

PMMD07.- Desarrollo de juegos 3D

1.1.- INTRODUCCION.

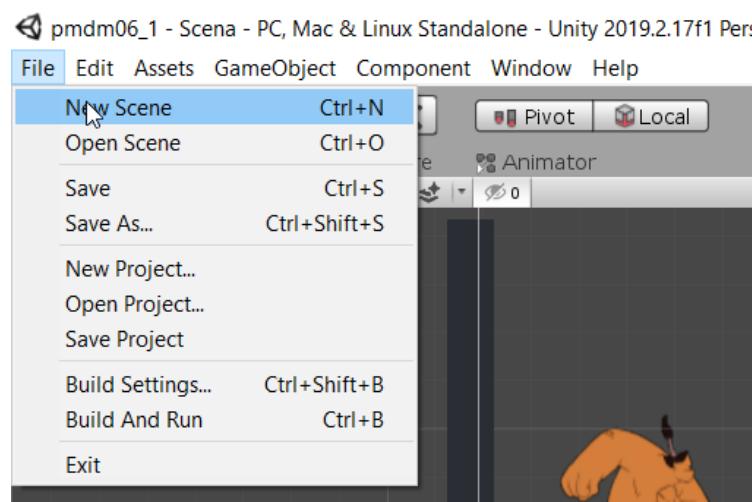
Lo primero que vamos a realizar es comprender la necesidad de incluir ciertos elementos dentro de un juego para conseguir que tenga una apariencia adecuada. Los elementos que vamos a ver son los siguientes:

- Un menú de inicio. El menú por lo menos debe dar la opción de iniciar el juego. Otros aspectos a tener en cuenta pueden ser las opciones de configurar las teclas, el idioma, etc.
- Un interfaz de usuario que incluya información relevante. Aunque no es nuestro caso, si tuvieras un juego diseñado para un dispositivo móvil, en el interfaz habría que incluir los botones necesarios.
- Una pantalla de Game over. Cuando el juego haya finalizado se debe mostrar una pantalla de fin en la que se indique el final del juego y sería conveniente incluir una lista con las puntuaciones más altas.

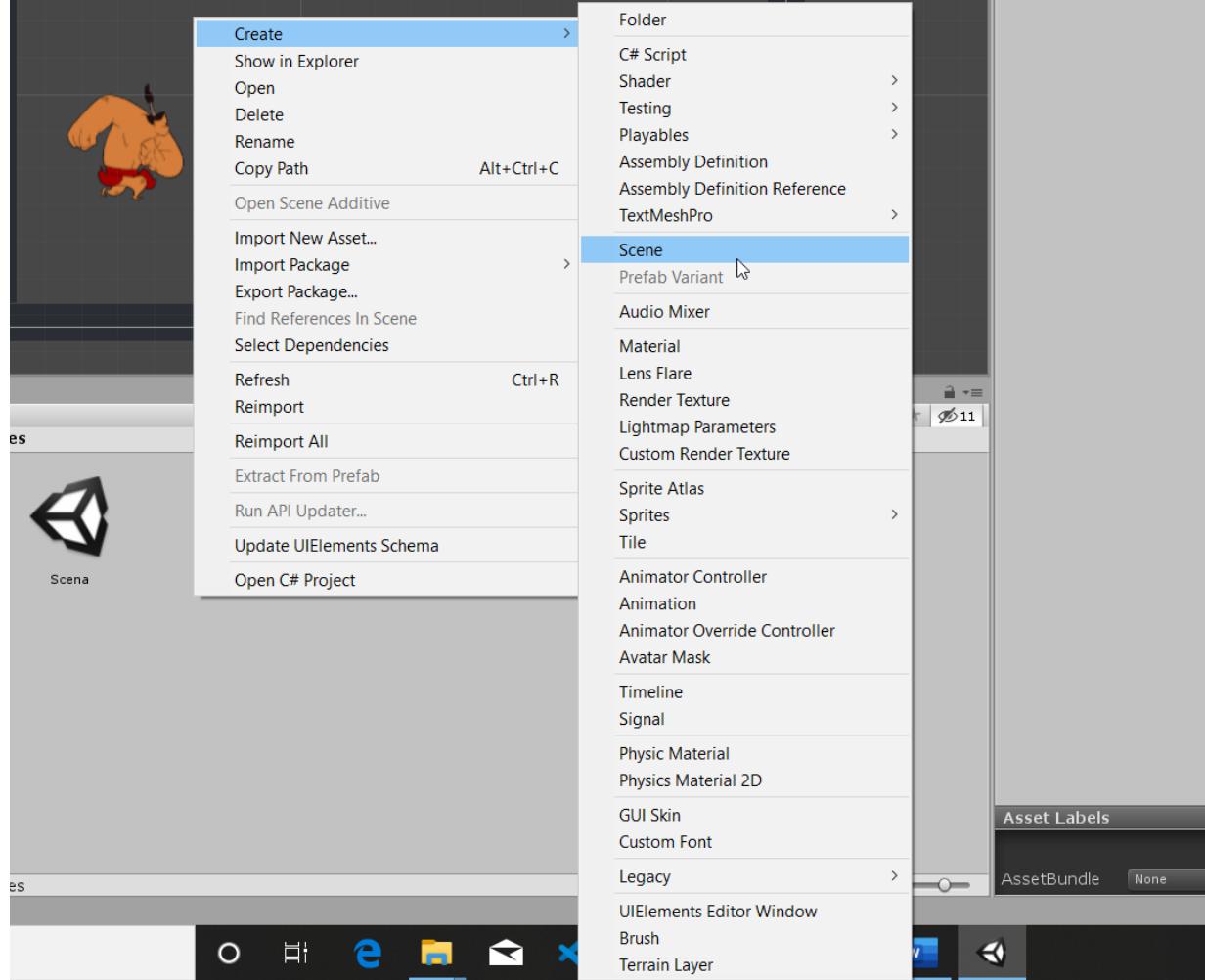
Para poder introducir los elementos mencionados vamos a tener que utilizar diferentes escenas y UIs. Para poder cargar las escenas tendremos que utilizar la funcionalidad que nos ofrece Unity para cargar las escenas que hemos incluido en el proyecto. Veremos que las escenas se numeran desde el cero y que la primera escena en cargarse es la que tiene el índice 0 y el resto se cargan utilizando su índice. Para evitar que la carga de una escena pesada genere dudas sobre si se ha seleccionado la opción de iniciar veremos como incluir un mensaje de carga. Por último, veremos como evitar que un GameObject no se destruya al cargarse una nueva escena. Esto es de ayuda cuando queremos mantener por ejemplo un controlador que se encargue de gestionar el juego.

1.2.- MENÚ DE INICIO.

Para realizar el menú de inicio vamos a necesitar una nueva escena. Si estamos en un proyecto nuevo utilizaremos la escena que nos ofrece Unity al iniciar un proyecto y en caso contrario crearemos una nueva escena. Para crear una nueva escena podemos hacerlo desde File->New Scene, en el proyecto create->scene.

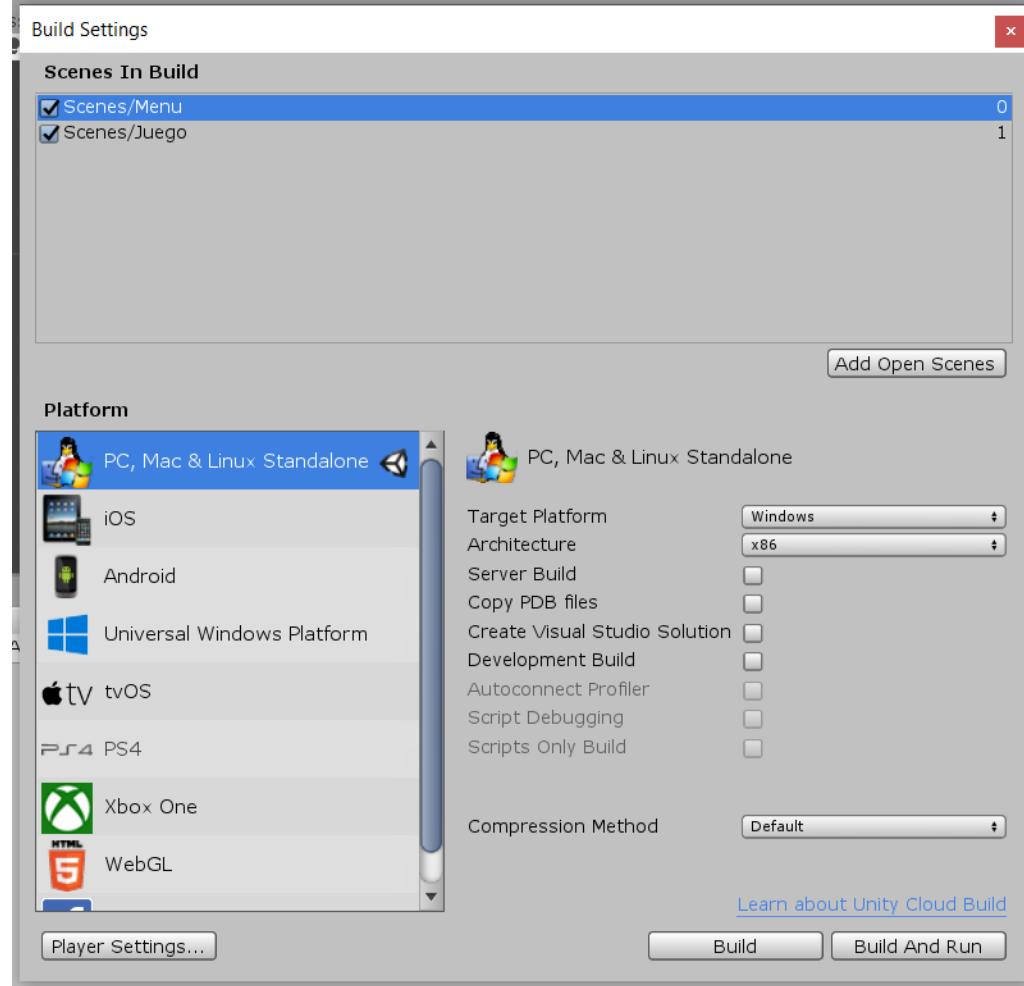


BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Lo que va a ser importante es que las organicemos en una carpeta para evitar confusiones. Además, para poder cargar las escenas será necesario que hayamos añadido las escenas al proyecto. Para ello, iremos a File->Build Setting y añadiremos todas las escenas que vayamos a utilizar en nuestro juego. En nuestro caso tendremos dos: el menú y el juego.



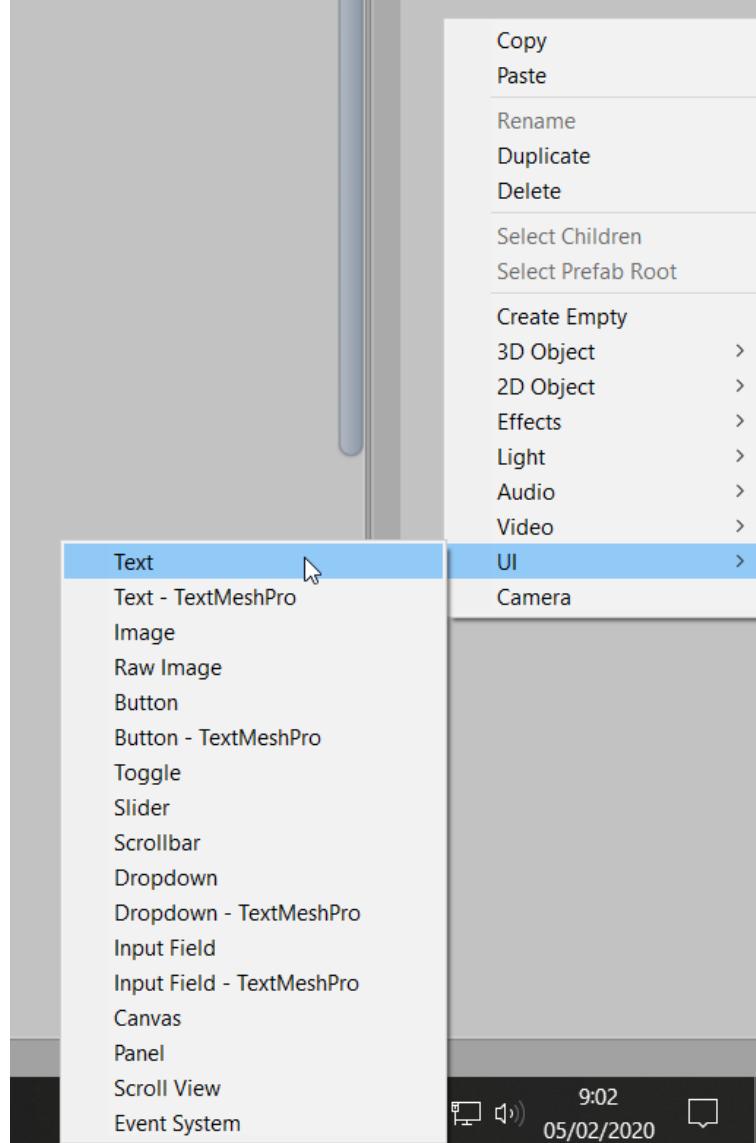
BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)) Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Los números que aparecen junto con el nombre de las escenas son los índices de las escenas. La primera que se va a cargar es la escena cero. Para cargar el resto de las escenas utilizaremos su índice.

1.2.1.- UI.

Una vez hemos tenemos las dos escenas, ya podemos comenzar a realizar el menú de inicio. En nuestro caso, la escena que vamos a utilizar como juego es la escena de la unidad anterior, aunque se puede utilizar cualquier otra escena.

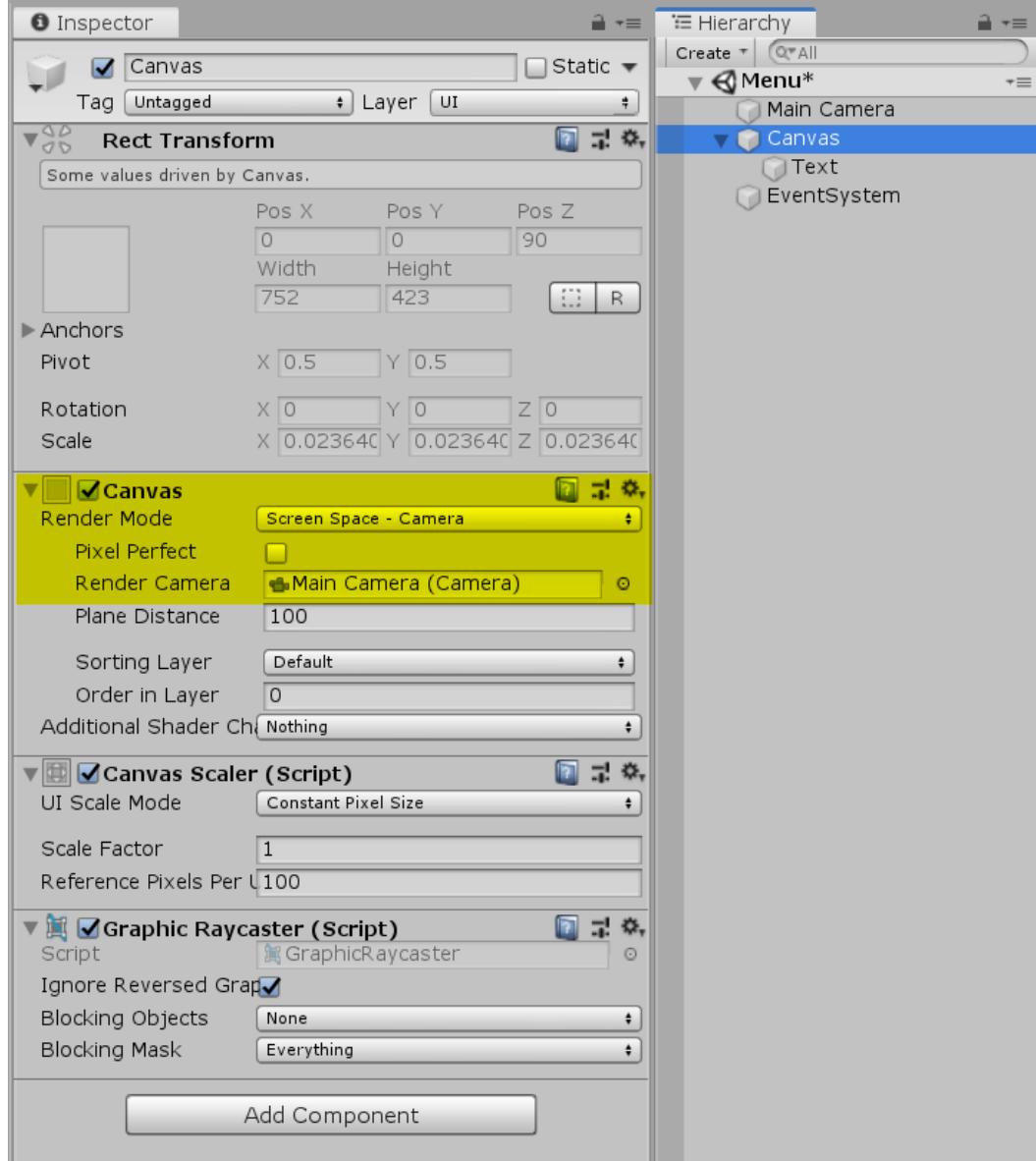
Lo primero será añadir en la jerarquía un Text



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Cuando añadimos el primer elemento, este elemento se incluye dentro de un Canvas y se añaden un EventSystem. Dentro del Canvas incluiremos además de texto con el nombre del juego un botón para iniciar el juego.

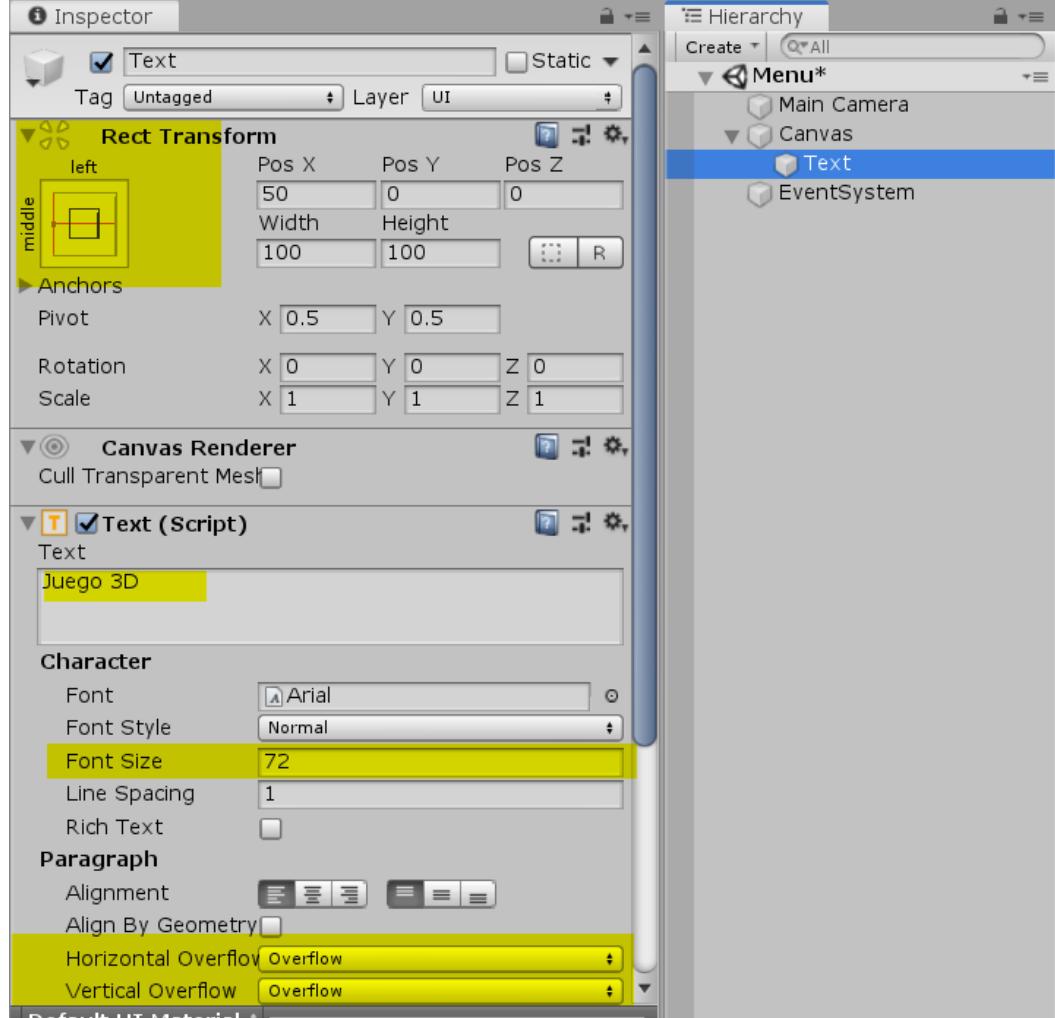
Panel inspector



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)) Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

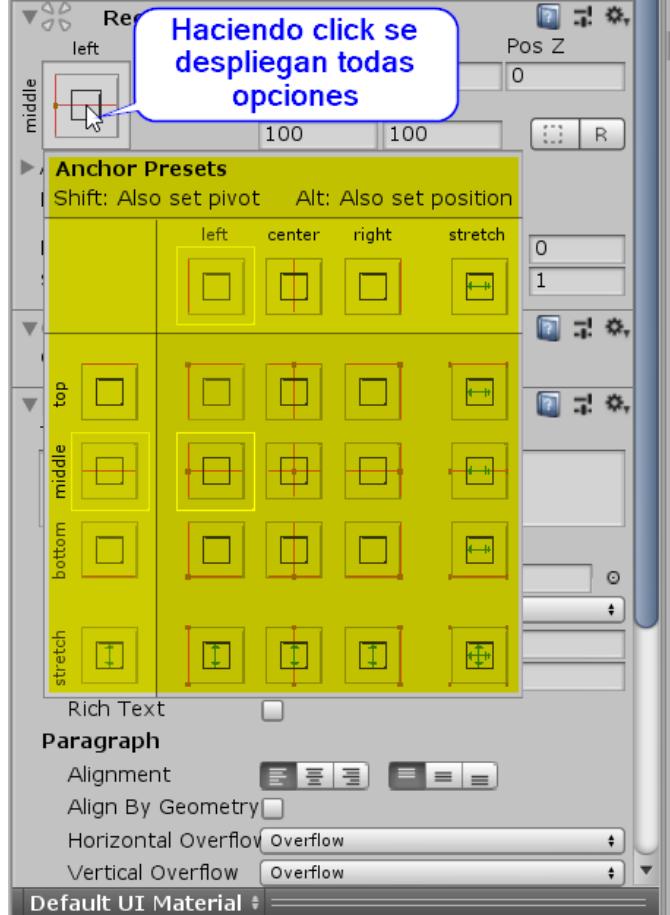
En el Canvas indicaremos el modo de renderizado que vamos a utilizar, Screen Space – Camera, y qué queremos utilice la cámara principal para renderizar.

Panel inspector



BIRT LH ([Copyright \(cita\)\).\(link: https://unity.com/\)](https://unity.com/)) Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

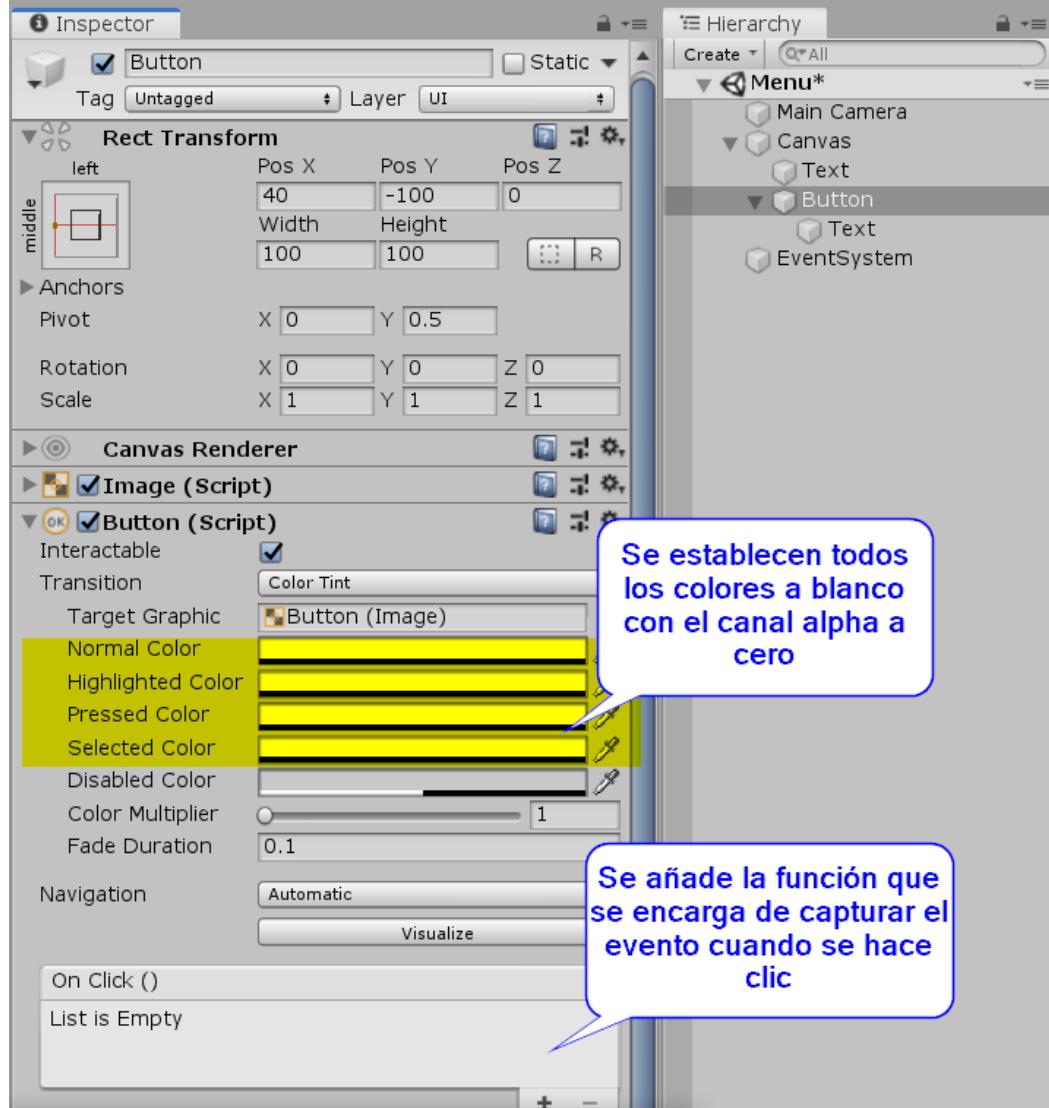
En el texto indicaremos el nombre del juego, el tamaño de la letra y permitiremos el overflow o desbordamiento. Por último, para posicionar el texto haremos uso de las anclas.



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

A continuación, añadiremos un botón para iniciar el juego. El botón aparecerá como un hijo del Canvas. A su vez el botón tendrá un hijo que nos permitirá especificar el texto del botón.

Panel inspector



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

1.2.2.- SCRIPT CARGA DE LA ESCENA.

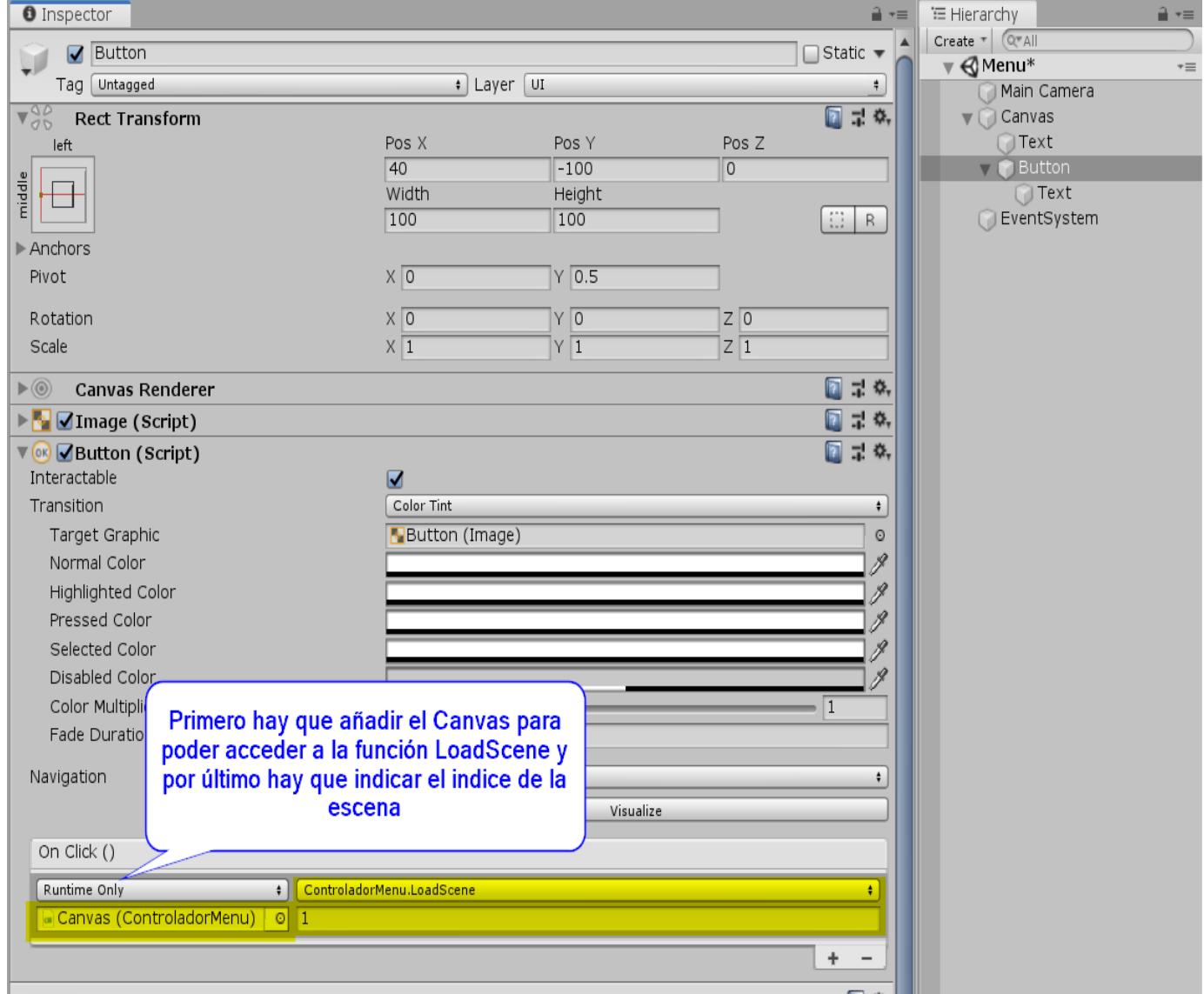
Una vez está definida la escena que contiene el menú se debe codificar el script que se va a encargar de cargar la escena. Para ello creamos un script con el siguiente contenido:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class ControladorMenu : MonoBehaviour
{
    public void LoadScene(int level)
    {
        SceneManager.LoadScene(level);
    }
}
```

El script se lo añadiremos al Canvas e indicaremos que queremos que se ejecute LoadScene una vez que se haga clic en el botón.

Panel inspector

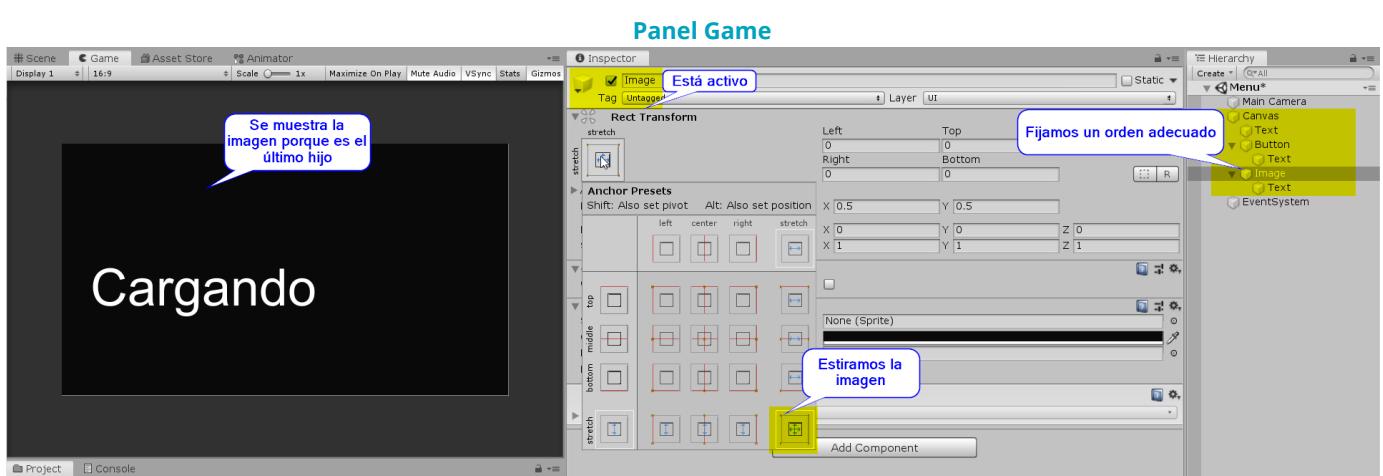


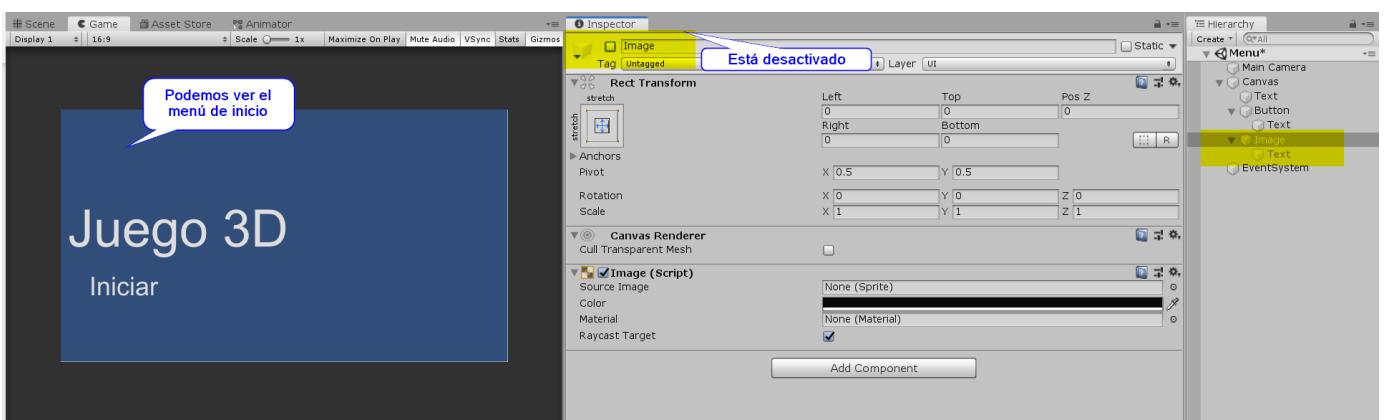
BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

1.2.3.- MENSAJE DE CARGA.

Para evitar que el jugador o jugadora no sepa si ha hecho clic sobre el botón cuando las escenas son pesadas vamos a incluir un mensaje que indique que se está realizando la carga. Otra opción puede ser cargar la escena previamente o utilizar una carga asíncrona.

Para ello vamos a incluir una imagen en el interfaz de usuario y vamos a añadir un texto que indique que se está produciendo la carga de la escena. Dentro del UI el orden de los hijos es importante ya que cuanto más abajo se encuentre un hijo se colocará en una capa superior y ocultará los elementos que están detrás. Por lo tanto, incluiremos la imagen de carga como último hijo y lo desactivaremos. Desde el script activaremos la imagen mientras se produce la carga.





BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

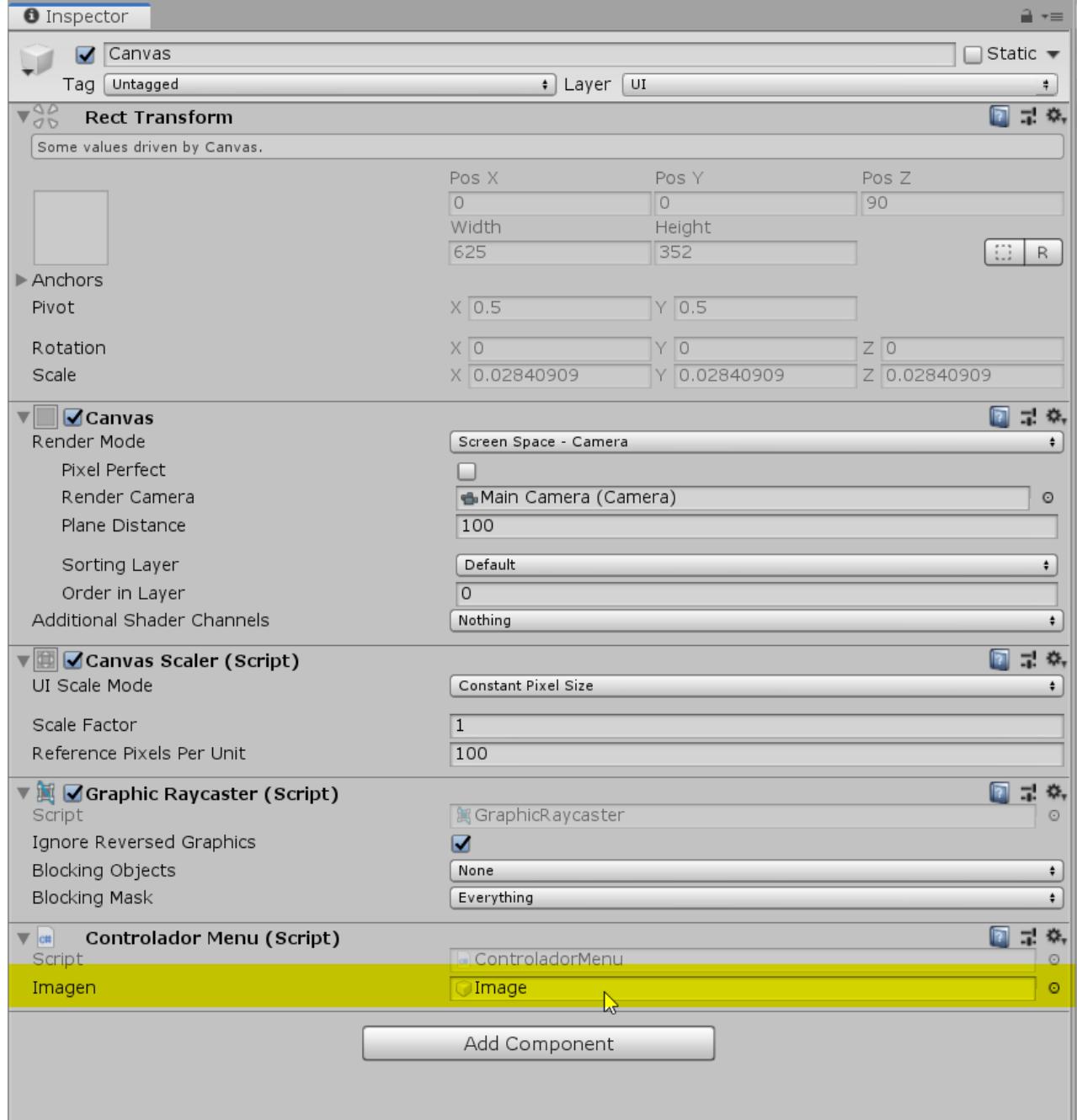
En el script que se encarga de realizar la carga de la escena realizaremos los siguientes cambios y añadiremos la referencia a la imagen en el editor:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class ControladorMenu : MonoBehaviour
{
    public GameObject imagen;

    public void LoadScene(int level)
    {
        imagen.SetActive(true);
        SceneManager.LoadScene(level);
    }
}
```

Panel inspector



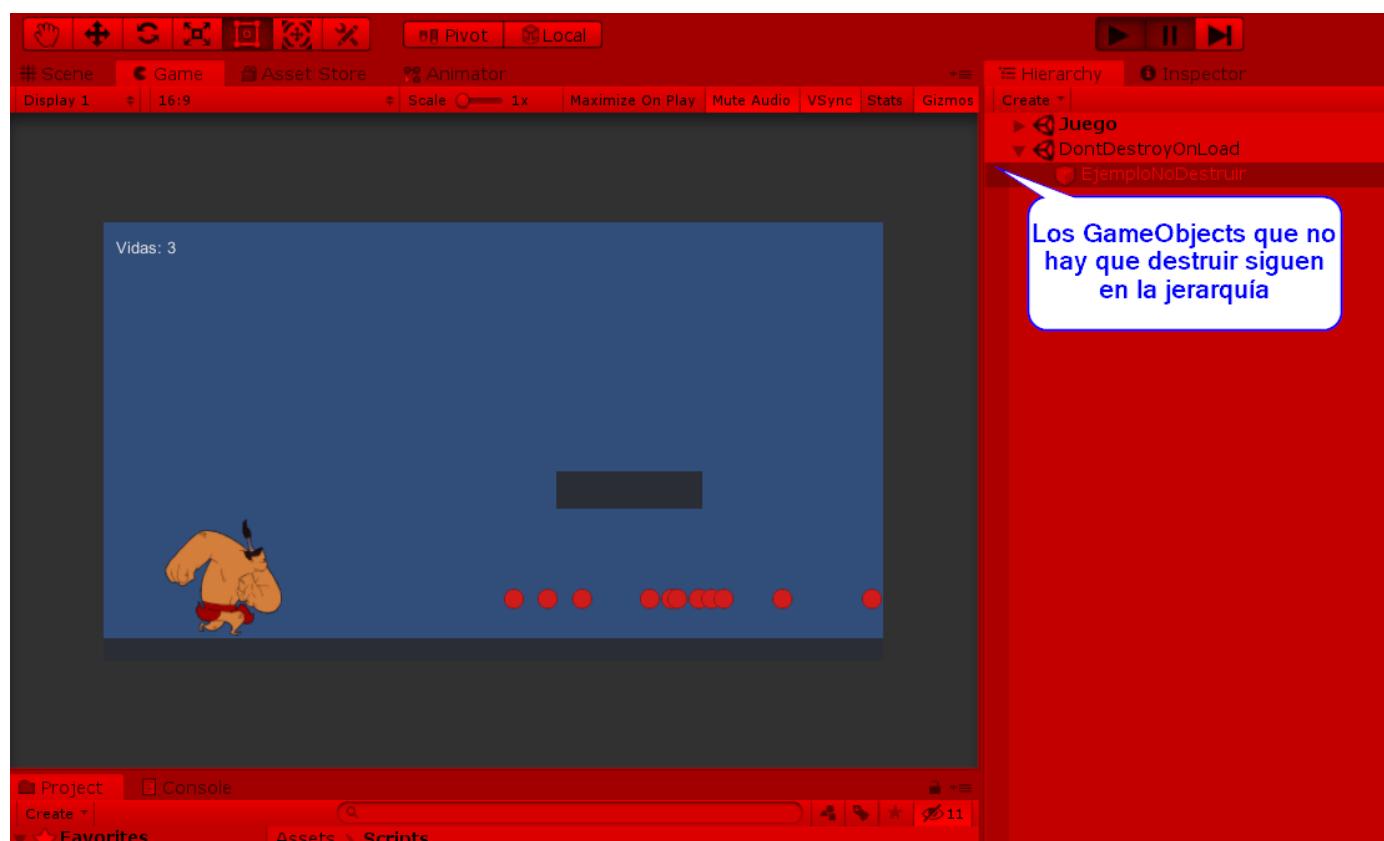
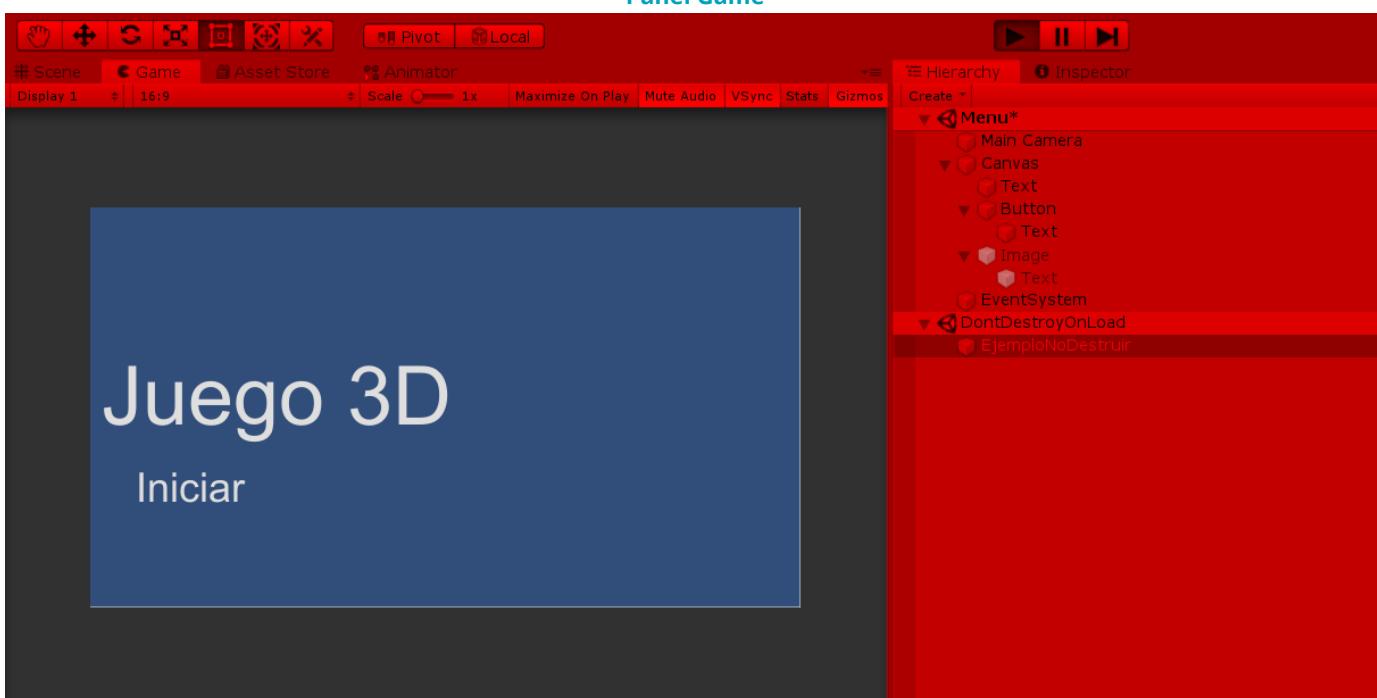
BIRT LH ([Copyright \(cita\)\).\(link: https://unity.com/\)](https://unity.com/). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

1.3.- DONTDESTROYONLOAD.

Cuanado se produce la carga de una nueva escena en Unity se destruyen todos los GameObjects que contiene la escena anterior. Aunque en nuestro ejemplo no tengamos que mantener elementos dentro de la jerarquía entre las dos escenas. Puede que queramos que un GameObject no se destruya al cargar una nueva escena. Para ello debemos utilizar la función DontDestroyOnLoad. Existen muchos casos en lo que no vamos a querer que se destruyan algunos GameObject, un ejemplo son los controladores. A continuación, se muestra un ejemplo:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NoDestruir : MonoBehaviour
{
    // Start is called before the first frame update
    void Awake()
    {
        DontDestroyOnLoad(gameObject);
    }
}
```



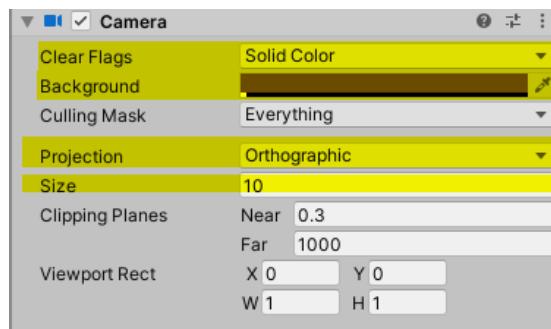
BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

1.4.- COMPONENTES BÁSICOS PARA DESARROLLO DE JUEGOS 3D.

Para realizar el estudio de los componentes básicos para el desarrollo de juegos 3D vamos a utilizar un proyecto que nos ofrece Unity: **Tanks!**. En el proyecto vamos a encontrar la mayoría de los componentes que se utilizan en un Juego 3D. Hay aspectos que no están dentro del alcance del curso y no los veremos y únicamente haremos uso de uso ellos.

1.4.1.- CÁMARA.

Por lo que se refiere a la cámara, vamos a ver como modificar los parámetros de la cámara para poder ver de forma adecuada los objetos que nos interesan. En el inspector podemos ver las siguientes propiedades:



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

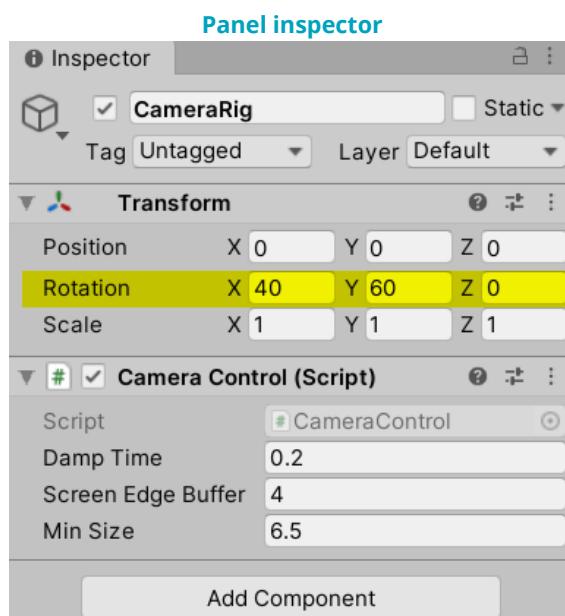
- Clear Flags y Background: indicamos que queremos utilizar un color de relleno en vez de un SkyBox e indicamos el color que queremos usar.
- Projection: en nuestro caso seleccionas la proyección ortográfica que nos permite que los objetos del mismo tamaño se muestren iguales sin tener en cuenta la distancia a la que se encuentran de la cámara.
- Size: el tamaño nos va a permitir aumentar o disminuir el zoom en función de las posiciones de los objetos que queremos mostrar. El cálculo se realizará en un script.

En el proyecto la cámara se encuentra dentro de CameraRig.



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Este objeto es que vamos a utilizar para posicionar la cámara. CameraRig va a estar en punto medio de los diferentes objetos que queremos visualizar y después haciendo uso del parámetro Size de la cámara, conseguiremos que se muestre los objetos de forma adecuada. CameraRig tiene una rotación que va a hacer que al distanciar la cámara en el eje z este se posicione de forma adecuada.



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Además, tal y como se puede ver el objeto cuenta con un Script que se encarga del control de la cámara. Para ello, tal y como se ha mencionado establece para posición de CameraRig y establece el Size de la cámara. El Script cuenta con las siguientes propiedades públicas:

- Damp Time: tiempo para realizar la transición
- Screen Edge Buffer: tamaño de relleno en los bordes, de esta forma se incluye relleno más allá del objeto más lejano.
- Min Size: tamaño mínimo del Size para evitar que se haga demasiado zoom.

El código del controlador es el siguiente:

```
using UnityEngine;

namespace Complete
{
    public class CameraControl : MonoBehaviour
    {
        public float m_DampTime = 0.2f;                      // Approximate time for the camera to refocus.
        public float m_ScreenEdgeBuffer = 4f;                  // Space between the top/bottom most target and the screen edge.
        public float m_MinSize = 6.5f;                         // The smallest orthographic size the camera can be.
        [HideInInspector] public Transform[] m_Tests;          // All the targets the camera needs to encompass.

        private Camera m_Camera;                            // Used for referencing the camera.
        private float m_ZoomSpeed;                         // Reference speed for the smooth damping of the orthographic size.
        private Vector3 m_MoveVelocity;                    // Reference velocity for the smooth damping of the position.
        private Vector3 m_DesiredPosition;                 // The position the camera is moving towards.

        private void Awake ()
        {
            m_Camera = GetComponentInChildren<Camera> ();
        }

        private void FixedUpdate ()
        {
            // Move the camera towards a desired position.
            Move ();

            // Change the size of the camera based.
            Zoom ();
        }

        private void Move ()
        {
            // Find the average position of the targets.
            FindAveragePosition ();

            // Smoothly transition to that position.
            transform.position = Vector3.SmoothDamp(transform.position, m_DesiredPosition, ref m_MoveVelocity, m_DampTime);
        }

        private void FindAveragePosition ()
        {
            Vector3 averagePos = new Vector3 ();
            int numTargets = 0;

            // Go through all the targets and add their positions together.
            for (int i = 0; i < m_Tests.Length; i++)
            {
                // If the target isn't active, go on to the next one.
                if (!m_Tests[i].gameObject.activeSelf)
                    continue;

                // Add to the average and increment the number of targets in the average.
                averagePos += m_Tests[i].position;
                numTargets++;
            }

            // If there are targets divide the sum of the positions by the number of them to find the average.
            if (numTargets > 0)
```

```

        averagePos /= numTargets;

        // Keep the same y value.
        averagePos.y = transform.position.y;

        // The desired position is the average position;
        m_DesiredPosition = averagePos;
    }

private void Zoom ()
{
    // Find the required size based on the desired position and smoothly transition to that size.
    float requiredSize = FindRequiredSize();
    m_Camera.orthographicSize = Mathf.SmoothDamp (m_Camera.orthographicSize, requiredSize, ref m_ZoomSpeed, m_ZoomTime);
}

private float FindRequiredSize ()
{
    // Find the position the camera rig is moving towards in its local space.
    Vector3 desiredLocalPos = transform.InverseTransformPoint(m_DesiredPosition);

    // Start the camera's size calculation at zero.
    float size = 0f;

    // Go through all the targets...
    for (int i = 0; i < m_Tests.Length; i++)
    {
        // ... and if they aren't active continue on to the next target.
        if (!m_Tests[i].gameObject.activeSelf)
            continue;

        // Otherwise, find the position of the target in the camera's local space.
        Vector3 targetLocalPos = transform.InverseTransformPoint(m_Tests[i].position);

        // Find the position of the target from the desired position of the camera's local space.
        Vector3 desiredPosToTarget = targetLocalPos - desiredLocalPos;

        // Choose the largest out of the current size and the distance of the tank 'up' or 'down' from the camera's local space.
        size = Mathf.Max(size, Mathf.Abs(desiredPosToTarget.y));

        // Choose the largest out of the current size and the calculated size based on the tank being to the right or left of the camera's local space.
        size = Mathf.Max(size, Mathf.Abs(desiredPosToTarget.x) / m_Camera.aspect);
    }

    // Add the edge buffer to the size.
    size += m_ScreenEdgeBuffer;

    // Make sure the camera's size isn't below the minimum.
    size = Mathf.Max (size, m_MinSize);

    return size;
}

public void SetStartPositionAndSize ()
{
    // Find the desired position.
    FindAveragePosition ();

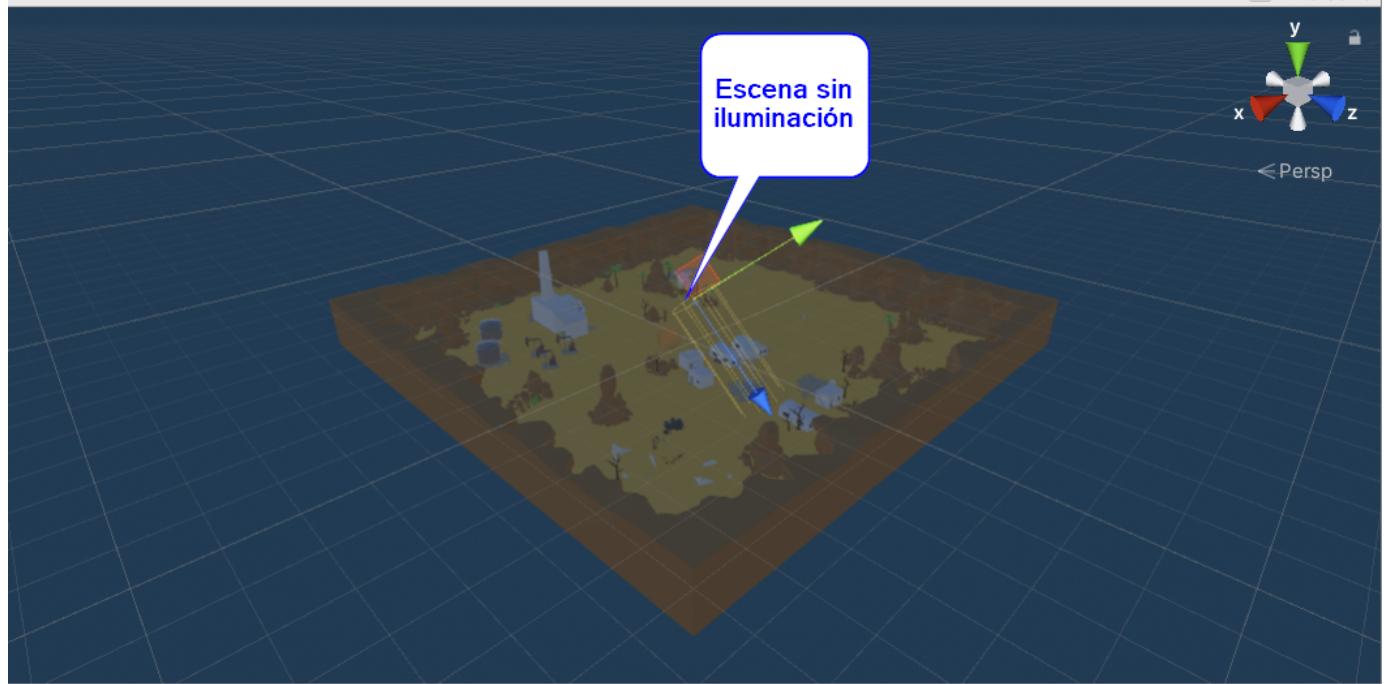
    // Set the camera's position to the desired position without damping.
    transform.position = m_DesiredPosition;

    // Find and set the required size of the camera.
    m_Camera.orthographicSize = FindRequiredSize ();
}
}
}

```

1.4.2.- ILUMINACIÓN.

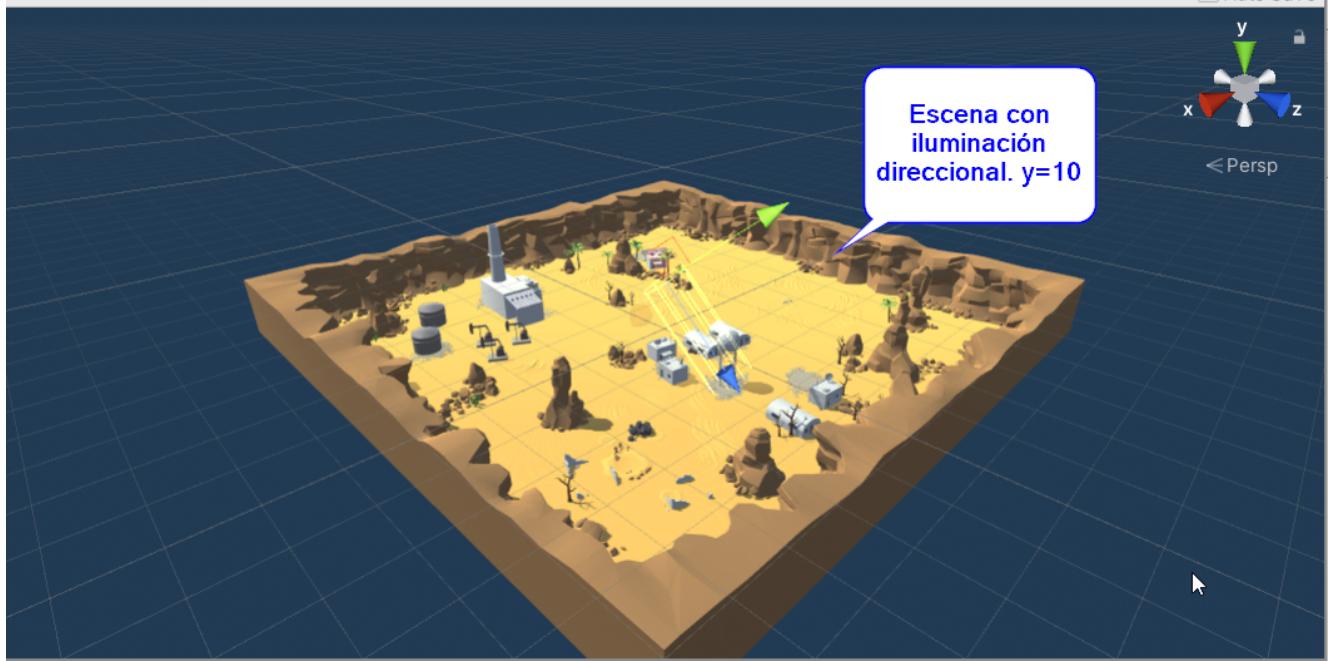
A diferencia de lo que vimos en el tema anterior la iluminación es fundamental en los juegos 3D en Unity y sin fuentes de luz no veríamos nada.



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

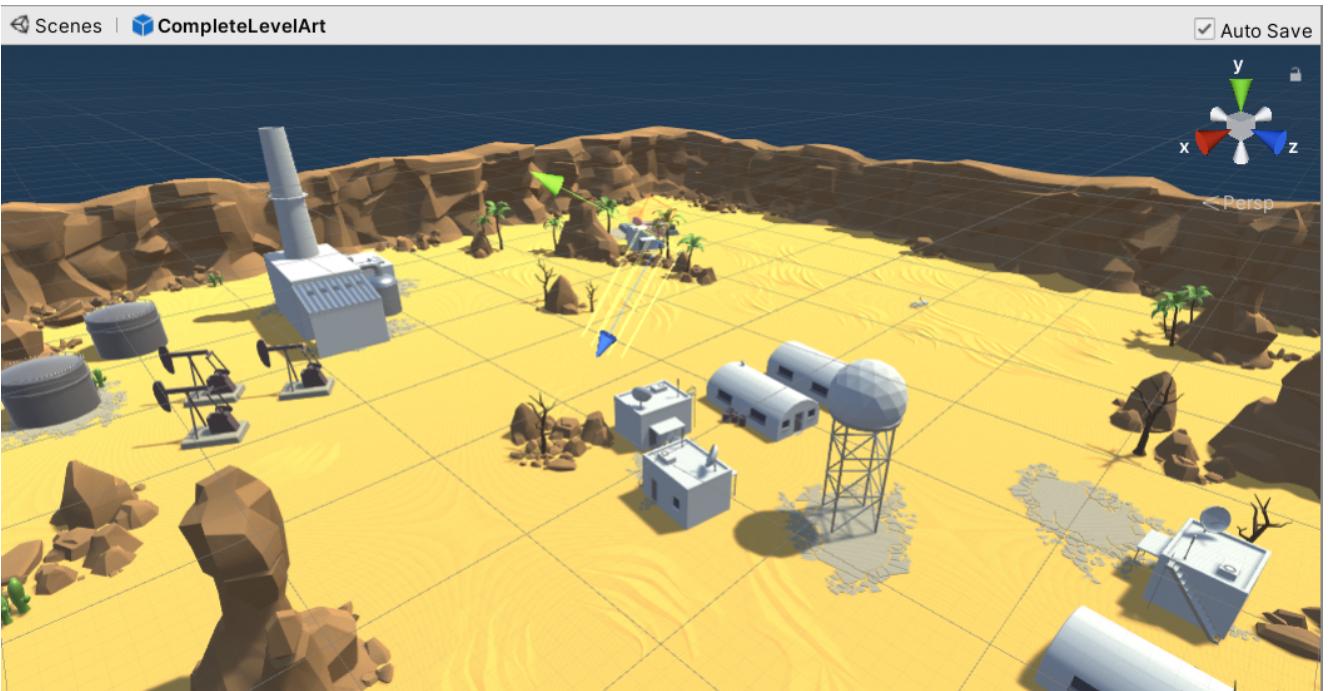
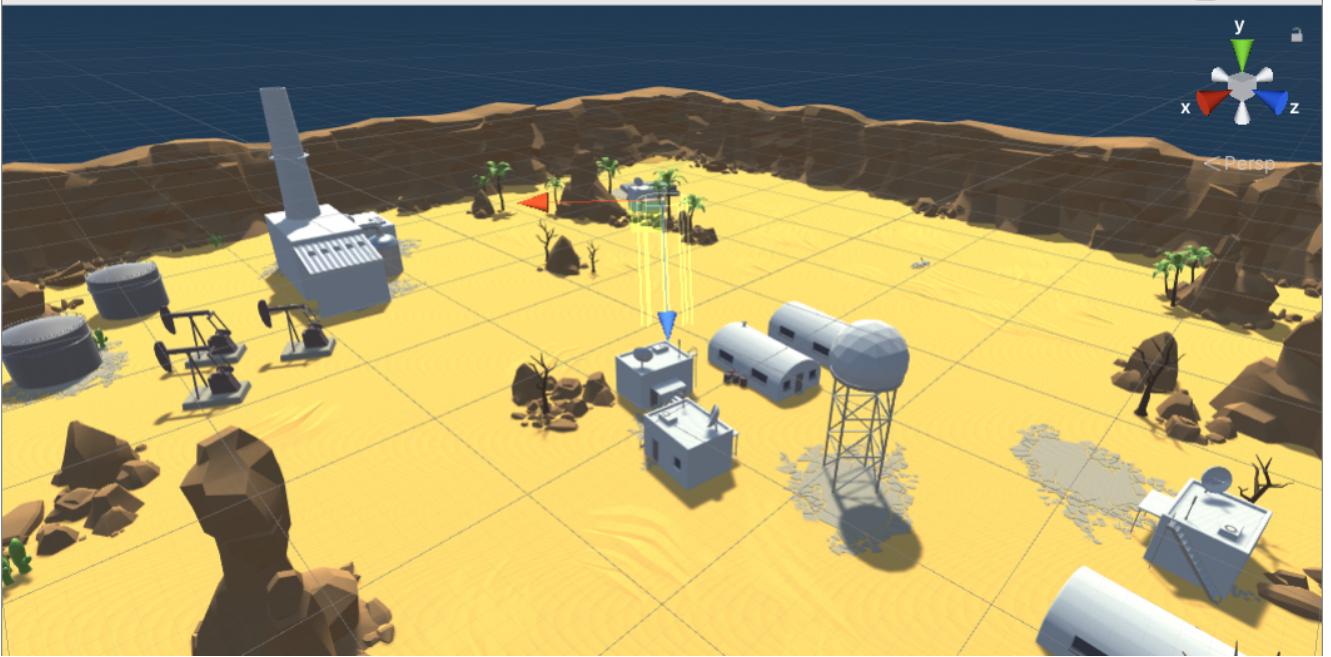
En Unity disponemos diferentes tipos de fuentes de luz. A continuación vamos a ver algunos de ellos:

- **Luz direccional:** está luz queda definida por la rotación del objeto que contiene el punto de luz. La posición en la que se encuentre el punto no es importante y se realizan cambios en la posición, no se van a apreciar cambios en la iluminación.



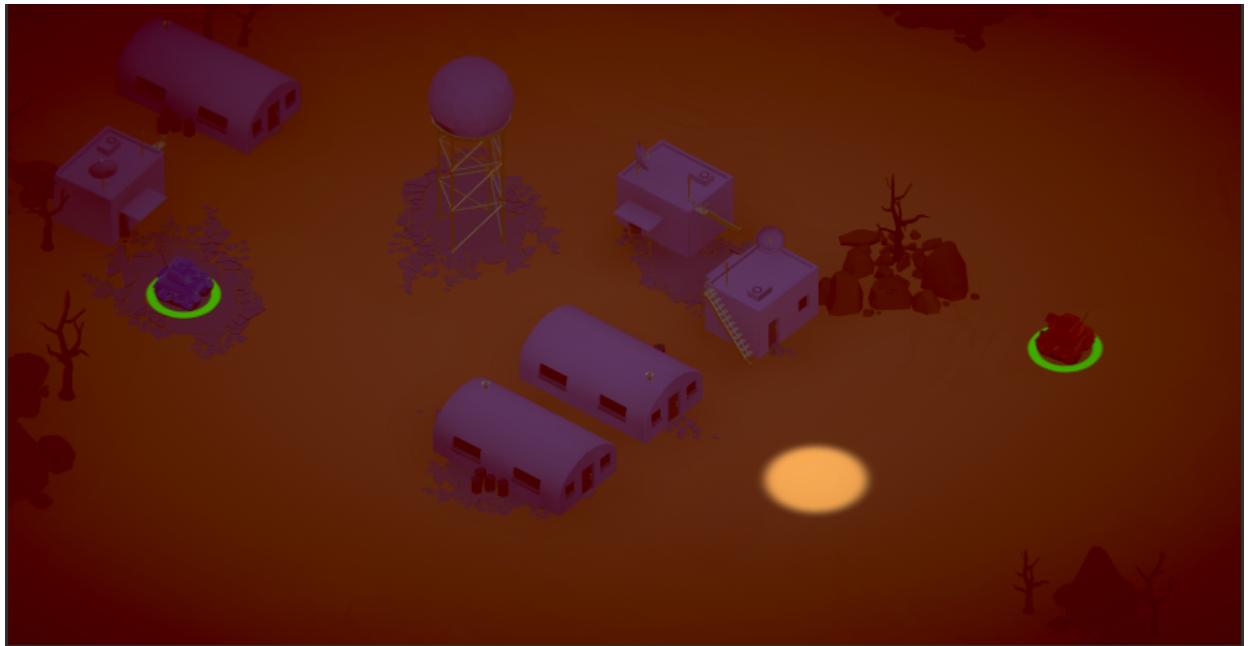
BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unit Technologies

Los cambios se producen al modificar la rotación. Cuando realizamos cambios se consigue que la iluminación de la escena se modifique. Por lo tanto, si queremos modificar mediante código la iluminación de una escena que utilice una luz direccional, modificaremos su rotación.



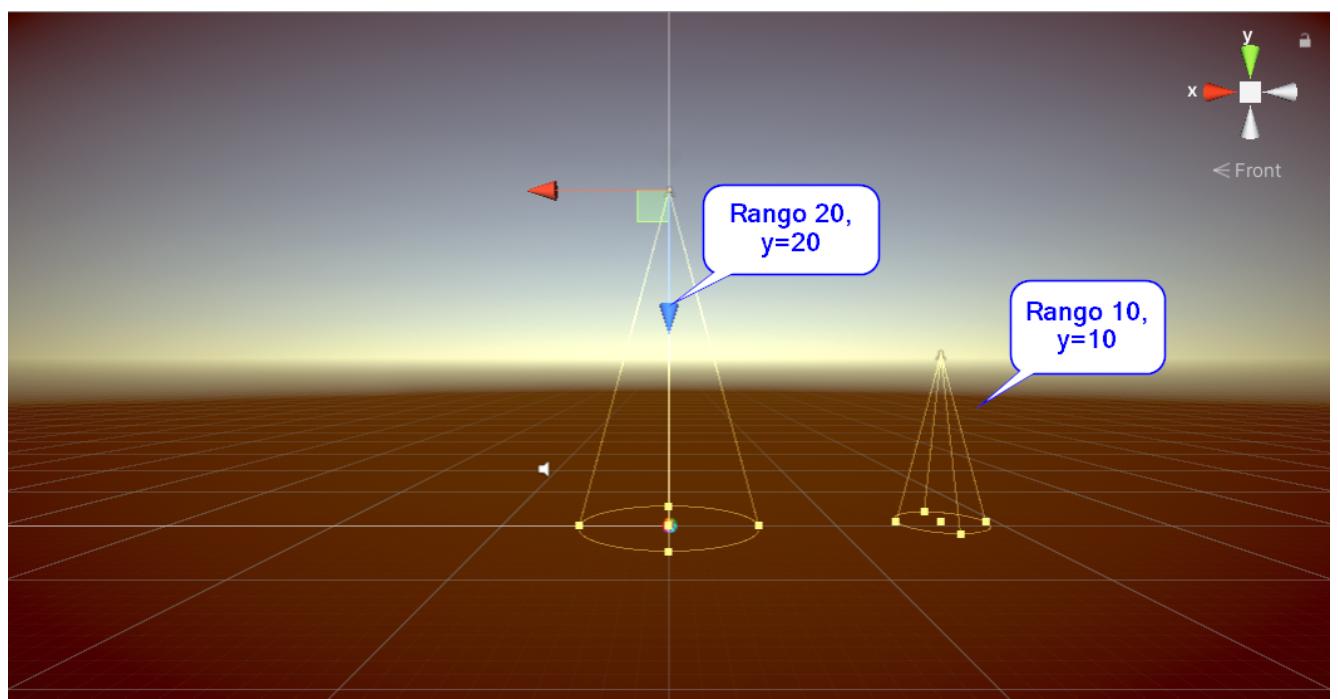
BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unit Technologies

- **Spot:** esta luz queda definida por el origen del punto de luz, el ángulo de apertura y el rango en el que la luz está activa. Aunque la definición pueda parecer difícil, estos puntos de luz se pueden asemejar a una linterna o un cono. En la siguiente imagen podemos ver lo que ocurre al cambiar la luz direccional por una luz de tipo Spot:

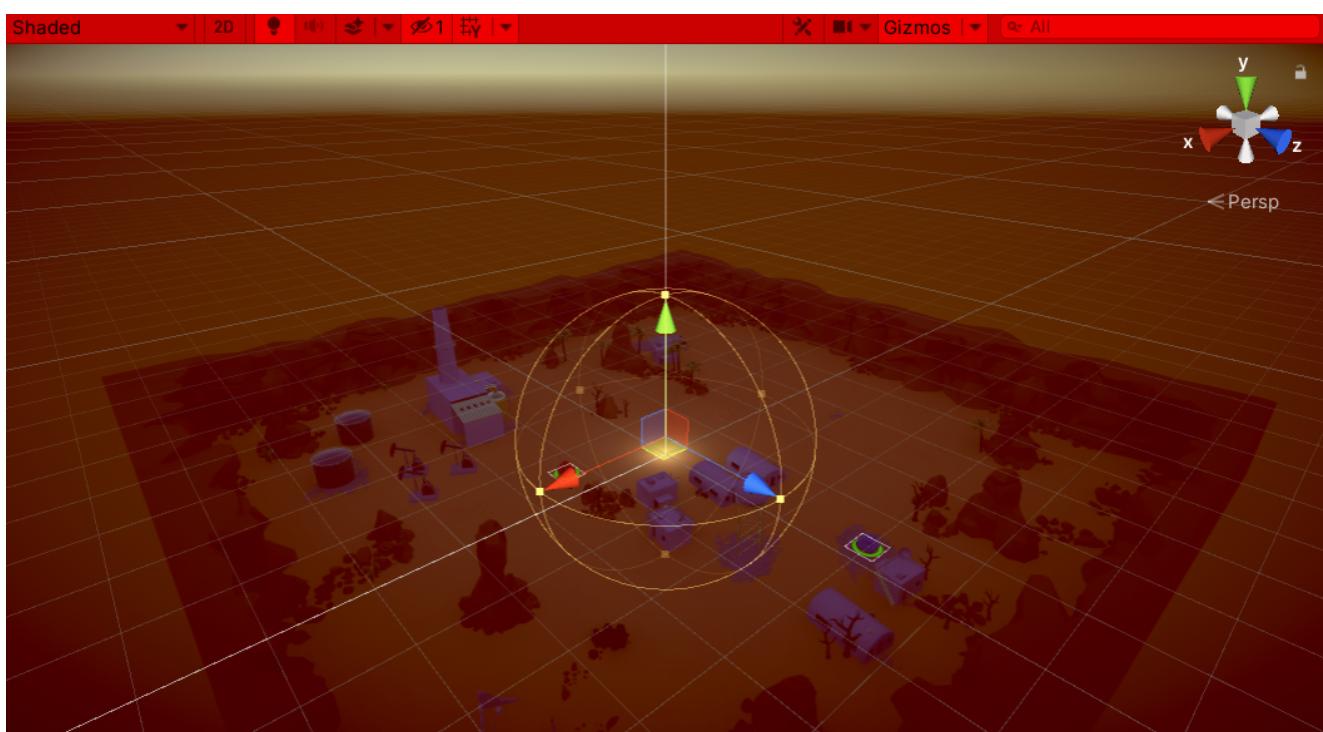


BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)) Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

A continuación, vamos a modificar las propiedades para ver como afectan al gizmo que representa la iluminación:



- **Point:** esta luz queda definida por el punto de emisión y el rango. La intensidad de la luz disminuye con la distancia de la luz, llegando a cero en un rango específico. El gizmo de este tipo de luz es una esfera.

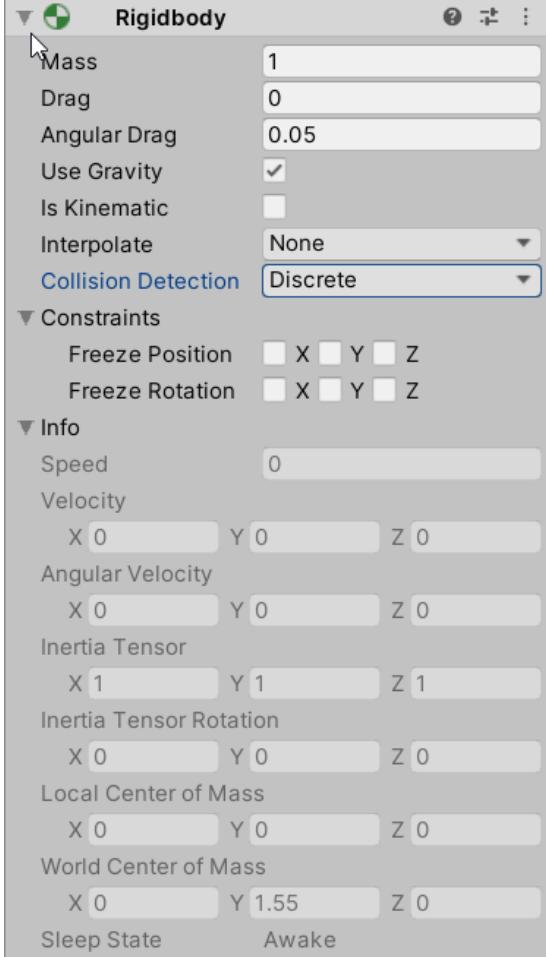


1.4.3.- FÍSICAS 3D.

El uso del motor de físicas que nos ofrece Unity va a hacer que sea más sencilla la manipulación de objetos que queremos que se comporten como si estuvieran bajo el efecto de la gravedad, por ejemplo. Haciendo uso del motor de físicas podremos aplicar una fuerza sobre un objeto para provocar un salto. Si no hacemos uso del motor de físicas tendremos que utilizar ecuaciones para calcular y asignar una posición a nuestro objeto en función de su peso, la fuerza hayamos aplicado y gravedad en cada frame. Es importante recordar que siempre que se realicen acciones que estén relacionadas con el motor de físicas deben realizarse en el método FixedUpdate del componente. De la misma forma si queremos que la cámara actualice su posición en función de un objeto que está controlado por el motor de físicas, tendremos que utilizar el método FixedUpdate.

Rigidbody

De igual manera a como ocurría en el motor de físicas 2D, cuando añadir un Rigidbody hacemos que el GameObject pase a estar controlado por el motor de físicas. En este caso, es el motor de físicas 3D es que será capaz de controlar el GameObject. A continuación, podemos ver un Ridigbody en el inspector:



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Entre las diferencias que podemos observar si lo comparamos con el componente del motor de físicas 2D podemos observar la posibilidad de "congelar" la posición y la rotación en cualquiera de los tres ejes. Por lo que se refiere a los tipos de cuerpos rígidos, únicamente disponemos de marcar el componente como Kinematic o no. Si lo marcamos como Kinematic, el GameObject no se verá afectado por la gravedad, fuerzas o colisiones. Pero si que se podrán mover haciendo uso de las funciones que realizan movimientos de forma explícita. El parámetro Interpolate funciona de la misma manera, por lo tanto, probaremos con la opción de interpolación discreta primero y si no observamos comportamientos extraños, no cambiaremos su valor.

Collider

Tal y como ocurría con los colisionadores 2D, tenemos diferentes colisionadores. En la mayoría de los casos los colisionadores se corresponden con figuras geométricas, ya que es más sencillo realizar los cálculos. A pesar de esto, existe la opción de utilizar un MeshCollider, que se va a ajustar mejor, pero va a consumir más recursos. Estos son los colisionadores disponibles:

- BoxCollider
- SphereCollider
- CapsuleCollider
- MeshCollider
- TerrainCollider
- WheelCollider

En la unidad anterior utilizamos los colisionadores con la propiedad Is Trigger deshabilitada. En esta unidad vamos a habilitar esta opción ya que no vamos a querer que los colisionadores produzcan desplazamiento de forma directa. En esta unidad, una vez que detectemos la colisión, calcularemos el daño en función de la distancia entre los dos colisionadores y por último aplicaremos una fuerza que sí que tendrá efecto sobre los colisionadores.

Uso de capas para realizar búsquedas

En el siguiente script se puede observar como se realiza la búsqueda de colisionadores dentro de una capa concreta. De esta forma se optimizan los cálculos. Además, se aplica una fuerza sobre el cuerpo

```

using UnityEngine;

namespace Complete
{
    public class ShellExplosion : MonoBehaviour
    {
        public LayerMask m_TankMask;
        public ParticleSystem m_ExplosionParticles;
        public AudioSource m_ExplosionAudio;
        public float m_MaxDamage = 100f;
        public float m_ExplosionForce = 1000f;
        public float m_MaxLifeTime = 2f;
        public float m_ExplosionRadius = 5f;

        // Used to filter what the explosion affects, this should
        // Reference to the particles that will play on explosion.
        // Reference to the audio that will play on explosion.
        // The amount of damage done if the explosion is centred.
        // The amount of force added to a tank at the centre of the explosion.
        // The time in seconds before the shell is removed.
        // The maximum distance away from the explosion tanks can be hit.

        private void Start ()
        {
            // If it isn't destroyed by then, destroy the shell after it's lifetime.
            Destroy (gameObject, m_MaxLifeTime);
        }

        private void OnTriggerEnter (Collider other)
        {
            // Collect all the colliders in a sphere from the shell's current position to a radius of the explosion.
            Collider[] colliders = Physics.OverlapSphere (transform.position, m_ExplosionRadius, m_TankMask);

            // Go through all the colliders...
            for (int i = 0; i < colliders.Length; i++)
            {
                // ... and find their rigidbody.
                Rigidbody targetRigidbody = colliders[i].GetComponent<Rigidbody> ();

                // If they don't have a rigidbody, go on to the next collider.
                if (!targetRigidbody)
                    continue;

                // Add an explosion force.
                targetRigidbody.AddExplosionForce (m_ExplosionForce, transform.position, m_ExplosionRadius);

                // Find the TankHealth script associated with the rigidbody.
                TankHealth targetHealth = targetRigidbody.GetComponent<TankHealth> ();

                // If there is no TankHealth script attached to the gameobject, go on to the next collider.
                if (!targetHealth)
                    continue;

                // Calculate the amount of damage the target should take based on it's distance from the shell.
                float damage = CalculateDamage (targetRigidbody.position);

                // Deal this damage to the tank.
                targetHealth.TakeDamage (damage);
            }

            // Unparent the particles from the shell.
            m_ExplosionParticles.transform.parent = null;

            // Play the particle system.
            m_ExplosionParticles.Play();

            // Play the explosion sound effect.
            m_ExplosionAudio.Play();

            // Once the particles have finished, destroy the gameobject they are on.
            ParticleSystem.MainModule mainModule = m_ExplosionParticles.main;
            Destroy (m_ExplosionParticles.gameObject, mainModule.duration);

            // Destroy the shell.
            Destroy (gameObject);
        }

        private float CalculateDamage (Vector3 targetPosition)
        {
            // Create a vector from the shell to the target.

```

```

        Vector3 explosionToTarget = targetPosition - transform.position;

        // Calculate the distance from the shell to the target.
        float explosionDistance = explosionToTarget.magnitude;

        // Calculate the proportion of the maximum distance (the explosionRadius) the target is away.
        float relativeDistance = (m_ExplosionRadius - explosionDistance) / m_ExplosionRadius;

        // Calculate damage as this proportion of the maximum possible damage.
        float damage = relativeDistance * m_MaxDamage;

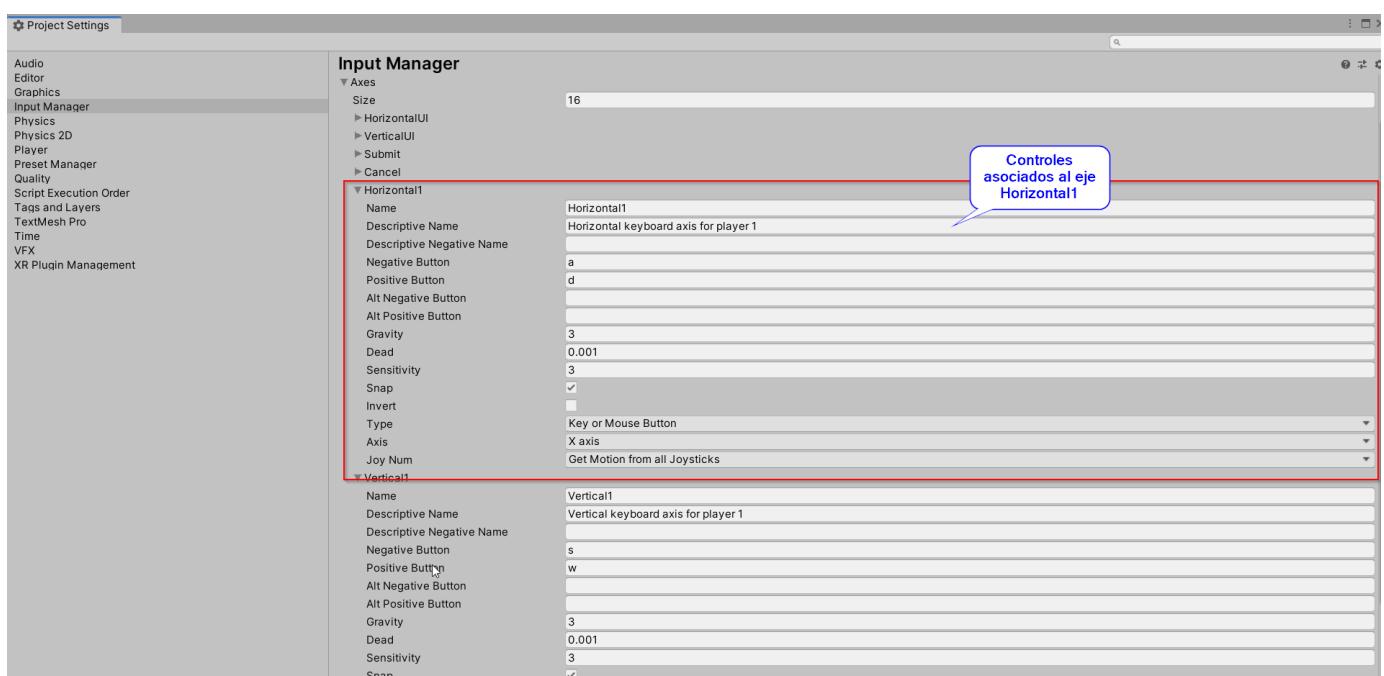
        // Make sure that the minimum damage is always 0.
        damage = Mathf.Max (0f, damage);

        return damage;
    }
}

```

Funciones para actualizar las posiciones de un GameObject a través de un Rigidbody

Para realizar los movimientos del tanque vamos a utilizar las funciones MovePosition y MoveRotation que nos ofrece Rigidbody. Antes de poder llamar a las funciones vamos a tener que conocer el punto al que queremos movernos o la rotación que queremos aplicar. Lo primero será capturas los valores de los ejes Horizontal y Vertical que corresponden al tanque. Para configurar estos valores Unity nos ofrece el siguiente panel:



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Para realizar el movimiento del tanque utilizaremos las funciones Update y FixedUpdate. En la primera actualizaremos los valores en función de las entradas. En la segunda realizaremos las llamadas a los métodos que se encargan de realizar los movimientos.

```

using UnityEngine;

namespace Complete
{
    public class TankMovement : MonoBehaviour
    {
        public int m_PlayerNumber = 1;                                // Used to identify which tank belongs to which player. This is
        public float m_Speed = 12f;                                    // How fast the tank moves forward and back.
        public float m_TurnSpeed = 180f;                               // How fast the tank turns in degrees per second.
        public AudioSource m_MovementAudio;                          // Reference to the audio source used to play engine sounds. NB:
        public AudioClip m_EngineIdling;                            // Audio to play when the tank isn't moving.
        public AudioClip m_EngineDriving;                           // Audio to play when the tank is moving.
        public float m_PitchRange = 0.2f;                            // The amount by which the pitch of the engine noises can
    }
}

```

```

private string m_MovementAxisName;           // The name of the input axis for moving forward and back.
private string m_TurnAxisName;                // The name of the input axis for turning.
private Rigidbody m_Rigidbody;               // Reference used to move the tank.
private float m_MovementInputValue;          // The current value of the movement input.
private float m_TurnInputValue;              // The current value of the turn input.
private float m_OriginalPitch;               // The pitch of the audio source at the start of the scene.
private ParticleSystem[] m_particleSystems; // References to all the particles systems used by the Tanks

private void Awake ()
{
    m_Rigidbody = GetComponent<Rigidbody> ();
}

private void OnEnable ()
{
    // When the tank is turned on, make sure it's not kinematic.
    m_Rigidbody.isKinematic = false;

    // Also reset the input values.
    m_MovementInputValue = 0f;
    m_TurnInputValue = 0f;

    // We grab all the Particle systems child of that Tank to be able to Stop/Play them on Deactivate/Activate
    // It is needed because we move the Tank when spawning it, and if the Particle System is playing while we
    // it "think" it move from (0,0,0) to the spawn point, creating a huge trail of smoke
    m_particleSystems = GetComponentsInChildren<ParticleSystem>();
    for (int i = 0; i < m_particleSystems.Length; ++i)
    {
        m_particleSystems[i].Play();
    }
}

private void OnDisable ()
{
    // When the tank is turned off, set it to kinematic so it stops moving.
    m_Rigidbody.isKinematic = true;

    // Stop all particle system so it "reset" it's position to the actual one instead of thinking we moved wh
    for(int i = 0; i < m_particleSystems.Length; ++i)
    {
        m_particleSystems[i].Stop();
    }
}

private void Start ()
{
    // The axes names are based on player number.
    m_MovementAxisName = "Vertical" + m_PlayerNumber;
    m_TurnAxisName = "Horizontal" + m_PlayerNumber;

    // Store the original pitch of the audio source.
    m_OriginalPitch = m_MovementAudio.pitch;
}

private void Update ()
{
    // Store the value of both input axes.
    m_MovementInputValue = Input.GetAxis (m_MovementAxisName);
    m_TurnInputValue = Input.GetAxis (m_TurnAxisName);

    EngineAudio ();
}

private void EngineAudio ()
{
    // If there is no input (the tank is stationary)...
    if (Mathf.Abs (m_MovementInputValue) < 0.1f && Mathf.Abs (m_TurnInputValue) < 0.1f)
    {
        // ... and if the audio source is currently playing the driving clip...
        if (m_MovementAudio.clip == m_EngineDriving)
        {
            /

```

```

        // ... change the clip to idling and play it.
        m_MovementAudio.clip = m_EngineIdling;
        m_MovementAudio.pitch = Random.Range (m_OriginalPitch - m_PitchRange, m_OriginalPitch + m_PitchRange);
        m_MovementAudio.Play ();
    }
}
else
{
    // Otherwise if the tank is moving and if the idling clip is currently playing...
    if (m_MovementAudio.clip == m_EngineIdling)
    {
        // ... change the clip to driving and play.
        m_MovementAudio.clip = m_EngineDriving;
        m_MovementAudio.pitch = Random.Range(m_OriginalPitch - m_PitchRange, m_OriginalPitch + m_PitchRange);
        m_MovementAudio.Play();
    }
}
}

private void FixedUpdate ()
{
    // Adjust the rigidbodies position and orientation in FixedUpdate.
    Move ();
    Turn ();
}

private void Move ()
{
    // Create a vector in the direction the tank is facing with a magnitude based on the input, speed and the
    Vector3 movement = transform.forward * m_MovementInputValue * m_Speed * Time.deltaTime;

    // Apply this movement to the rigidbody's position.
    m_Rigidbody.MovePosition(m_Rigidbody.position + movement);
}

private void Turn ()
{
    // Determine the number of degrees to be turned based on the input, speed and time between frames.
    float turn = m_TurnInputValue * m_TurnSpeed * Time.deltaTime;

    // Make this into a rotation in the y axis.
    Quaternion turnRotation = Quaternion.Euler (0f, turn, 0f);

    // Apply this rotation to the rigidbody's rotation.
    m_Rigidbody.MoveRotation (m_Rigidbody.rotation * turnRotation);
}
}
}

```

1.4.4.- SISTEMAS DE PARTÍCULAS.

En esta sección vamos a ver como utilizar sistemas de partículas que ya están diseñados. Por lo tanto, lo único que haremos será hacer uso de ellos. Para iniciar un sistema de partículas utilizaremos la función Play y para pararlo la función Stop. A las funciones de Play y Stop les llamaremos desde uno de los métodos de inicialización y desde uno de los métodos de desmantelamiento. A continuación, se muestra un fragmento de un controlador en el que podemos observar cómo se inicia el sistema de partículas y cómo se para.

```

private void OnEnable ()
{
    // When the tank is turned on, make sure it's not kinematic.
    m_Rigidbody.isKinematic = false;

    // Also reset the input values.
    m_MovementInputValue = 0f;
    m_TurnInputValue = 0f;

    // We grab all the Particle systems child of that Tank to be able to Stop/Play them on Deactivate/Activate
    // It is needed because we move the Tank when spawning it, and if the Particle System is playing while we
    // it "think" it move from (0,0,0) to the spawn point, creating a huge trail of smoke
    m_particleSystems = GetComponentsInChildren<ParticleSystem>();
    for (int i = 0; i < m_particleSystems.Length; ++i)

```

```

    {
        m_particleSystems[i].Play();
    }

}

private void OnDisable ()
{
    // When the tank is turned off, set it to kinematic so it stops moving.
    m_Rigidbody.isKinematic = true;

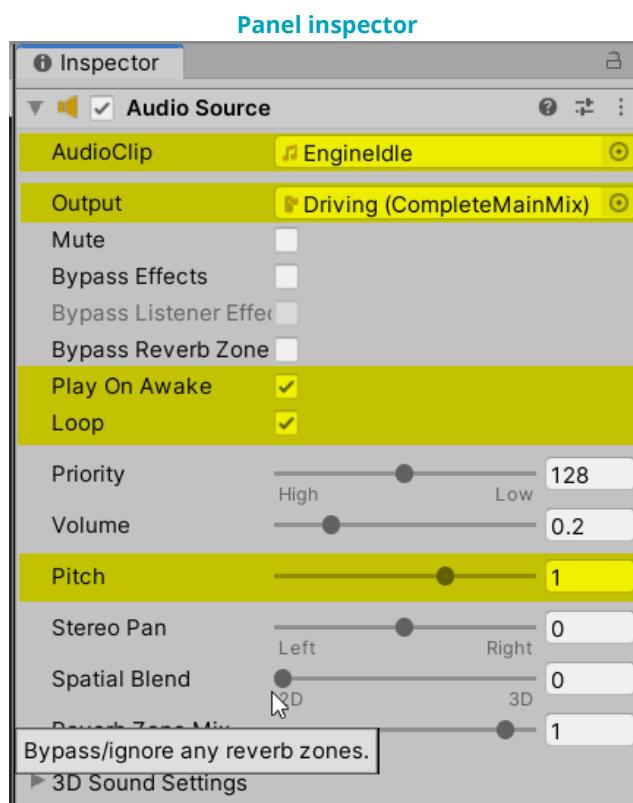
    // Stop all particle system so it "reset" its position to the actual one instead of thinking we moved wh
    for(int i = 0; i < m_particleSystems.Length; ++i)
    {
        m_particleSystems[i].Stop();
    }
}

```

1.4.5.- AUDIO.

Por último, vamos a ver los componentes relacionados con el audio. Para incluir audios en Unity podemos utilizar el componente AudioSource. Aunque el componente tiene diferentes propiedades nos vamos a centrar en las siguientes:

- Audio Clip: nos permite indicar el audio que queremos reproducir. Esta propiedad se puede modificar mediante código, pero al hacerlo el reproductor se detiene. Por lo tanto, si queremos realizar un cambio de clip y que el audio siga reproduciéndose, tendremos que indicarlos de forma explícita.
- Output: nos permite indicar la salida del audio. De esta forma podremos configurar el Audio Mixer
- Play On Awake: nos permite indicar si queremos que la reproducción se inicie cuando se ejecute el método OnAwake del componente
- Loop: nos permite indicar si queremos que el audio este en bucle.
- Pitch: nos permite indicar el tono del audio. Modificando esta propiedad a través de código se puede conseguir que el mismo audio suene de forma diferente.



BIRT LH ([Copyright \(cita\)](#)).([link: https://unity.com/](https://unity.com/)). Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

A continuación, se muestra como modificar el Audio Clip y el pitch mediante código en función de estado del GameObject:

```
private void EngineAudio ()
{
    // If there is no input (the tank is stationary)...
    if (Mathf.Abs (m_MovementInputValue) < 0.1f && Mathf.Abs (m_TurnInputValue) < 0.1f)
    {
        // ... and if the audio source is currently playing the driving clip...
        if (m_MovementAudio.clip == m_EngineDriving)
        {
            // ... change the clip to idling and play it.
            m_MovementAudio.clip = m_EngineIdling;
            m_MovementAudio.pitch = Random.Range (m_OriginalPitch - m_PitchRange, m_OriginalPitch + m_PitchRa
            m_MovementAudio.Play ();
        }
    }
    else
    {
        // Otherwise if the tank is moving and if the idling clip is currently playing...
        if (m_MovementAudio.clip == m_EngineIdling)
        {
            // ... change the clip to driving and play.
            m_MovementAudio.clip = m_EngineDriving;
            m_MovementAudio.pitch = Random.Range(m_OriginalPitch - m_PitchRange, m_OriginalPitch + m_PitchRan
            m_MovementAudio.Play();
        }
    }
}
```

Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0 (*link: <http://creativecommons.org/licenses/by-sa/4.0/>*)