para el super-salto
            other.SendMessage("SetJumpHeight", m_SuperJumpHeight);
            // TODO 4 - Desactivo el renderer y el collider de mi gameObject
            // Pista: atributo "enabled"
            this.GetComponent<Renderer>().enabled = false;
            this.GetComponent<Collider>().enabled = false;

            // TODO Refactor 1 - Iniciar el timer del GUIManager (método
StartPowerUpTimer)

            // TODO Refactor 2 - Obtener el componente TrailRenderer del jugador
y activarlo
        }

        yield return new WaitForSeconds(m_duration);

        // TODO 5 - Envío un mensaje recuperando la altura del salto anterior
(por defecto, 6)
        other.SendMessage("RestoreJumpHeight");

        // TODO Refactor 2 - Obtener el componente TrailRenderer del jugador y
desactivarlo

        Destroy(gameObject);
    }
```

- Añadir un componente SuperJumpPowerUp a nuestro potenciador **SuperJumpPowerUp**.
- Tenemos que modificar la fuerza con el mensaje, añadimos a ThirdPcontrollerCharacter el siguiente método.

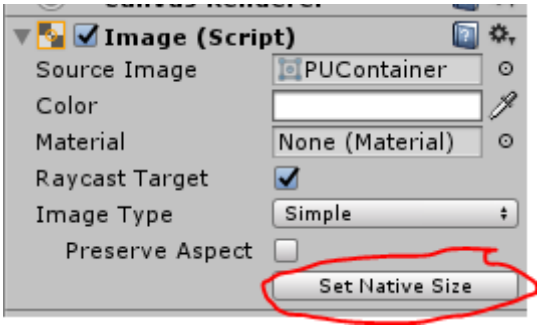
```
        public void SetJumpHeight(float height)
        {
            m_lastPowerUp = m_JumpPower;
            m_JumpPower = height;
        }


public void RestoreJumpHeight() { m_JumpPower = m_lastPowerUp; }
```

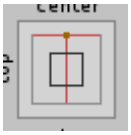
Mostrar un temporizador.

El siguiente paso que vamos a dar en nuestro juego va a ser modificar el GUI que ya tenemos para mostrar un temporizador que indique al jugador cuánto tiempo le queda de Power Up.

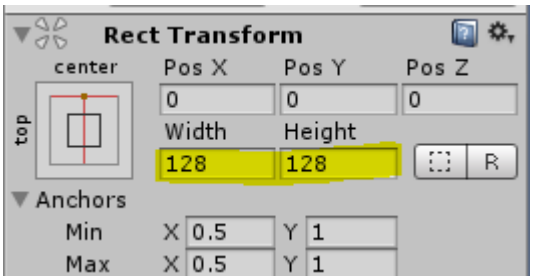
- Dentro del **Canvas** añadimos un **UI > Image**. Cambiamos su nombre a **CoolDownCounter** y asignamos como **Source Image** la textura **Arte/PUContainer**. Por último pulsamos en el botón **Set Native Size**.



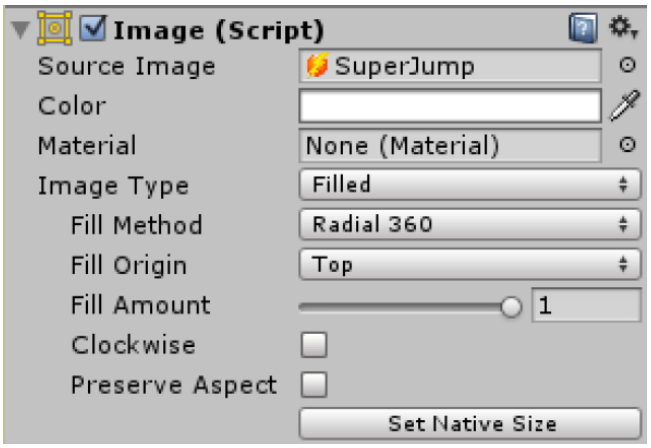
- Colocamos la imagen en la parte superior del Canvas y centrada usando los Anchors. Pulsar Shift + Art para colocar el pivote y la posición en la parte superior centrada.



- Cambiamos su tamaño



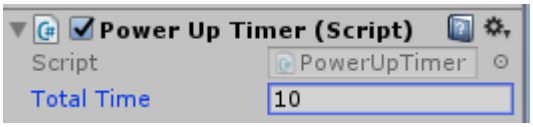
- Añadimos otra imagen como hija de **CoolDownCounter (Create>UI>Image)**. En este caso, escogemos como **Source Image** a **SuperJump** y de nuevo pulsamos en **Set Native Size**
- Cambiamos el **Image Type** de la imagen hija que acabamos de crear a **Filled** y ajustamos sus atributos como se muestra en la captura.



- Añadimos a la nueva imagen el script **PowerUpTimer** y resolvemos sus TODOS

```
    void Update ()
    {
            // TODO 1 - Comprobamos si se ha acabado el tiempo
        if (m_RemainingTime <= 0)
        {
            // TODO 2 - Desactivamos el gameobject para que no se pinte
            gameObject.SetActive(false);
        }
        else
        {
            // TODO 3 - Calculamos cuánto powerup hay que pintar (entre 0 y
1)
            // dependiendo del tiempo que nos queda
            float portion = Mathf.InverseLerp(0, TotalTime, m_RemainingTime);
            // TODO 4 - Asignamos este valor al fillAmount de la imagen
            m_Image.fillAmount = portion;
            // TODO 5 - Restamos al tiempo restante el tiempo que ha pasado
            m_RemainingTime -= Time.deltaTime;
        }
    }
```

- Configuramos los valores como se muestra en la captura



- Probamos a ver si funciona ☺

El resultado será que, al iniciar la partida el temporizador comienza lleno y va vaciándose según pasa el tiempo. Ahora sólo nos falta activarlo en el momento en que el jugador coja el Power Up. Para eso vamos a crear un manager para el GUI.

- Seleccionamos el **Canvas** y le añadimos el script **GUIManager.**
- Rellenamos el campo **Power Up Timer** del componente con la última imagen que añadimos, la que tiene el componente **PowerUpTimer**
- Sólo nos queda llamar al temporizador cuando el jugador coja el Power Up: modificamos el código de **SuperJumpPowerUp.cs** en el **Refactor 1** para invocar al método **StartPowerUpTimer.**

```
            // TODO Refactor 1 - Iniciar el timer del GUIManager (método
StartPowerUpTimer)
            GUIManager.Instance.StartPowerUpTimer(m_duration);
```

Vamos a ver cómo funciona el GUIManager. Nuestra primera aproximación a un Manager en Unity.

```
    void Awake()
```


```
{
        // Si ya existe un GUIManager nos destruimos
        if (Instance != null && Instance != this)
                Destroy(this);
        else
        {
                Instance = this;
        }
}
```

En el Awake comprobamos si la instancia existe si existe la destruimos (el awake sólo se ejecuta una vez y sustituye al constructor)

Una vez esté cargado **Instance** tendrá la variable pública con el GUIManager instanciado.

¿Cómo podemos mejorar esto?

Añadimos en el **Awake** la desactivación de la imagen.

```
Instance.PowerUpTimer.gameObject.SetActive(false);
```

Trail render en el personaje

Como retoque final, añadiremos un feedback gráfico al personaje cuando tenga el ítem de super-salto activo.

Este feedback será una estela que vaya dejando a su paso. En **Unity**, este tipo de efecto gráfico se consigue utilizando **Trail Renderers**.

- Añadir a nuestro Third Person Controller un componente **Effects > Trail Render**.
- Crear un nuevo material llamado MaterialRender de tipo **Particles>PriorityAdditive**.
- Arrastrar el material al componente Trail.
- Implementar **SuperJumpPowerUp** (Refactor 2)
- Desactivar inicialmente el Trail.

```
            // TODO Refactor 2 - Obtener el componente TrailRenderer del jugador
y activarlo
            trailRenderer = other.GetComponent<TrailRenderer>();
            if(trailRenderer!=null)
                trailRenderer.enabled = true;
```