EBook Gratis

APRENDIZAJE unity3d

Free unaffiliated eBook created from
Stack Overflow contributors.

#unity3d

Tabla de contenido

Acerca de ... 1

Capítulo 1: Empezando con unity3d ... 2

Observaciones ... 2

Versiones ... 2

Examples ... 5

Instalación o configuración ... 5

Visión general ... 5

Instalación ... 6

Instalando múltiples versiones de Unity ... 6

Editor y código básico ... 6

Diseño ... 6

Diseño de Linux ... 7

Uso básico ... 7

Scripting básico ... 8

Diseños de editor ... 8

Personalizando tu espacio de trabajo ... 10

Capítulo 2: Agrupación de objetos ... 13

Examples ... 13

Conjunto de objetos ... 13

Conjunto de objetos simples ... 15

Otro grupo de objetos simples ... 17

Capítulo 3: Animación de la unidad ... 19

Examples ... 19

Animación básica para correr ... 19

Creación y uso de clips de animación ... 20

Animación 2D Sprite ... 22

Curvas de animacion ... 24

Capítulo 4: API de CullingGroup ... 27

Observaciones ... 27

Examples ... 27

Recoger distancias de objetos 27
Visibilidad de objetos de selección 29
Distancias delimitadas 30
Visualizando distancias de límite. 30
Capítulo 5: Atributos 32
Sintaxis 32
Observaciones 32
SerializeField 32
Examples 33
Atributos de inspector comunes 33
Atributos del componente 35
Atributos de tiempo de ejecución 36
Atributos del menú 37
Atributos del editor 39
Capítulo 6: Capas 43
Examples 43
Uso de la capa 43
Estructura LayerMask 43
Capítulo 7: Colisión 45
Examples 45
Colisionadores 45
Box Collider 45
Propiedades 45
Ejemplo 46
Colisionador de esfera 46
Propiedades 46
Ejemplo 47
Colisionador de cápsulas 47
Propiedades 48
Ejemplo 48
Colisionador de ruedas 48

Propiedades ... 48
Muelle de suspensión ... 49
Ejemplo ... 49
Colisionador de malla ... 49
Propiedades ... 50
Ejemplo ... 51
Colisionador de ruedas ... 51
Colisionadores de gatillo ... 53
Métodos ... 53
Trigger Collider Scripting ... 54
Ejemplo ... 54
Capítulo 8: Cómo utilizar paquetes de activos ... 55
Examples ... 55
Paquetes de activos ... 55
Importando un paquete .unity ... 55
Capítulo 9: Comunicación con el servidor ... 57
Examples ... 57
Obtener ... 57
Post simple (Campos Post) ... 57
Publicar (subir un archivo) ... 58
Subir un archivo zip al servidor ... 58
Enviando una solicitud al servidor ... 58
Capítulo 10: Coroutines ... 61
Sintaxis ... 61
Observaciones ... 61
Consideraciones de rendimiento ... 61
Reducir la basura mediante el almacenamiento en caché de instrucciones de rendimiento ... 61
Examples ... 61
Coroutines ... 61
Ejemplo ... 63
Acabando con una coroutine ... 63

MonoBehaviour métodos que pueden ser los coroutines. 65
Encadenamiento de coroutines 65
Maneras de ceder 67
Capítulo 11: Cuaterniones 70
Sintaxis 70
Examples 70
Introducción a Quaternion vs Euler 70
Quaternion Look Rotation 70
Capítulo 12: Desarrollo multiplataforma 72
Examples 72
Definiciones del compilador 72
Plataforma organizativa de métodos específicos para clases parciales. 72
Capítulo 13: Encontrar y coleccionar GameObjects 74
Sintaxis 74
Observaciones 74
Qué método utilizar 74
Yendo mas profundo 74
Examples 75
Buscando por nombre de GameObject 75
Buscando por las etiquetas de GameObject 75
Insertado a scripts en el modo de edición 75
Encontrando guiones de GameObjects by MonoBehaviour 75
Encuentra GameObjects por nombre de objetos secundarios 76
Capítulo 14: Etiquetas 77
Introducción 77
Examples 77
Creación y aplicación de etiquetas 77
Estableciendo etiquetas en el editor 77
Configuración de etiquetas a través de secuencias de comandos 77
Creación de etiquetas personalizadas 78
Encontrar GameObjects por etiqueta: 79

Encontrar un solo objeto de GameObject ... 79
Encontrar una matriz de instancias de GameObject ... 80
Comparando etiquetas ... 80
Capítulo 15: Extendiendo el Editor ... 82
Sintaxis ... 82
Parámetros ... 82
Examples ... 82
Inspector personalizado ... 82
Cajón de propiedad personalizada ... 84
Elementos de menú ... 87
Gizmos ... 92
Ejemplo uno ... 92
Ejemplo dos ... 94
Resultado ... 94
No seleccionado ... 95
Seleccionado ... 95
Ventana del editor ... 96
¿Por qué una ventana de editor? ... 96
Crear un Editor de Windows básico ... 96
Ejemplo simple ... 96
Yendo mas profundo ... 97
Temas avanzados ... 100
Dibujando en el SceneView ... 100
Capítulo 16: Física ... 105
Examples ... 105
Cuerpos rígidos ... 105
Visión general ... 105
Añadiendo un componente de Rigidbody ... 105
Moviendo un objeto Rigidbody ... 105
Masa ... 105
Arrastrar ... 105

es cinemático 106
Restricciones 106
Colisiones 106
Gravedad en Cuerpo Rigido 107
Capítulo 17: Iluminación de la unidad 109
Examples 109
Tipos de luz 109
Luz de área 109
Luz direccional 109
Luz puntual 110
Destacar 111
Nota sobre las sombras 112
Emisión 113
Capítulo 18: Implementación de la clase MonoBehaviour. 115
Examples 115
No hay métodos anulados 115
Capítulo 19: Importadores y (Post) Procesadores 116
Sintaxis 116
Observaciones 116
Examples 116
Postprocesador de texturas 116
Un importador básico 117
Capítulo 20: Integración de anuncios 121
Introducción 121
Observaciones 121
Examples 121
Conceptos básicos de Unity Ads en C # 121
Conceptos básicos de Unity Ads en JavaScript 122
Capítulo 21: Mejoramiento 123
Observaciones 123
Examples 123

Cheques rápidos y eficientes ... 123
Controles de distancia / rango ... 123
Cheques de límites ... 123
Advertencias ... 123
Poder coroutine ... 124
Uso ... 124
División de rutinas de larga ejecución en varios marcos ... 124
Realización de acciones caras con menos frecuencia ... 124
Errores comunes ... 125
Instrumentos de cuerda ... 125
Las operaciones de cuerdas construyen basura ... 125
Caché tus operaciones de cadena ... 125
La mayoría de las operaciones de cadena son mensajes de depuración ... 126
Comparación de cuerdas ... 127
Referencias de caché ... 127
Evitar los métodos de llamada utilizando cadenas ... 128
Evita los métodos de unidad vacía ... 129
Capítulo 22: Patrones de diseño ... 130
Examples ... 130
Modelo de diseño del controlador de vista (MVC) ... 130
Capítulo 23: Plataformas moviles ... 134
Sintaxis ... 134
Examples ... 134
Detección de toque ... 134
TouchPhase ... 134
Capítulo 24: Plugins de Android 101 - Una introducción ... 136
Introducción ... 136
Observaciones ... 136
Comenzando con los complementos de Android ... 136
Esquema para crear un plugin y terminología ... 136

Elegir entre los métodos de creación de plugins. 137
Examples 137
UnityAndroidPlugin.cs 137
UnityAndroidNative.java 137
UnityAndroidPluginGUI.cs 138
Capítulo 25: Prefabricados 139
Sintaxis 139
Examples 139
Introducción 139
Creación de prefabs 139
Inspector prefabricado 140
Creación de prefabs 141
Tiempo de diseño de instanciación 141
Instanciación en tiempo de ejecución 142
Prefabricados anidados 142
Capítulo 26: Raycast 147
Parámetros 147
Examples 147
Física Raycast 147
Physics2D Raycast2D 147
Encapsulando llamadas Raycast 148
Otras lecturas 149
Capítulo 27: Realidad Virtual (VR) 150
Examples 150
Plataformas VR 150
SDKs: 150
Documentación: 150
Habilitar el soporte de VR 150
Hardware 151
Capítulo 28: Recursos 153
Examples 153

Introducción 153
Recursos 101 153
Introducción 153
Poniendolo todo junto 154
Notas finales 154
Capítulo 29: Redes 156
Observaciones 156
Modo sin cabeza en la unidad 156
Examples 157
Creando un servidor, un cliente, y enviando un mensaje 157
La Clase que estamos utilizando para serializar 157
Creando un servidor 157
El cliente 159
Capítulo 30: ScriptableObject 161
Observaciones 161
ScriptableObjects con AssetBundles 161
Examples 161
Introducción 161
Creación de activos ScriptableObject 161
Crear instancias de ScriptableObject a través de código 162
ScriptableObjects se serializan en el editor incluso en PlayMode 162
Encuentra ScriptableObjects existentes durante el tiempo de ejecución 163
Capítulo 31: Singletons en la unidad 164
Observaciones 164
Otras lecturas 164
Examples 165
Implementación utilizando RuntimeInitializeOnLoadMethodAttribute 165
Un sencillo Singleton MonoBehaviour en Unity C # 165
Unidad avanzada de Singleton 166
Implementación Singleton a través de clase base 169
Patrón Singleton utilizando el sistema Entity-Component de Unitys 170

Clase de Singleton basada en MonoBehaviour y ScriptableObject ... 171
Capítulo 32: Sistema de audio ... 176
Introducción ... 176
Examples ... 176
Clase de audio - Reproducir audio ... 176
Capítulo 33: Sistema de entrada ... 177
Examples ... 177
Leyendo la lectura de teclas y la diferencia entre GetKey, GetKeyDown y GetKeyUp ... 177
Sensor de acelerómetro de lectura (básico) ... 178
Lea el sensor del acelerómetro (avance) ... 179
Sensor de acelerómetro de lectura (precisión) ... 179
Haga clic en el botón del mouse (izquierda, central, derecha) Clics ... 180
Capítulo 34: Sistema de interfaz de usuario (UI) ... 183
Examples ... 183
Suscripción al evento en código ... 183
Añadiendo oyentes del mouse ... 183
Capítulo 35: Sistema de interfaz de usuario gráfico de modo inmediato (IMGUI) ... 185
Sintaxis ... 185
Examples ... 185
GUILayout ... 185
Capítulo 36: Tienda de activos ... 186
Examples ... 186
Accediendo a la Tienda de Activos ... 186
Compra de activos ... 186
Importando activos ... 187
Publicación de Activos ... 187
Confirmar el número de factura de una compra ... 188
Capítulo 37: Transforma ... 189
Sintaxis ... 189
Examples ... 189
Visión general ... 189
Padres e hijos ... 190

Capítulo 38: Unity Profiler ... 192
Observaciones ... 192
Usando Profiler en diferentes dispositivos ... 192
Androide ... 192
iOS ... 193
Examples ... 193
Marcador de perfiles ... 193
Usando la Clase Profiler ... 193
Capítulo 39: Usando el control de fuente Git con Unity ... 195
Examples ... 195
Usando Git Large File Storage (LFS) con Unity ... 195
Prefacio ... 195
Instalación de Git y Git-LFS ... 195
Opción 1: usar una aplicación Git GUI ... 195
Opción 2: instalar Git y Git-LFS ... 196
Configurando Git Large File Storage en tu proyecto ... 196
Configuración de un repositorio Git para Unity ... 196
Unidad ignorar carpetas ... 196
Configuraciones del Proyecto Unity ... 197
Configuración adicional ... 197
Combinación de escenas y prefabricados ... 198
Capítulo 40: Vector3 ... 199
Introducción ... 199
Sintaxis ... 199
Examples ... 199
Valores estáticos ... 199
Vector3.zero y Vector3.one ... 199
Direcciones estáticas ... 200
Índice ... 202
Creando un Vector3 ... 202

Constructores 202
Convertir desde un Vector2 o Vector4 203
Aplicando movimiento 203
Lerp y LerpUnclamped 204
MoveTowards 205
SmoothDamp 206
Creditos 209

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: unity3d

It is an unofficial and free unity3d ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official unity3d.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con unity3d

Observaciones

Unity proporciona un entorno de desarrollo de juegos multiplataforma para desarrolladores. Los desarrolladores pueden usar UnityScript basado en lenguaje C # y / o JavaScript para la programación del juego. Las plataformas de despliegue de destino se pueden cambiar fácilmente en el editor. Todo el código del juego principal permanece igual, excepto algunas características dependientes de la plataforma. Una lista de todas las versiones y descargas correspondientes y notas de la versión se puede encontrar aquí: https://unity3d.com/get-unity/download/archive .

Versiones

Versión	Fecha de lanzamiento
Unidad 2017.1.0	2017-07-10
5.6.2	2017-06-21
5.6.1	2017-05-11
5.6.0	2017-03-31
5.5.3	2017-03-31
5.5.2	2017-02-24
5.5.1	2017-01-24
5.5	2016-11-30
5.4.3	2016-11-17
5.4.2	2016-10-21
5.4.1	2016-09-08
5.4.0	2016-07-28
5.3.6	2016-07-20
5.3.5	2016-05-20
5.3.4	2016-03-15
5.3.3	2016-02-23

Versión	Fecha de lanzamiento
5.3.2	2016-01-28
5.3.1	2015-12-18
5.3.0	2015-12-08
5.2.5	2016-06-01
5.2.4	2015-12-16
5.2.3	2015-11-19
5.2.2	2015-10-21
5.2.1	2015-09-22
5.2.0	2015-09-08
5.1.5	2015-06-07
5.1.4	2015-10-06
5.1.3	2015-08-24
5.1.2	2015-07-16
5.1.1	2015-06-18
5.1.0	2015-06-09
5.0.4	2015-07-06
5.0.3	2015-06-09
5.0.2	2015-05-13
5.0.1	2015-04-01
5.0.0	2015-03-03
4.7.2	2016-05-31
4.7.1	2016-02-25
4.7.0	2015-12-17
4.6.9	2015-10-15
4.6.8	2015-08-26

Versión	Fecha de lanzamiento
4.6.7	2015-07-01
4.6.6	2015-06-08
4.6.5	2015-04-30
4.6.4	2015-03-26
4.6.3	2015-02-19
4.6.2	2015-01-29
4.6.1	2014-12-09
4.6.0	2014-11-25
4.5.5	2014-10-13
4.5.4	2014-09-11
4.5.3	2014-08-12
4.5.2	2014-07-10
4.5.1	2014-06-12
4.5.0	2014-05-27
4.3.4	2014-01-29
4.3.3	2014-01-13
4.3.2	2013-12-18
4.3.1	2013-11-28
4.3.0	2013-11-12
4.2.2	2013-10-10
4.2.1	2013-09-05
4.2.0	2013-07-22
4.1.5	2013-06-08
4.1.4	2013-06-06
4.1.3	2013-05-23

Versión	Fecha de lanzamiento
4.1.2	2013-03-26
4.1.0	2013-03-13
4.0.1	2013-01-12
4.0.0	2012-11-13
3.5.7	2012-12-14
3.5.6	2012-09-27
3.5.5	2012-08-08
3.5.4	2012-07-20
3.5.3	2012-06-30
3.5.2	2012-05-15
3.5.1	2012-04-12
3.5.0	2012-02-14
3.4.2	2011-10-26
3.4.1	2011-09-20
3.4.0	2011-07-26

Examples

Instalación o configuración

Visión general

Unity se ejecuta en Windows y Mac. También hay una versión alfa de Linux disponible.

Hay 4 planes de pago diferentes para Unity:

1. **Personal** - Gratis *(ver abajo)*
2. **Más** - $ 35 USD por mes por asiento *(ver más abajo)*
3. **Pro** - $ 125 USD por mes por asiento: después de suscribirte al plan Pro durante 24 meses consecutivos, tienes la opción de dejar de suscribirte y conservar la versión que tienes.
4. **Empresa** - Póngase en contacto con Unity para más información.

*De acuerdo con el EULA: las compañías o entidades incorporadas que tuvieron un volumen de negocios superior a US $ 100,000 en su último año fiscal deben usar **Unity Plus** (o una licencia más alta); más de US $ 200,000 deben usar **Unity Pro** (o Enterprise).*

Instalación

1. Descarga el asistente de descarga de Unity .
2. Ejecute el asistente y elija los módulos que desea descargar e instalar, como el editor de Unity, el IDE de MonoDevelop, la documentación y los módulos de creación de plataforma deseados.

Si tiene una versión anterior, puede actualizar a la última versión estable .

Si desea instalar el asistente de descarga de Unity sin Unity, puede obtener los **instaladores de componentes** de las notas de la versión de Unity 5.5.1 .

Instalando múltiples versiones de Unity

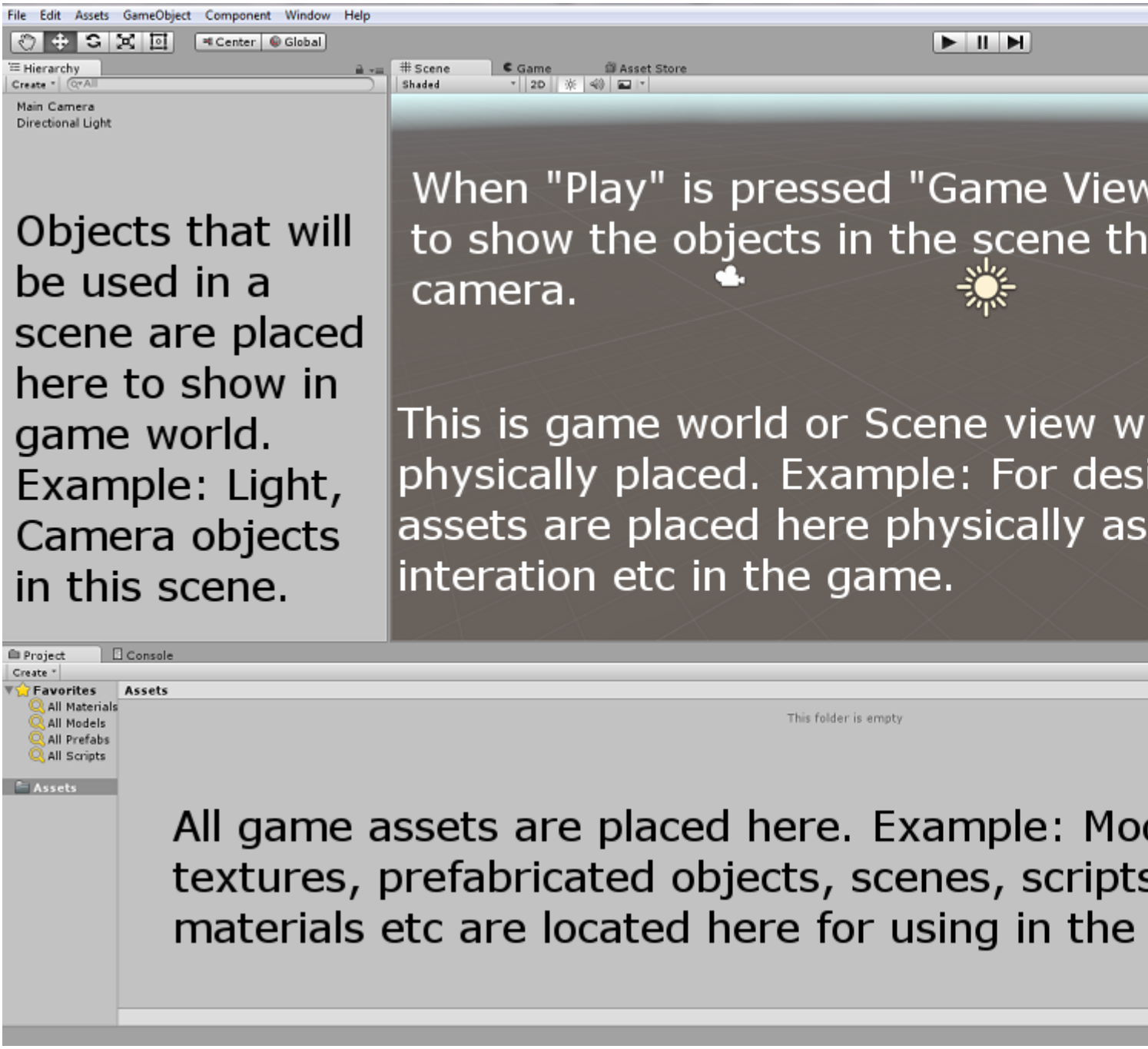
A menudo es necesario instalar varias versiones de Unity al mismo tiempo. Para hacerlo:

- En Windows, cambie el directorio de instalación predeterminado a una carpeta vacía que haya creado anteriormente, como `Unity 5.3.1f1` .
- En Mac, el instalador siempre se instalará en `/Applications/Unity` . Cambie el nombre de esta carpeta para su instalación existente (por ejemplo, a `/Applications/Unity5.3.1f1`) antes de ejecutar el instalador para la versión diferente.
- Puede mantener presionada la `tecla Alt` cuando inicie Unity para forzarla a que le permita elegir un proyecto para abrir. De lo contrario, el último proyecto cargado intentará cargar (si está disponible) y puede pedirle que actualice un proyecto que no desea que se actualice.

Editor y código básico.

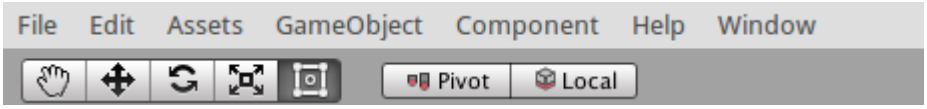
Diseño

El editor básico de Unity se verá a continuación. Las funcionalidades básicas de algunas ventanas / pestañas predeterminadas se describen en la imagen.



Diseño de Linux

Hay una pequeña diferencia en el diseño del menú de la versión de Linux, como la captura de pantalla a continuación,



Uso básico

Cree un `GameObject` vacío `GameObject` clic derecho en la ventana Jerarquía y seleccione `Create Empty` . Cree un nuevo script haciendo clic con el botón derecho en la ventana Proyecto y seleccione `Create` > `C# Script` . Renombrarlo según sea necesario.

Cuando se selecciona el `GameObject` vacío en la ventana Jerarquía, arrastre y suelte el script recién

creado en la ventana del Inspector. Ahora el script se adjunta al objeto en la ventana Jerarquía. Abra el script con el IDE de MonoDevelop predeterminado o su preferencia.

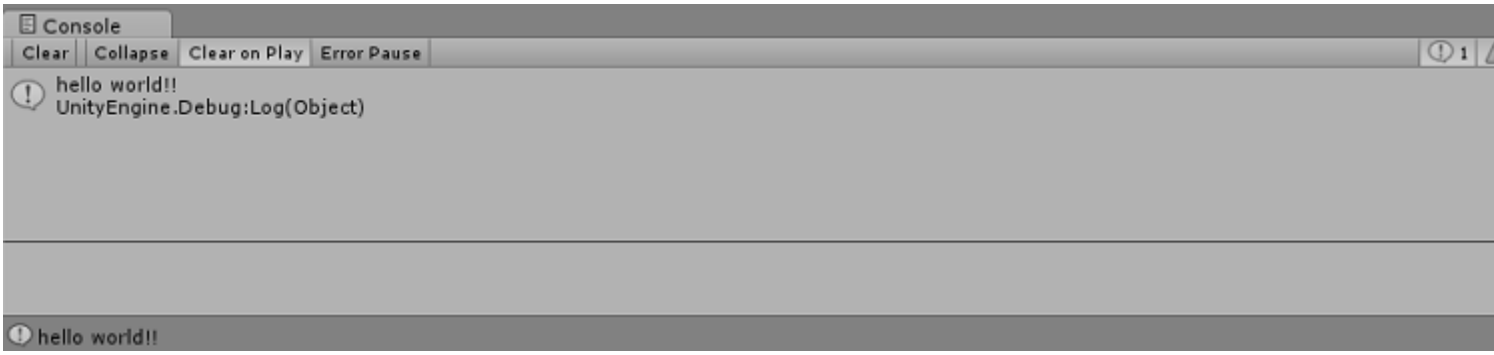
Scripting básico

El código básico se verá como a continuación, excepto la línea `Debug.Log("hello world!!");` .

```
using UnityEngine;
using System.Collections;

public class BasicCode : MonoBehaviour {

    // Use this for initialization
    void Start () {
        Debug.Log("hello world!!");
    }

    // Update is called once per frame
    void Update () {

    }
}
```

Añade la línea `Debug.Log("hello world!!");` en el método `void Start()` . Guarde el script y vuelva al editor. Ejecútalo presionando **Play** en la parte superior del editor.

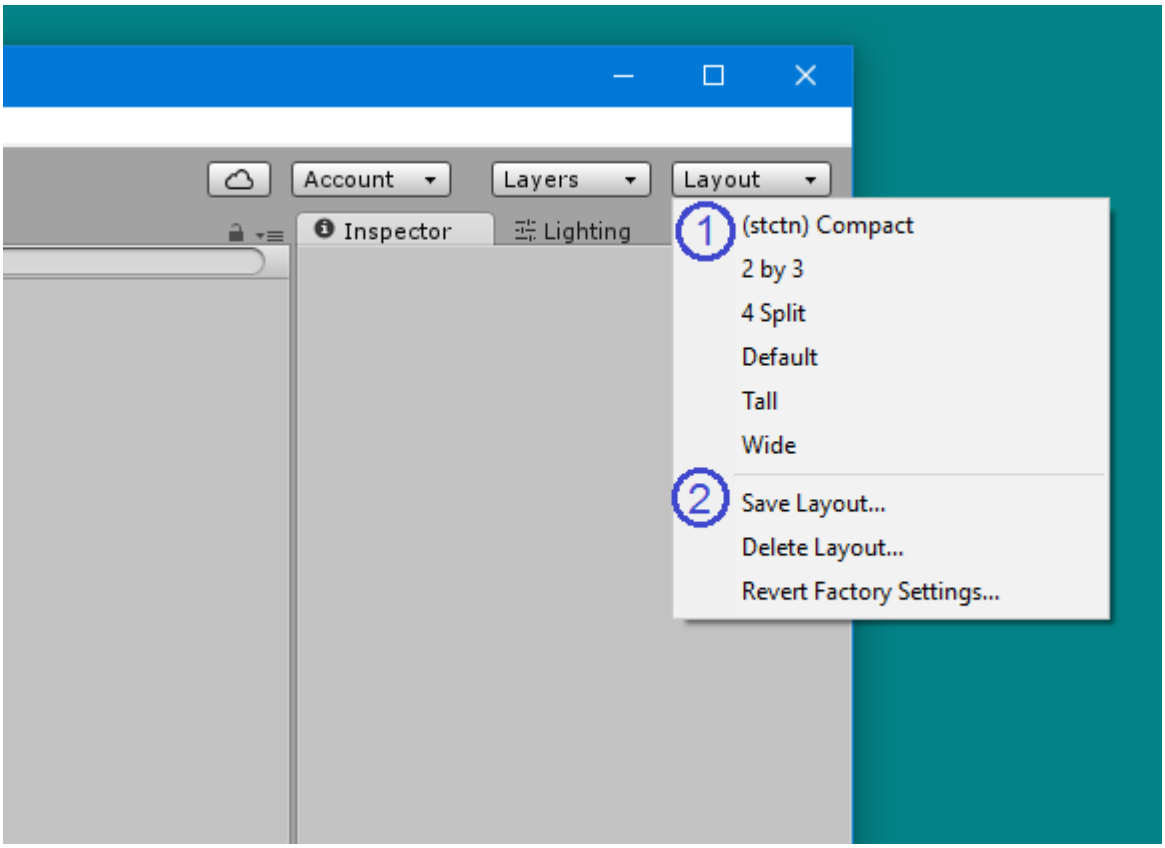
El resultado debe ser como abajo en la ventana de la consola:



Diseños de editor

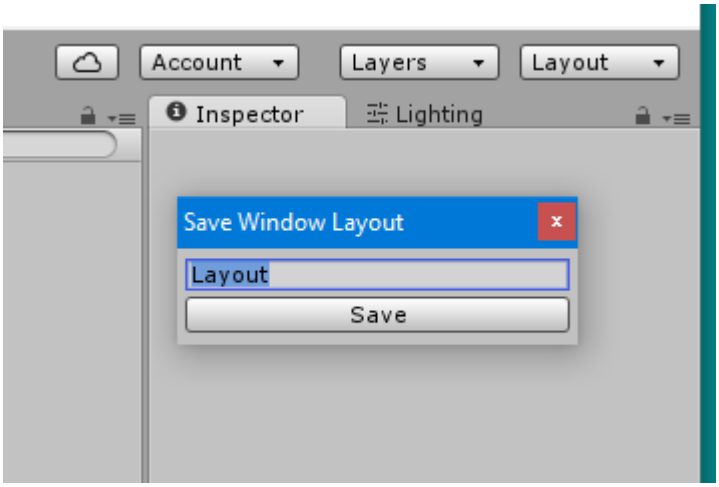
Puede guardar el diseño de sus pestañas y ventanas para estandarizar su entorno de trabajo.

El menú de diseños se puede encontrar en la esquina superior derecha de Unity Editor:

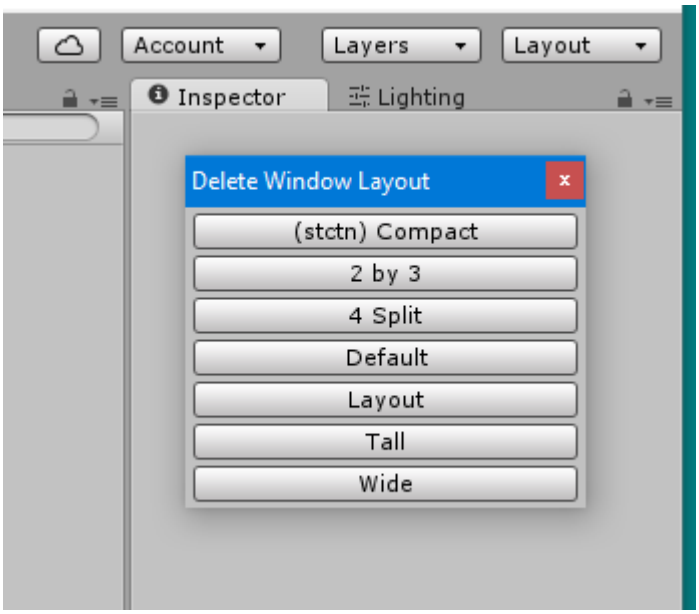


Unity se envía con 5 diseños predeterminados (2 por 3, 4 Dividir, Predeterminado, Alto, Ancho) *(marcado con 1)* . En la imagen de arriba, además de los diseños predeterminados, también hay un diseño personalizado en la parte superior.

Puede agregar sus propios diseños haciendo clic en el botón **"Guardar diseño ..."** en el menú *(marcado con 2)* :



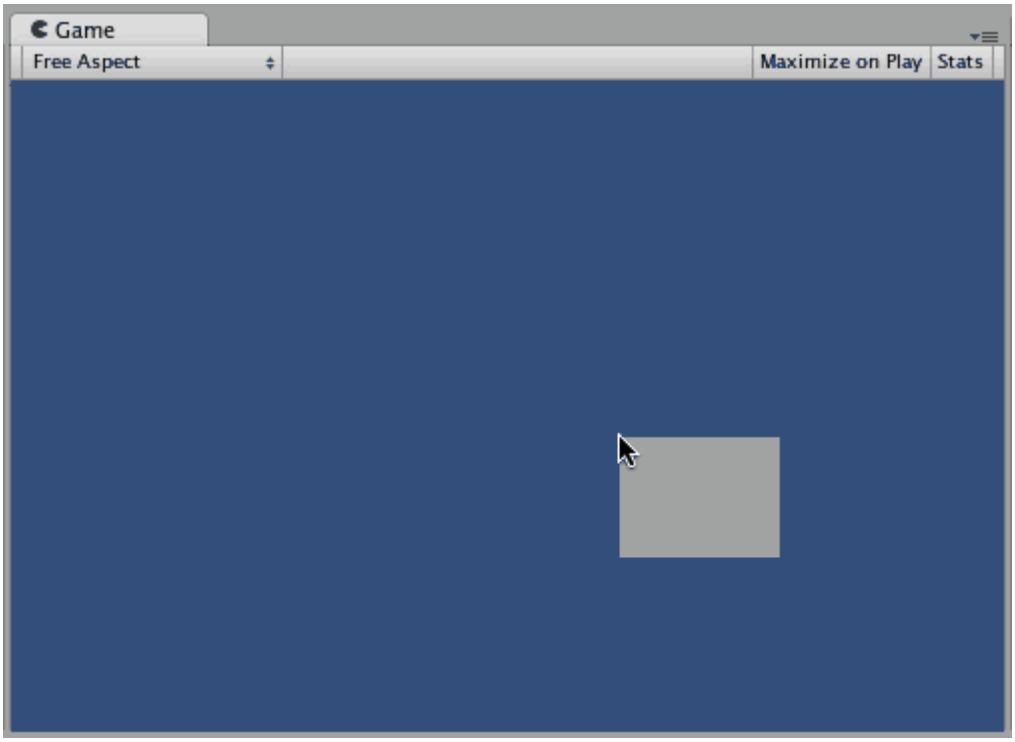
También puede eliminar cualquier diseño haciendo clic en el botón **"Eliminar diseño ..."** en el menú *(marcado con 2)* :



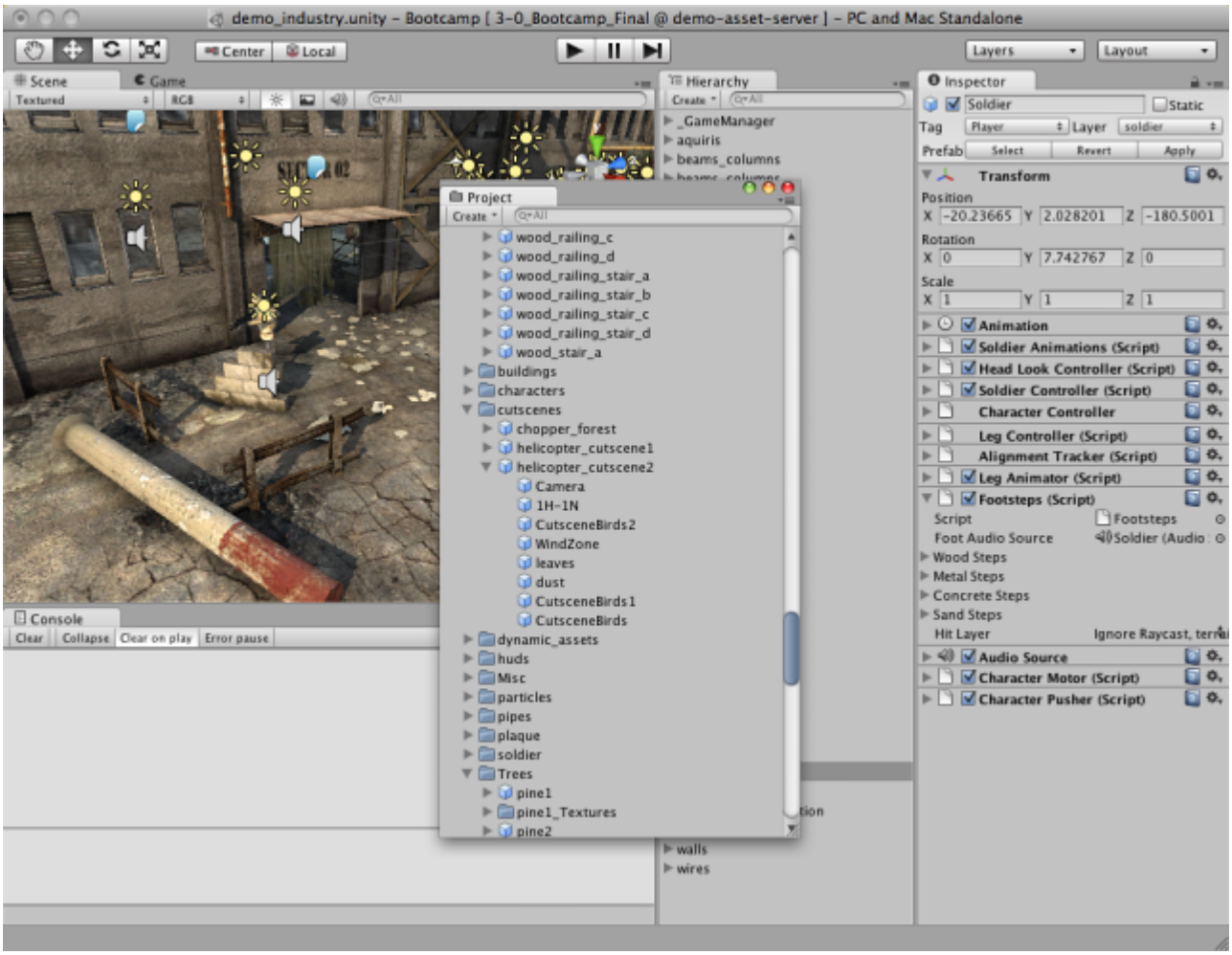
El botón **"Revertir configuración de fábrica ..."** elimina todos los diseños personalizados y restaura los diseños predeterminados *(marcados con 2)* .

Personalizando tu espacio de trabajo

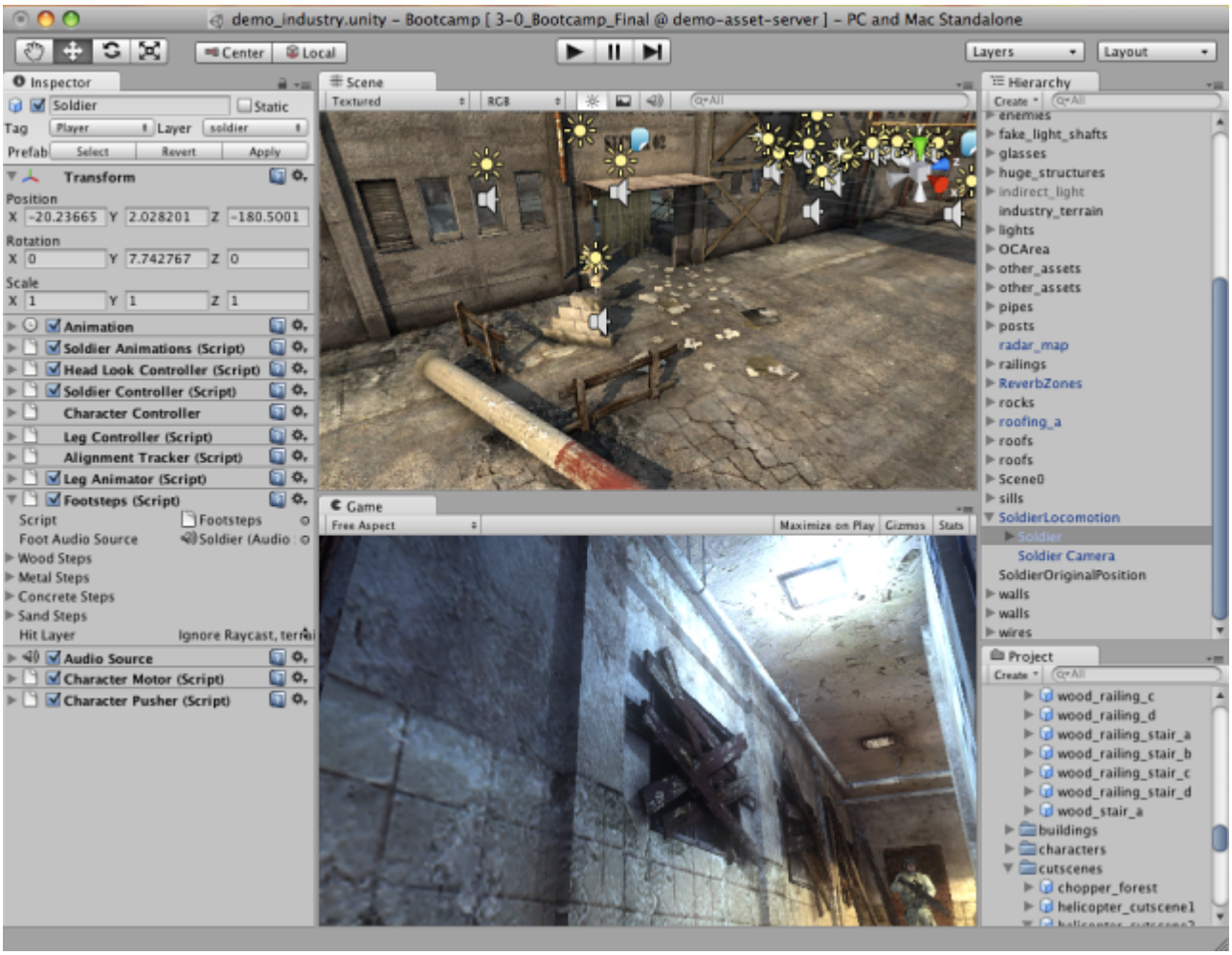
Puede personalizar su Diseño de Vistas haciendo clic y arrastrando la pestaña de cualquier Vista a una de varias ubicaciones. Al soltar una pestaña en el área de pestañas de una ventana existente, se agregará la pestaña junto a las pestañas existentes. Alternativamente, al soltar una pestaña en cualquier zona de acoplamiento se agregará la vista en una nueva ventana.



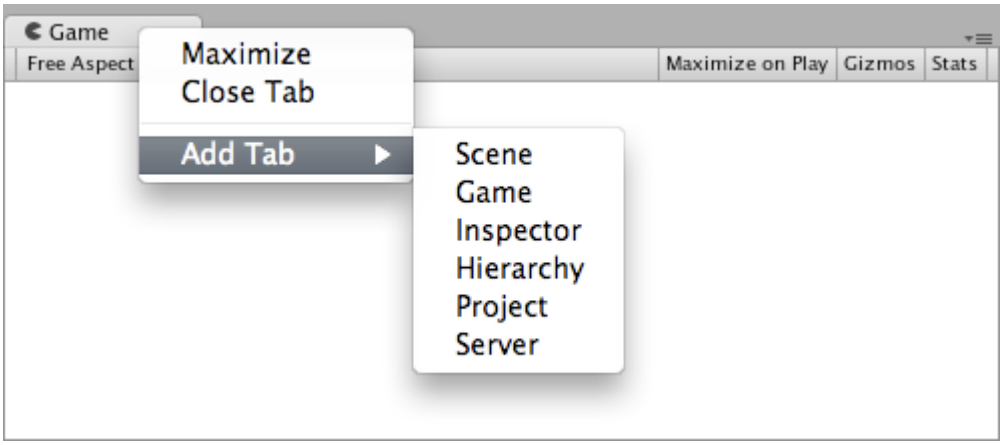
Las pestañas también pueden separarse de la ventana principal del editor y organizarse en su propio editor flotante de Windows. Las ventanas flotantes pueden contener disposiciones de Vistas y pestañas como la ventana del Editor principal.



Cuando haya creado un diseño de editor, puede guardar el diseño y restaurarlo en cualquier momento. Consulte este ejemplo para diseños de editor .



En cualquier momento, puede hacer clic derecho en la pestaña de cualquier vista para ver opciones adicionales como Maximizar o agregar una nueva pestaña a la misma ventana.



Lea Empezando con unity3d en línea: https://riptutorial.com/es/unity3d/topic/846/empezando-con-unity3d

Capítulo 2: Agrupación de objetos

Examples

Conjunto de objetos

A veces, cuando creas un juego necesitas crear y destruir muchos objetos del mismo tipo una y otra vez. Simplemente puede hacer esto haciendo una prefabulación y creando una instancia / destruyéndolo cuando lo necesite, sin embargo, hacerlo es ineficiente y puede ralentizar su juego.

Una forma de solucionar este problema es la agrupación de objetos. Básicamente, lo que esto significa es que tiene un conjunto (con o sin límite a la cantidad) de objetos que va a reutilizar siempre que pueda para evitar la creación de instancias innecesarias o la destrucción.

A continuación se muestra un ejemplo de un grupo de objetos simple

```
public class ObjectPool : MonoBehaviour
{
    public GameObject prefab;
    public int amount = 0;
    public bool populateOnStart = true;
    public bool growOverAmount = true;

    private List<GameObject> pool = new List<GameObject>();

    void Start()
    {
        if (populateOnStart && prefab != null && amount > 0)
        {
            for (int i = 0; i < amount; i++)
            {
                var instance = Instantiate(Prefab);
                instance.SetActive(false);
                pool.Add(instance);
            }
        }
    }

    public GameObject Instantiate (Vector3 position, Quaternion rotation)
    {
        foreach (var item in pool)
        {
            if (!item.activeInHierarchy)
            {
                item.transform.position = position;
                item.transform.rotation = rotation;
                item.SetActive( true );
                return item;
            }
        }

        if (growOverAmount)
        {
            var instance = (GameObject)Instantiate(prefab, position, rotation);
```

```
            pool.Add(instance);
            return instance;
        }

        return null;
    }
}
```

Repasemos primero las variables

```
public GameObject prefab;
public int amount = 0;
public bool populateOnStart = true;
public bool growOverAmount = true;

private List<GameObject> pool = new List<GameObject>();
```

- `GameObject prefab` : esta es la prefab que el grupo de objetos usará para instanciar nuevos objetos en el grupo.
- `int amount` : Esta es la cantidad máxima de elementos que pueden estar en la agrupación. Si desea crear una instancia de otro elemento y la agrupación ya ha alcanzado su límite, se utilizará otro elemento de la agrupación.
- `bool populateOnStart` : puede elegir rellenar el grupo al inicio o no. Al hacerlo, se llenará el grupo con instancias de la prefab para que la primera vez que llame a `Instantiate` , obtenga un objeto ya existente.
- `bool growOverAmount` : establecer esto en true permite que la agrupación crezca siempre que se solicite más de la cantidad en un determinado período de tiempo. No siempre es capaz de predecir con precisión la cantidad de elementos para colocar en su grupo, por lo que esto agregará más a su grupo cuando sea necesario.
- `List<GameObject> pool` : este es el grupo, el lugar donde se almacenan todos los objetos creados / destruidos.

Ahora vamos a ver la función de `Start`

```
void Start()
{
    if (populateOnStart && prefab != null && amount > 0)
    {
        for (int i = 0; i < amount; i++)
        {
            var instance = Instantiate(Prefab);
            instance.SetActive(false);
            pool.Add(instance);
        }
    }
}
```

En la función de inicio, verificamos si debemos rellenar la lista en el inicio y hacerlo si se ha establecido la `prefab` y la cantidad es mayor que 0 (de lo contrario, estaríamos creando de forma indefinida).

Esto es solo un simple para crear un bucle de objetos nuevos y ponerlos en el grupo. Una cosa a

la que hay que prestar atención es que configuramos todos los casos como inactivos. De esta manera aún no están visibles en el juego.

A continuación, está la función de `Instantiate` , que es donde ocurre la mayor parte de la magia.

```
public GameObject Instantiate (Vector3 position, Quaternion rotation)
{
    foreach (var item in pool)
    {
        if (!item.activeInHierarchy)
        {
            item.transform.position = position;
            item.transform.rotation = rotation;
            item.SetActive(true);
            return item;
        }
    }

    if (growOverAmount)
    {
        var instance = (GameObject)Instantiate(prefab, position, rotation);
        pool.Add(instance);
        return instance;
    }

    return null;
}
```

La función de `Instantiate` parece a la función de `Instantiate` propia de Unity, excepto que la prefabulación ya se ha proporcionado anteriormente como miembro de la clase.

El primer paso de la función de `Instantiate` es verificar si hay un objeto inactivo en la agrupación en este momento. Esto significa que podemos reutilizar ese objeto y devolvérselo al solicitante. Si hay un objeto inactivo en el grupo, establecemos la posición y la rotación, lo configuramos para que esté activo (de lo contrario, podría reutilizarse por accidente si olvida activarlo) y se lo devolverá al solicitante.

El segundo paso solo ocurre si no hay elementos inactivos en el grupo y se permite que el grupo crezca sobre la cantidad inicial. Lo que sucede es simple: se crea otra instancia del prefab y se agrega a la agrupación. Permitir el crecimiento de la piscina le ayuda a tener la cantidad correcta de objetos en la piscina.

El tercer "paso" solo ocurre si no hay elementos inactivos en el grupo y el grupo *no* puede crecer. Cuando esto suceda, el solicitante recibirá un GameObject nulo, lo que significa que no había nada disponible y debería manejarse adecuadamente para evitar `NullReferenceExceptions` .

¡Importante!

Para asegurarse de que sus artículos ponen de nuevo en la piscina que **no** deben destruir los objetos del juego. Lo único que debe hacer es establecerlos como inactivos y eso hará que estén disponibles para su reutilización a través de la piscina.

Conjunto de objetos simples

A continuación se muestra un ejemplo de un grupo de objetos que permite alquilar y devolver un tipo de objeto determinado. Para crear el grupo de objetos, se requiere un Func para la función de creación y una Acción para destruir el objeto para dar flexibilidad al usuario. Al solicitar un objeto cuando el grupo está vacío, se creará un nuevo objeto y al solicitar cuando el grupo tiene objetos, los objetos se eliminan del grupo y se devuelven.

Conjunto de objetos

```
public class ResourcePool<T> where T : class
{
    private readonly List<T> objectPool = new List<T>();
    private readonly Action<T> cleanUpAction;
    private readonly Func<T> createAction;

    public ResourcePool(Action<T> cleanUpAction, Func<T> createAction)
    {
        this.cleanUpAction = cleanUpAction;
        this.createAction = createAction;
    }

    public void Return(T resource)
    {
        this.objectPool.Add(resource);
    }

    private void PurgeSingleResource()
    {
        var resource = this.Rent();
        this.cleanUpAction(resource);
    }

    public void TrimResourcesBy(int count)
    {
        count = Math.Min(count, this.objectPool.Count);
        for (int i = 0; i < count; i++)
        {
            this.PurgeSingleResource();
        }
    }

    public T Rent()
    {
        int count = this.objectPool.Count;
        if (count == 0)
        {
            Debug.Log("Creating new object.");
            return this.createAction();
        }
        else
        {
            Debug.Log("Retrieving existing object.");
            T resource = this.objectPool[count-1];
            this.objectPool.RemoveAt(count-1);
            return resource;
        }
    }
}
```

Uso de la muestra

```
public class Test : MonoBehaviour
{
    private ResourcePool<GameObject> objectPool;

    [SerializeField]
    private GameObject enemyPrefab;

    void Start()
    {
        this.objectPool = new ResourcePool<GameObject>(Destroy,() =>
Instantiate(this.enemyPrefab) );
    }

    void Update()
    {
        // To get existing object or create new from pool
        var newEnemy = this.objectPool.Rent();
        // To return object to pool
        this.objectPool.Return(newEnemy);
        // In this example the message 'Creating new object' should only be seen on the frame
call
        // after that the same object in the pool will be returned.
    }
}
```

Otro grupo de objetos simples

Otro ejemplo: un arma que dispara balas.

El arma actúa como un grupo de objetos para las balas que crea.

```
public class Weapon : MonoBehaviour {

    // The Bullet prefab that the Weapon will create
    public Bullet bulletPrefab;

    // This List is our object pool, which starts out empty
    private List<Bullet> availableBullets = new List<Bullet>();

    // The Transform that will act as the Bullet starting position
    public Transform bulletInstantiationPoint;

    // To spawn a new Bullet, this method either grabs an available Bullet from the pool,
    // otherwise Instantiates a new Bullet
    public Bullet CreateBullet () {
        Bullet newBullet = null;

        // If a Bullet is available in the pool, take the first one and make it active
        if (availableBullets.Count > 0) {
            newBullet = availableBullets[availableBullets.Count - 1];

            // Remove the Bullet from the pool
            availableBullets.RemoveAt(availableBullets.Count - 1);

            // Set the Bullet's position and make its GameObject active
            newBullet.transform.position = bulletInstantiationPoint.position;
```

```
            newBullet.gameObject.SetActive(true);
        }
        // If no Bullets are available in the pool, Instantiate a new Bullet
        else {
            newBullet newObject = Instantiate(bulletPrefab, bulletInstantiationPoint.position,
Quaternion.identity);

            // Set the Bullet's Weapon so we know which pool to return to later on
            newBullet.weapon = this;
        }

        return newBullet;
    }

}

public class Bullet : MonoBehaviour {

    public Weapon weapon;

    // When Bullet collides with something, rather than Destroying it, we return it to the
pool
    public void ReturnToPool () {
        // Add Bullet to the pool
        weapon.availableBullets.Add(this);

        // Disable the Bullet's GameObject so it's hidden from view
        gameObject.SetActive(false);
    }

}
```

Lea Agrupación de objetos en línea: https://riptutorial.com/es/unity3d/topic/2276/agrupacion-de-objetos

Capítulo 3: Animación de la unidad

Examples

Animación básica para correr

Este código muestra un ejemplo simple de animación en Unity.

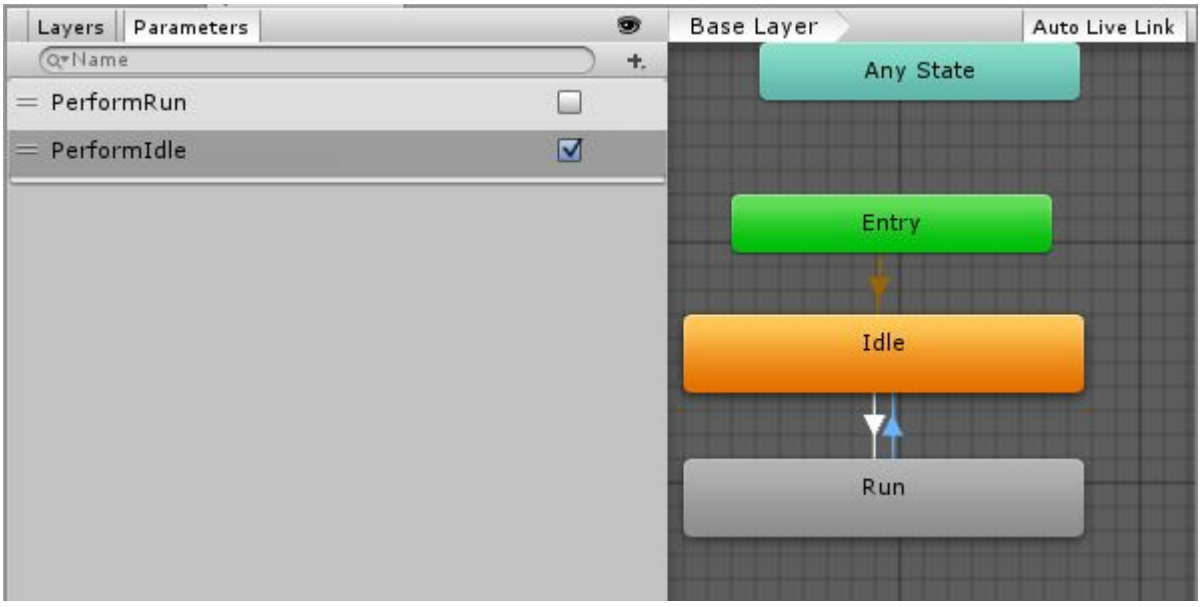
Para este ejemplo, deberías tener 2 clips de animación; Ejecutar y inactivo. Esas animaciones deben ser movimientos Stand-In-Place. Una vez que se seleccionan los clips de animación, crear un controlador de animador. Agregue este controlador al jugador o al objeto del juego que desea animar.

Abra la ventana del animador desde la opción de Windows. Arrastre los 2 clips de animación a la ventana del animador y se crearán 2 estados. Una vez creado, use la pestaña de parámetros de la izquierda para agregar 2 parámetros, ambos como bool. Nombre uno como "PerformRun" y otro como "PerformIdle". Establezca "PerformIdle" en true.

Realice transiciones desde el estado inactivo a Ejecutar y Ejecutar a inactivo (consulte la imagen). Haga clic en Inactivo-> Ejecutar transición y en la ventana del Inspector, deseleccione HasExit. Haz lo mismo para la otra transición. Para Inactivo-> Ejecutar transición, agregue una condición: PerformIdle. Para Ejecutar-> Inactivo, agregue una condición: PerformRun. Agregue el script C # dado abajo al objeto del juego. Debe ejecutarse con animación usando el botón Arriba y rotar con los botones Izquierda y Derecha.

```
using UnityEngine;
using System.Collections;

public class RootMotion : MonoBehaviour {

//Public Variables
[Header("Transform Variables")]
public float RunSpeed = 0.1f;
public float TurnSpeed = 6.0f;


Animator animator;

void Start()
{
    /**
    * Initialize the animator that is attached on the current game object i.e. on which you
will attach this script.
    */
    animator = GetComponent<Animator>();
}

void Update()
{
    /**
    * The Update() function will get the bool parameters from the animator state machine and
set the values provided by the user.
```

```
    * Here, I have only added animation for Run and Idle. When the Up key is pressed, Run
animation is played. When we let go, Idle is played.
    */

    if (Input.GetKey (KeyCode.UpArrow)) {
            animator.SetBool ("PerformRun", true);
            animator.SetBool ("PerformIdle", false);
        } else {
            animator.SetBool ("PerformRun", false);
            animator.SetBool ("PerformIdle", true);
        }
}

void OnAnimatorMove()
    {
        /**
         * OnAnimatorMove() function will shadow the "Apply Root Motion" on the animator. Your
game objects psoition will now be determined
         * using this fucntion.
         */
        if (Input.GetKey (KeyCode.UpArrow)){
            transform.Translate (Vector3.forward * RunSpeed);
            if (Input.GetKey (KeyCode.RightArrow)) {
                transform.Rotate (Vector3.up * Time.deltaTime * TurnSpeed);
            }
            else if (Input.GetKey (KeyCode.LeftArrow)) {
                transform.Rotate (-Vector3.up * Time.deltaTime * TurnSpeed);
            }
        }

    }



}
```

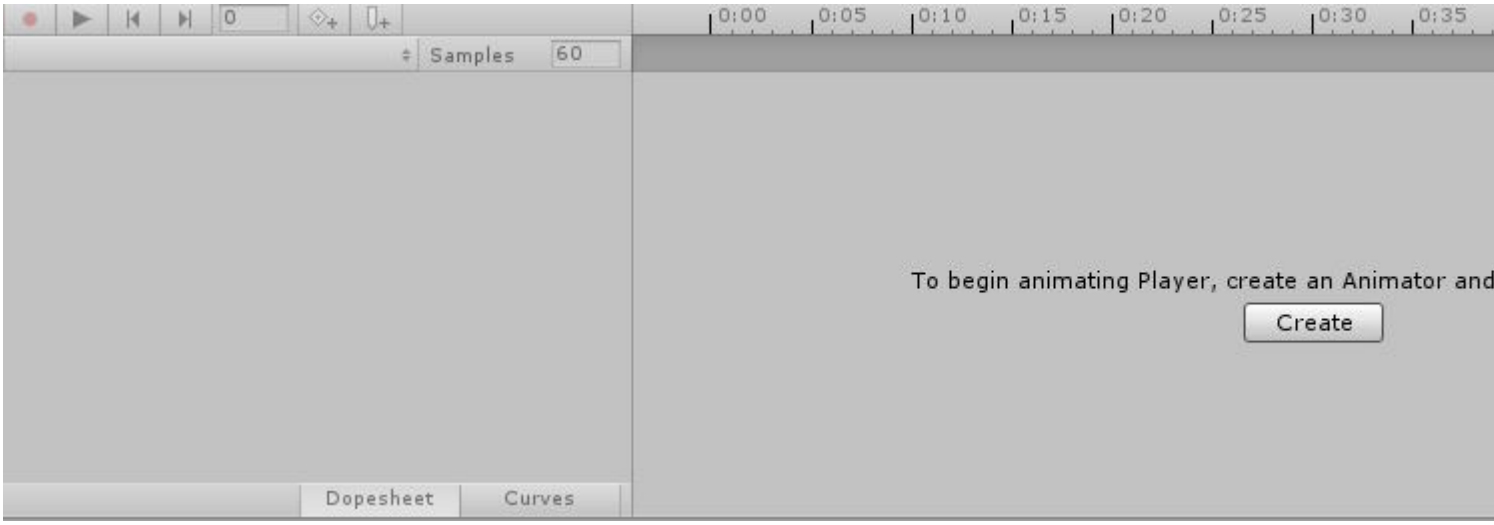


Creación y uso de clips de animación

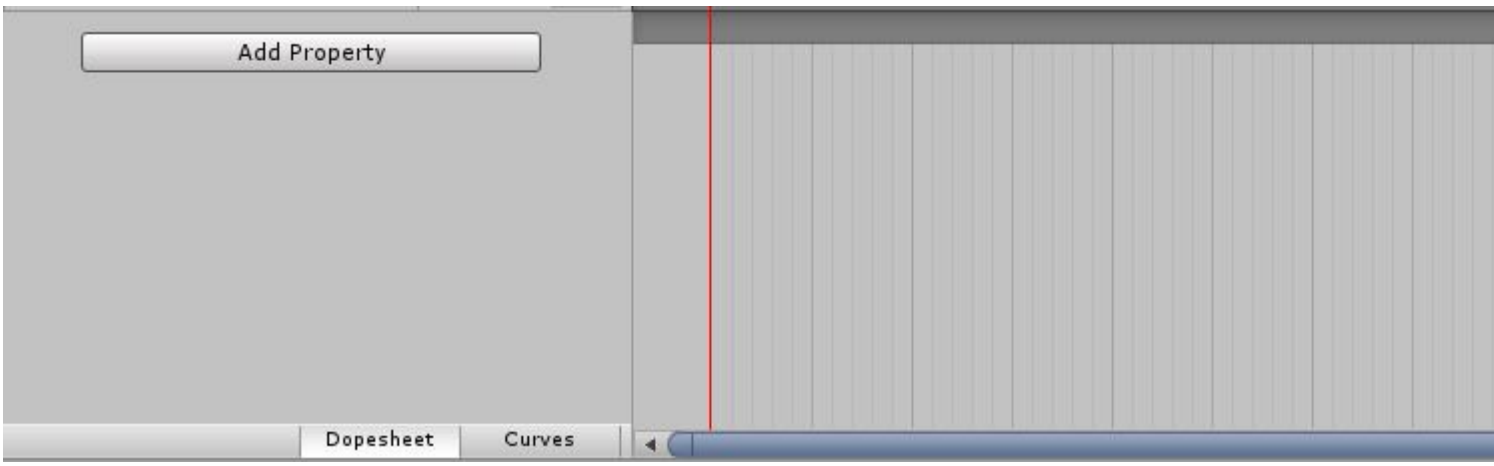
Este ejemplo mostrará cómo hacer y usar clips de animación para objetos de juego o jugadores.

Tenga en cuenta que los modelos utilizados en este ejemplo se descargan desde Unity Asset Store. El reproductor se descargó desde el siguiente enlace: https://www.assetstore.unity3d.com/en/#!/content/21874 .

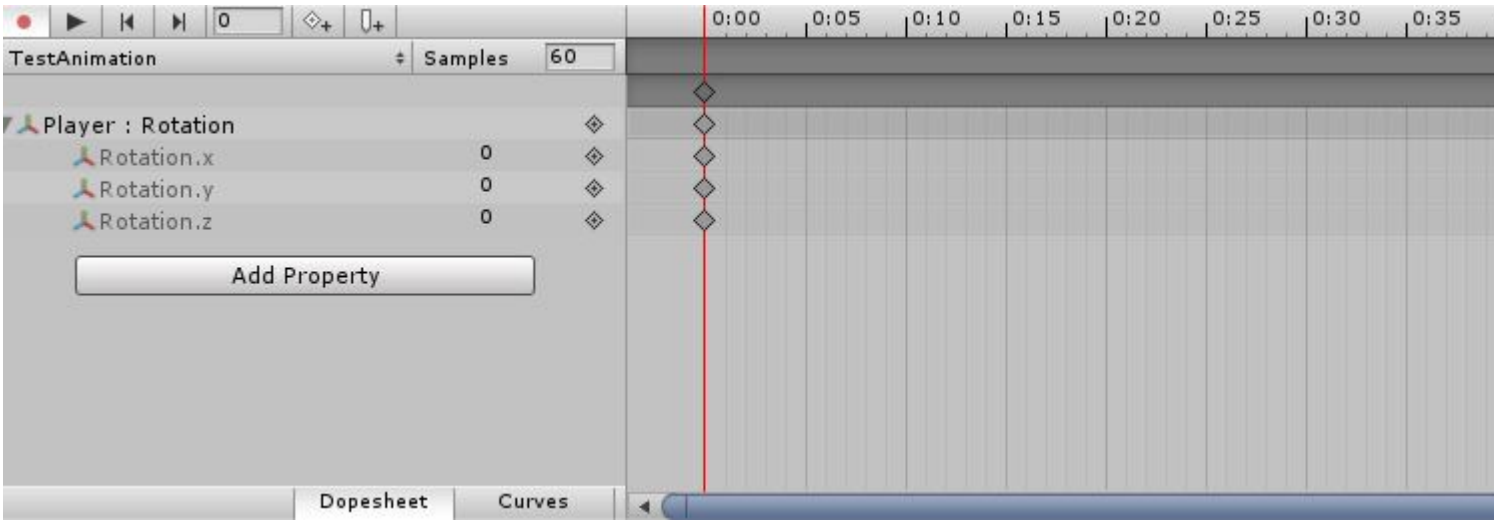
Para crear animaciones, primero abra la ventana de animación. Puede abrirlo haciendo clic en Ventana y Seleccionar animación o presione Ctrl + 6. Seleccione el objeto del juego al que desea aplicar el clip de animación, desde la ventana de jerarquía, y luego haga clic en el botón Crear en la ventana de animación.



Nombra tu animación (como IdlePlayer, SprintPlayer, DyingPlayer, etc.) y guárdala. Ahora, desde la ventana de animación, haga clic en el botón Agregar propiedad. Esto te permitirá cambiar la propiedad del objeto o jugador del juego con respecto al tiempo. Esto puede incluir Transformaciones como rotación, posición y escala y cualquier otra propiedad que esté asociada al objeto del juego, por ejemplo, Colisionador, Representador de malla, etc.



Para crear una animación en ejecución para el objeto del juego, necesitarás un modelo 3D humanoide. Puede descargar el modelo desde el enlace de arriba. Siga los pasos anteriores para crear una nueva animación. Agregue una propiedad de transformación y seleccione Rotación para una de las partes del personaje.



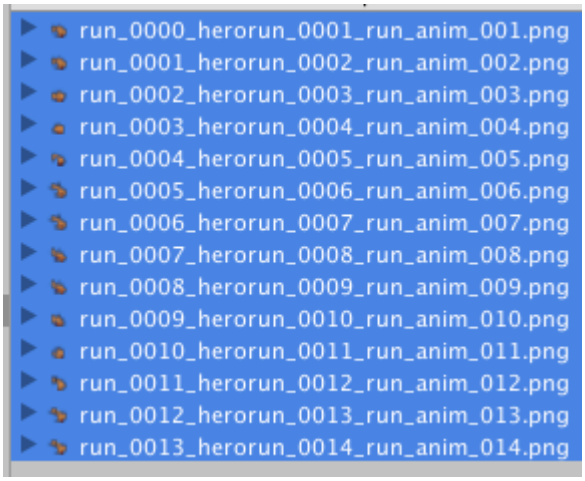
En este momento, su botón de Juego y los valores de Rotación en la propiedad del objeto del juego se habrían puesto rojos. Haga clic en la flecha desplegable para ver los valores de rotación X, Y y Z. El tiempo de animación predeterminado se establece en 1 segundo. Las animaciones utilizan fotogramas clave para interpolar entre valores. Para animar, agregue claves en diferentes puntos en el tiempo y cambie los valores de rotación desde la ventana del Inspector. Por ejemplo, el valor de rotación en el tiempo 0.0s puede ser 0.0. En el tiempo 0.5s, el valor puede ser 20.0 para X. En el tiempo 1.0s el valor puede ser 0.0. Podemos terminar nuestra animación en 1.0s.

La duración de la animación depende de las últimas teclas que agregue a la animación. Puede agregar más teclas para que la interpolación sea más suave.

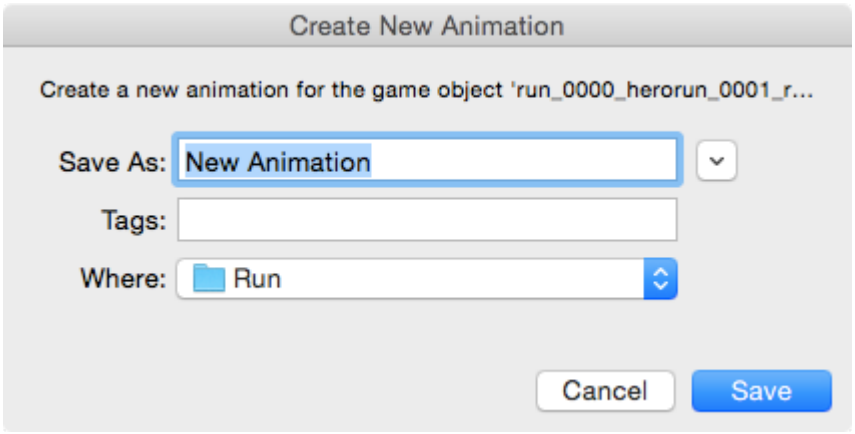
Animación 2D Sprite

La animación Sprite consiste en mostrar una secuencia existente de imágenes o marcos.

Primero importa una secuencia de imágenes a la carpeta de activos. Cree algunas imágenes desde cero o descargue algunas desde Asset Store. (Este ejemplo utiliza este activo gratis).

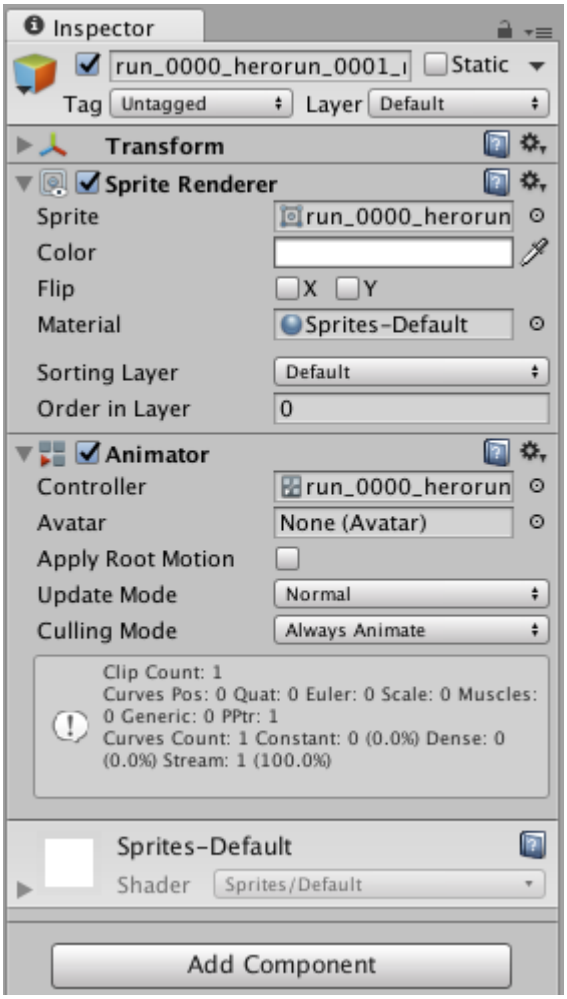


Arrastre cada imagen individual de una sola animación desde la carpeta de activos a la vista de escena. Unity mostrará un diálogo para nombrar el nuevo clip de animación.

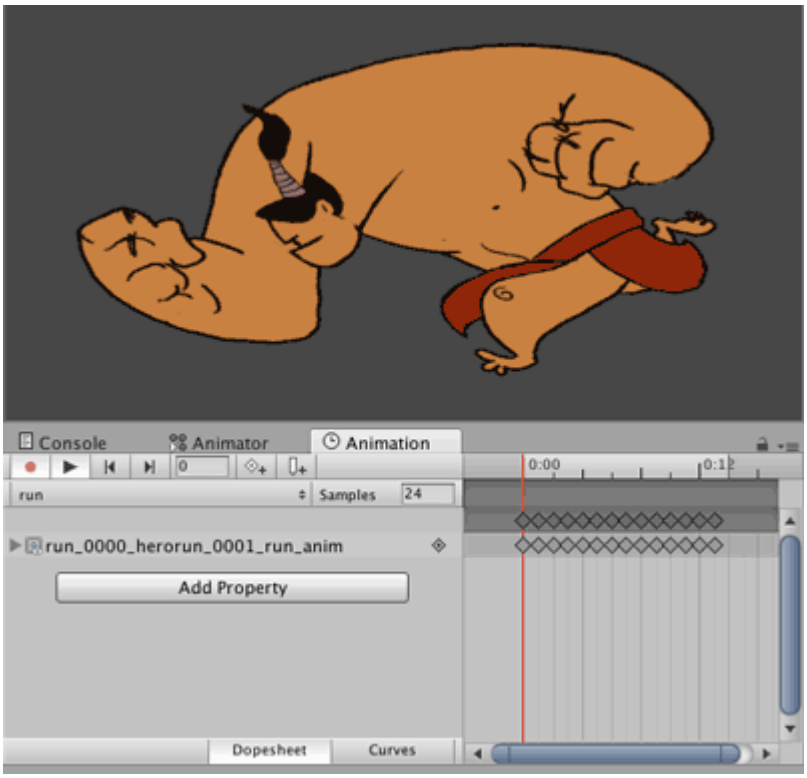


Este es un atajo útil para:

- creando nuevos objetos de juego
- asignando dos componentes (un Sprite Renderer y un Animator)
- creando controladores de animación (y vinculando el nuevo componente Animator a ellos)
- Creando clips de animación con los marcos seleccionados.



Previsualice la reproducción en la pestaña de animación haciendo clic en Reproducir:



Se puede usar el mismo método para crear nuevas animaciones para el mismo objeto de juego, y luego eliminar el nuevo objeto de juego y el controlador de animación. Agregue el nuevo clip de animación al controlador de animación de ese objeto de la misma manera que con la animación 3D.

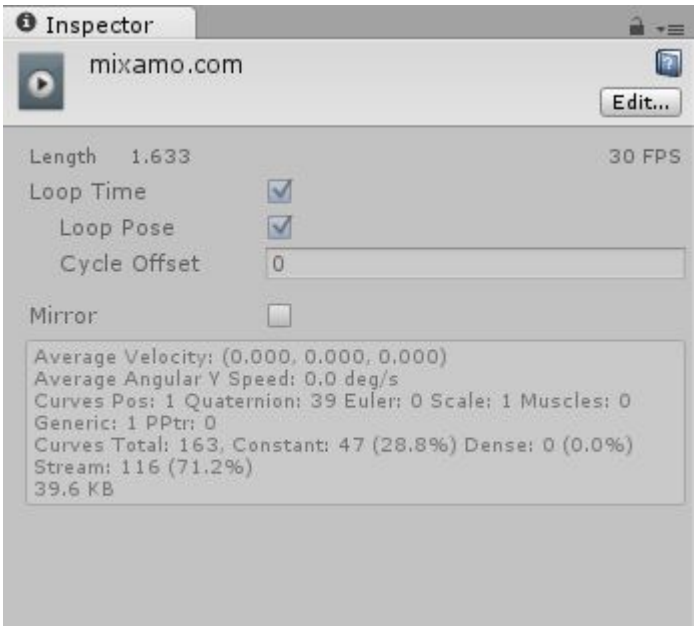
Curvas de animacion

Las curvas de animación le permiten cambiar un parámetro flotante mientras se reproduce la animación. Por ejemplo, si hay una animación de 60 segundos de duración y desea un valor / parámetro flotante, llámelo X, para variar a través de la animación (como en el tiempo de animación = 0.0s; X = 0.0, en el tiempo de animación = 30.0s; X = 1.0, en tiempo de animación = 60.0s; X = 0.0).

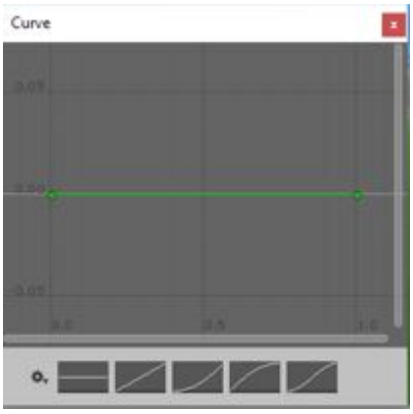
Una vez que tenga el valor flotante, puede usarlo para traducir, rotar, escalar o usarlo de cualquier otra forma.

Para mi ejemplo, mostraré un objeto de juego de jugador en ejecución. Cuando se reproduce la animación para correr, la velocidad de traducción del jugador debe aumentar a medida que avanza la animación. Cuando la animación llega a su fin, la velocidad de traducción debe disminuir.

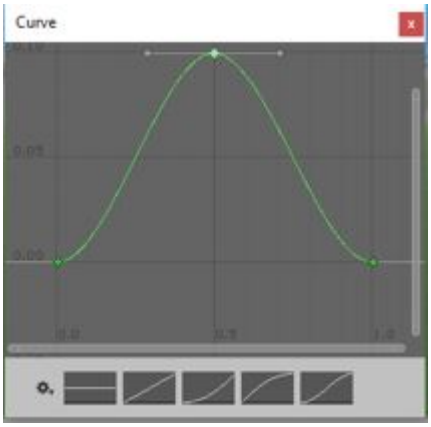
Tengo un clip de animación en ejecución creado. Seleccione el clip y luego en la ventana del inspector, haga clic en Editar.



Una vez allí, desplácese hasta Curvas. Haga clic en el signo + para agregar una curva. Nombra la curva, por ejemplo, ForwardRunCurve. Haga clic en la curva en miniatura a la derecha. Se abrirá una pequeña ventana con una curva predeterminada en ella.



Queremos una curva en forma parabólica donde sube y luego cae. Por defecto, hay 2 puntos en la línea. Puedes agregar más puntos haciendo doble clic en la curva. Arrastre los puntos para crear una forma similar a la siguiente.



En la ventana del animador, agregue el clip en ejecución. Además, agregue un parámetro flotante con el mismo nombre que la curva, es decir, ForwardRunCurve.

Cuando se reproduce la animación, el valor flotante cambiará de acuerdo con la curva. El siguiente código mostrará cómo usar el valor flotante:

```
using UnityEngine;
using System.Collections;

public class RunAnimation : MonoBehaviour {

Animator animator;
float curveValue;

void Start()
{
    animator = GetComponent<Animator>();
}

void Update()
{
    curveValue = animator.GetFloat("ForwardRunCurve");

    transform.Translate (Vector3.forward * curveValue);
}

}
```

La variable curveValue mantiene el valor de la curva (ForwardRunCruve) en un momento dado. Estamos utilizando ese valor para cambiar la velocidad de la traducción. Puedes adjuntar este script al objeto del juego del jugador.

Lea Animación de la unidad en línea: https://riptutorial.com/es/unity3d/topic/5448/animacion-de-la-unidad

Capítulo 4: API de CullingGroup

Observaciones

Dado que el uso de CullingGroups no siempre es muy sencillo, puede ser útil para encapsular la mayor parte de la lógica detrás de una clase de administrador.

A continuación se muestra un modelo de cómo podría operar un gerente de este tipo.

```
using UnityEngine;
using System;
public interface ICullingGroupManager
{
    int ReserveSphere();
    void ReleaseSphere(int sphereIndex);
    void SetPosition(int sphereIndex, Vector3 position);
    void SetRadius(int sphereIndex, float radius);
    void SetCullingEvent(int sphereIndex, Action<CullingGroupEvent> sphere);
}
```

La esencia de esto es que usted reserva una esfera de selección del administrador que devuelve el índice de la esfera reservada. A continuación, utiliza el índice dado para manipular su esfera reservada.

Examples

Recoger distancias de objetos

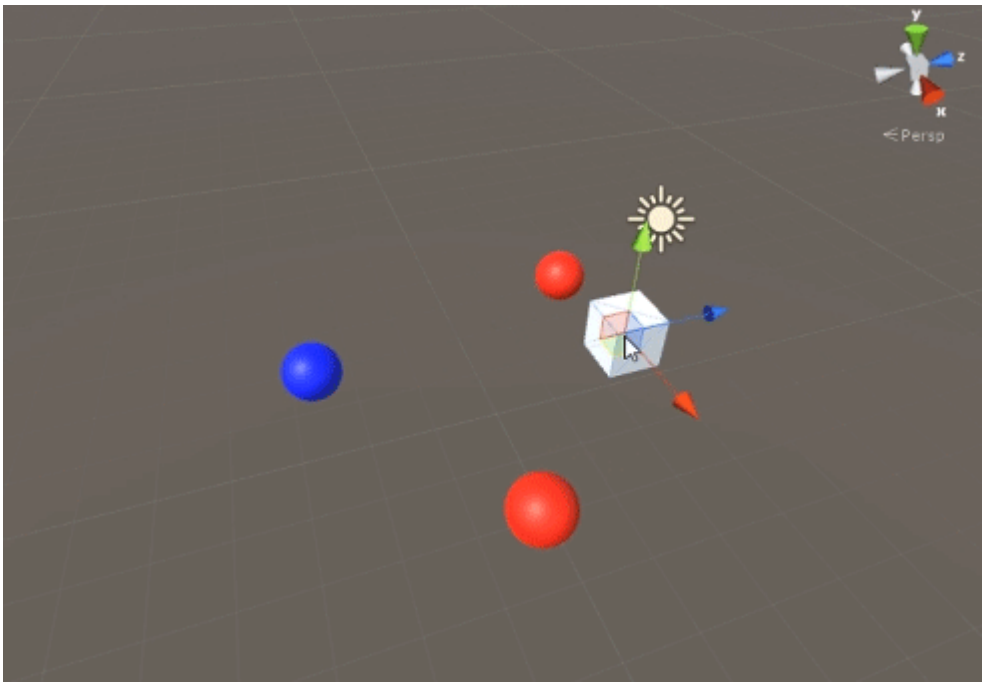
El siguiente ejemplo ilustra cómo usar CullingGroups para obtener notificaciones de acuerdo con el punto de referencia de la distancia.

> Esta secuencia de comandos se ha simplificado por motivos de brevedad y utiliza varios métodos pesados de rendimiento.

```
using UnityEngine;
using System.Linq;

public class CullingGroupBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;
    Transform[] meshTransforms;
    BoundingSphere[] cullingPoints;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();

        meshRenderers = FindObjectsOfType<MeshRenderer>()
                .Where((MeshRenderer m) => m.gameObject != this.gameObject)
```

```
                .ToArray();

        cullingPoints = new BoundingSphere[meshRenderers.Length];
        meshTransforms = new Transform[meshRenderers.Length];

        for (var i = 0; i < meshRenderers.Length; i++)
        {
            meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
            cullingPoints[i].position = meshTransforms[i].position;
            cullingPoints[i].radius = 4f;
        }

        localCullingGroup.onStateChanged = CullingEvent;
        localCullingGroup.SetBoundingSpheres(cullingPoints);
        localCullingGroup.SetBoundingDistances(new float[] { 0f, 5f });
        localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
        localCullingGroup.targetCamera = Camera.main;
    }

    void FixedUpdate()
    {
        localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
        for (var i = 0; i < meshTransforms.Length; i++)
        {
            cullingPoints[i].position = meshTransforms[i].position;
        }
    }

    void CullingEvent(CullingGroupEvent sphere)
    {
        Color newColor = Color.red;
        if (sphere.currentDistance == 1) newColor = Color.blue;
        if (sphere.currentDistance == 2) newColor = Color.white;
        meshRenderers[sphere.index].material.color = newColor;
    }

    void OnDisable()
    {
        localCullingGroup.Dispose();
    }
}
```

Agrega el script a un GameObject (en este caso un cubo) y pulsa Play. Todos los demás GameObject en escena cambian de color de acuerdo con su distancia al punto de referencia.



Visibilidad de objetos de selección

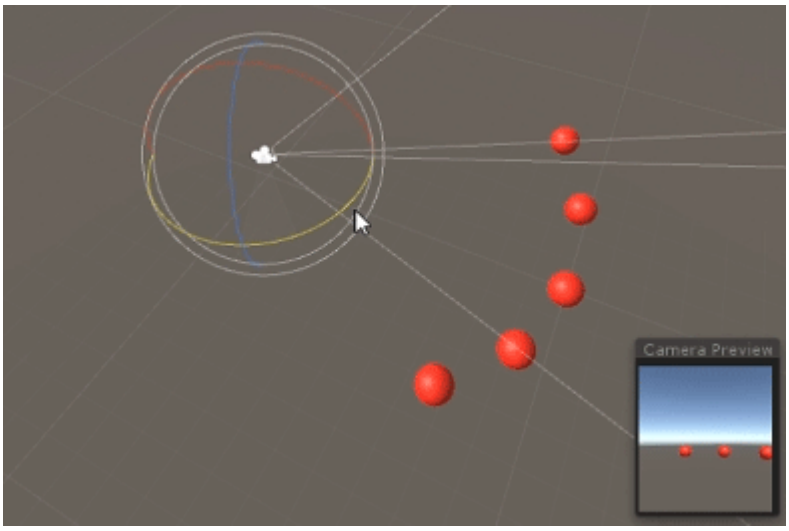
El siguiente script ilustra cómo recibir eventos según la visibilidad de una cámara establecida.

> Este script utiliza varios métodos de rendimiento pesado para la brevedad.

```
using UnityEngine;
using System.Linq;

public class CullingGroupCameraBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();

        meshRenderers = FindObjectsOfType<MeshRenderer>()
            .Where((MeshRenderer m) => m.gameObject != this.gameObject)
            .ToArray();

        BoundingSphere[] cullingPoints = new BoundingSphere[meshRenderers.Length];
        Transform[] meshTransforms = new Transform[meshRenderers.Length];

        for (var i = 0; i < meshRenderers.Length; i++)
        {
            meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
            cullingPoints[i].position = meshTransforms[i].position;
            cullingPoints[i].radius = 4f;
        }

        localCullingGroup.onStateChanged = CullingEvent;
        localCullingGroup.SetBoundingSpheres(cullingPoints);
        localCullingGroup.targetCamera = Camera.main;
    }
```

```
    void CullingEvent(CullingGroupEvent sphere)
    {
        meshRenderers[sphere.index].material.color = sphere.isVisible ? Color.red :
Color.white;
    }

    void OnDisable()
    {
        localCullingGroup.Dispose();
    }
}
```

Agrega el script a la escena y pulsa Play. Toda la geometría en escena cambiará de color en función de su visibilidad.



> Se puede lograr un efecto similar utilizando el método `MonoBehaviour.OnBecameVisible()` si el objeto tiene un componente `MeshRenderer` . Utilice CulingGroups cuando necesite seleccionar GameObjects vacíos, coordenadas `Vector3` , o cuando desee un método centralizado de seguimiento de visibilidades de objetos.

Distancias delimitadas

Puede agregar distancias de límite en la parte superior del radio del punto de selección. Son, de alguna manera, condiciones de activación adicionales fuera del radio principal de los puntos de selección, como "cerca", "lejos" o "muy lejos".

```
cullingGroup.SetBoundingDistances(new float[] { 0f, 10f, 100f});
```

> Las distancias de límite afectan solo cuando se usan con un punto de referencia de distancia. No tienen efecto durante el sacrificio de la cámara.

Visualizando distancias de límite.

Lo que inicialmente puede causar confusión es cómo se agregan las distancias de los límites en

la parte superior de los radios de la esfera.

Primero, el grupo de selección calcula el *área* de la esfera delimitadora y la distancia delimitadora. Las dos áreas se suman, y el resultado es el área de activación para la banda de distancia. El radio de esta área se puede utilizar para visualizar el campo de efecto de la distancia de delimitación.

```
float cullingPointArea = Mathf.PI * (cullingPointRadius * cullingPointRadius);
float boundingArea = Mathf.PI * (boundingDistance * boundingDistance);
float combinedRadius = Mathf.Sqrt((cullingPointArea + boundingArea) / Mathf.PI);
```

Lea API de CullingGroup en línea: https://riptutorial.com/es/unity3d/topic/4574/api-de-cullinggroup

Capítulo 5: Atributos

Sintaxis

- [AddComponentMenu (string menuName)]
- [AddComponentMenu (string menuName, int order)]
- [CanEditMultipleObjects]
- [ContextMenuItem (nombre de cadena, función de cadena)]
- [ContextMenu (nombre de cadena)]
- [CustomEditor (Type inspectedType)]
- [CustomEditor (Type inspectedType, bool editorForChildClasses)]
- [CustomPropertyDrawer (tipo de tipo)]
- [CustomPropertyDrawer (tipo de tipo, bool useForChildren)]
- [DisallowMultipleComponent]
- [DrawGizmo (GizmoType gizmo)]
- [DrawGizmo (GizmoType gizmo, Type drawnGizmoType)]
- [ExecuteInEditMode]
- [Encabezado (encabezado de cadena)]
- [HideInInspector]
- [InitializeOnLoad]
- [InitializeOnLoadMethod]
- [MenuItem (string itemName)]
- [MenuItem (string itemName, bool isValidateFunction)]
- [MenuItem (string itemName, bool isValidateFunction, int prioridad)]
- [Multilínea (líneas int)]
- [PreferenceItem (nombre de cadena)]
- [Rango (flotar min, flotar max)]
- [RequireComponent (tipo de tipo)]
- [RuntimeInitializeOnLoadMethod]
- [RuntimeInitializeOnLoadMethod (RuntimeInitializeLoadType loadType)]
- [SerializeField]
- [Espacio (altura de flotación)]
- [TextArea (int minLines, int maxLines)]
- [Tooltip (string tooltip)]

Observaciones

SerializeField

El sistema de serialización de Unity se puede usar para hacer lo siguiente:

- **Puede** serializar campos públicos no estáticos (de tipos serializables)
- **Puede** serializar campos no públicos no públicos marcados con el atributo [SerializeField]

- **No se pueden** serializar campos estáticos
- **No se pueden** serializar propiedades estáticas

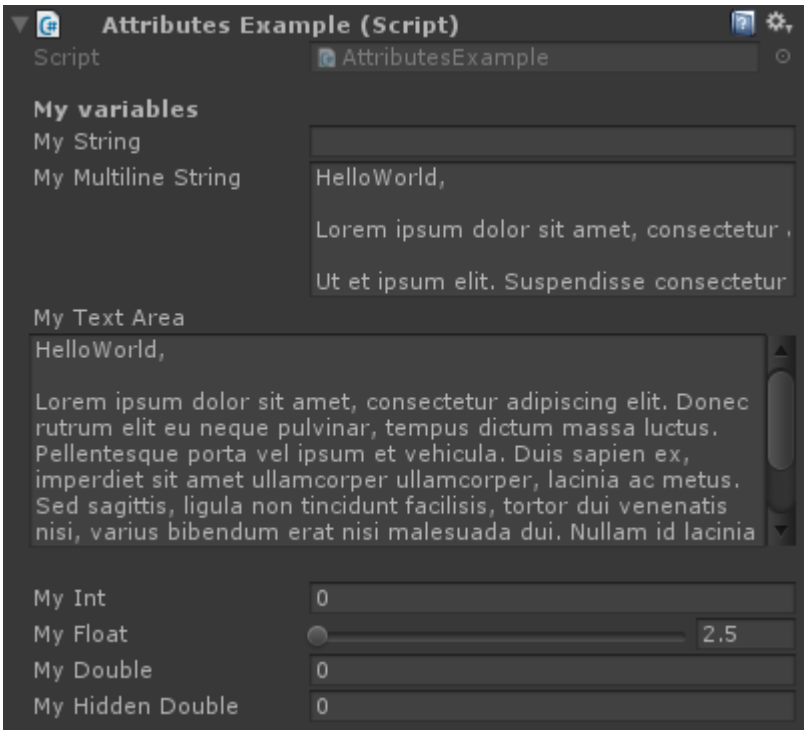
Su campo, incluso si está marcado con el atributo SerializeField, solo se atribuirá si es de un tipo que Unity puede serializar, que son:

- Todas las clases heredadas de UnityEngine.Object (por ejemplo, GameObject, Component, MonoBehaviour, Texture2D)
- Todos los tipos de datos básicos como int, string, float, bool
- Algunos tipos incorporados como Vector2 / 3/4, Quaternion, Matrix4x4, Color, Rect, LayerMask
- Arreglos de un tipo serializable.
- Lista de un tipo serializable
- Enums
- Estructuras

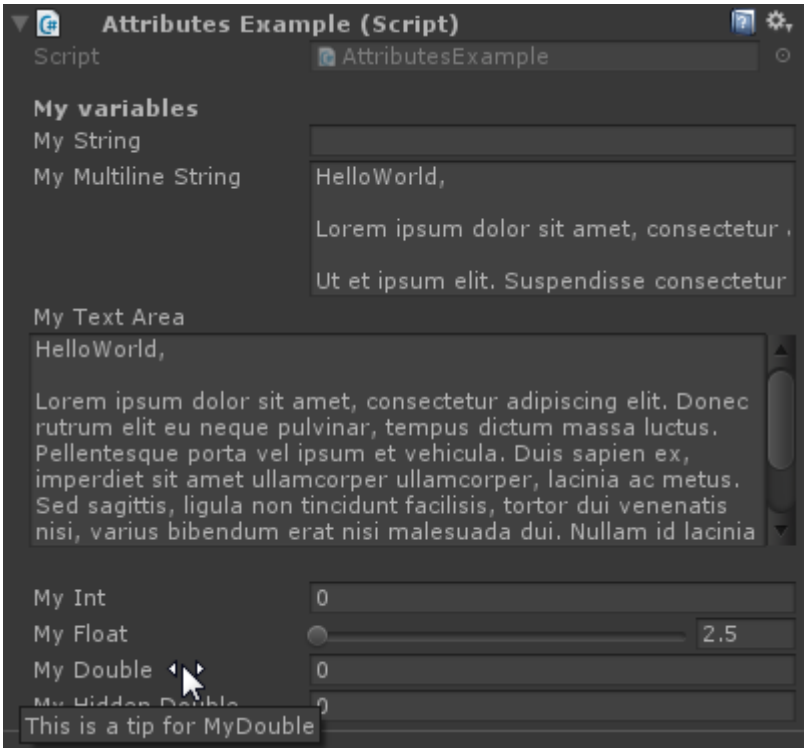
Examples

Atributos de inspector comunes

```
[Header( "My variables" )]
public string MyString;

[HideInInspector]
public string MyHiddenString;

[Multiline( 5 )]
public string MyMultilineString;

[TextArea( 2, 8 )]
public string MyTextArea;

[Space( 15 )]
public int MyInt;

[Range( 2.5f, 12.5f )]
public float MyFloat;

[Tooltip( "This is a tip for MyDouble" )]
public double MyDouble;

[SerializeField]
private double myHiddenDouble;
```



Al pasar el cursor sobre la etiqueta de un campo:



```
[Header( "My variables" )]
public string MyString;
```

El encabezado coloca una etiqueta en negrita que contiene el texto sobre el campo atribuido. Esto se usa a menudo para etiquetar grupos para que se destaquen frente a otras etiquetas.

```
[HideInInspector]
public string MyHiddenString;
```

HideInInspector evita que los campos públicos se muestren en el inspector. Esto es útil para acceder a campos de otras partes del código donde no son visibles o mutables.

```
[Multiline( 5 )]
public string MyMultilineString;
```

Multiline crea un cuadro de texto con un número específico de líneas. Superar esta cantidad no expandirá el cuadro ni envolverá el texto.

```
[TextArea( 2, 8 )]
public string MyTextArea;
```

TextArea permite texto de estilo multilínea con **ajuste** automático de texto y barras de desplazamiento si el texto supera el área asignada.

```
[Space( 15 )]
public int MyInt;
```

El espacio obliga al inspector a agregar espacio adicional entre los elementos anteriores y actuales, lo que es útil para distinguir y separar grupos.

```
[Range( 2.5f, 12.5f )]
public float MyFloat;
```

El rango fuerza un valor numérico entre un mínimo y un máximo. Este atributo también funciona en enteros y dobles, aunque min y max se especifican como flotantes.

```
[Tooltip( "This is a tip for MyDouble" )]
public double MyDouble;
```

La información sobre herramientas muestra una descripción adicional cada vez que se mueve el cursor sobre la etiqueta del campo.

```
[SerializeField]
private double myHiddenDouble;
```

SerializeField obliga a Unity a serializar el campo, útil para campos privados.

Atributos del componente

```
[DisallowMultipleComponent]
[RequireComponent( typeof( Rigidbody ) )]
public class AttributesExample : MonoBehaviour
{
    [...]
}
```

```
[DisallowMultipleComponent]
```

El atributo DisallowMultipleComponent evita que los usuarios agreguen varias instancias de este componente a un GameObject.

```
[RequireComponent( typeof( Rigidbody ) )]
```

El atributo RequireComponent le permite especificar otro componente (o más) como requisitos para cuando este componente se agrega a un GameObject. Cuando agrega este componente a un GameObject, los componentes requeridos se agregarán automáticamente (si no están ya presentes) y esos componentes no podrán eliminarse hasta que se elimine el que los requiere.

Atributos de tiempo de ejecución

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
{

    [RuntimeInitializeOnLoadMethod]
    private static void FooBar()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
    private static void Foo()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
    private static void Bar()
    {
        [...]
    }

    void Update()
    {
        if ( Application.isEditor )
        {
            [...]
        }
        else
        {
            [...]
        }
    }
}
```

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
```

El atributo ExecuteInEditMode obliga a Unity a ejecutar los métodos mágicos de este script incluso cuando el juego no se está ejecutando.

> Las funciones no se llaman constantemente como en el modo de juego

- La actualización solo se llama cuando algo en la escena cambia.
- Se llama a OnGUI cuando la vista de juego recibe un evento.
- OnRenderObject y las demás funciones de devolución de llamada de renderización se activan en cada repintado de la Vista de escena o Vista de juego.

```
[RuntimeInitializeOnLoadMethod]
private static void FooBar()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
private static void Foo()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
private static void Bar()
```

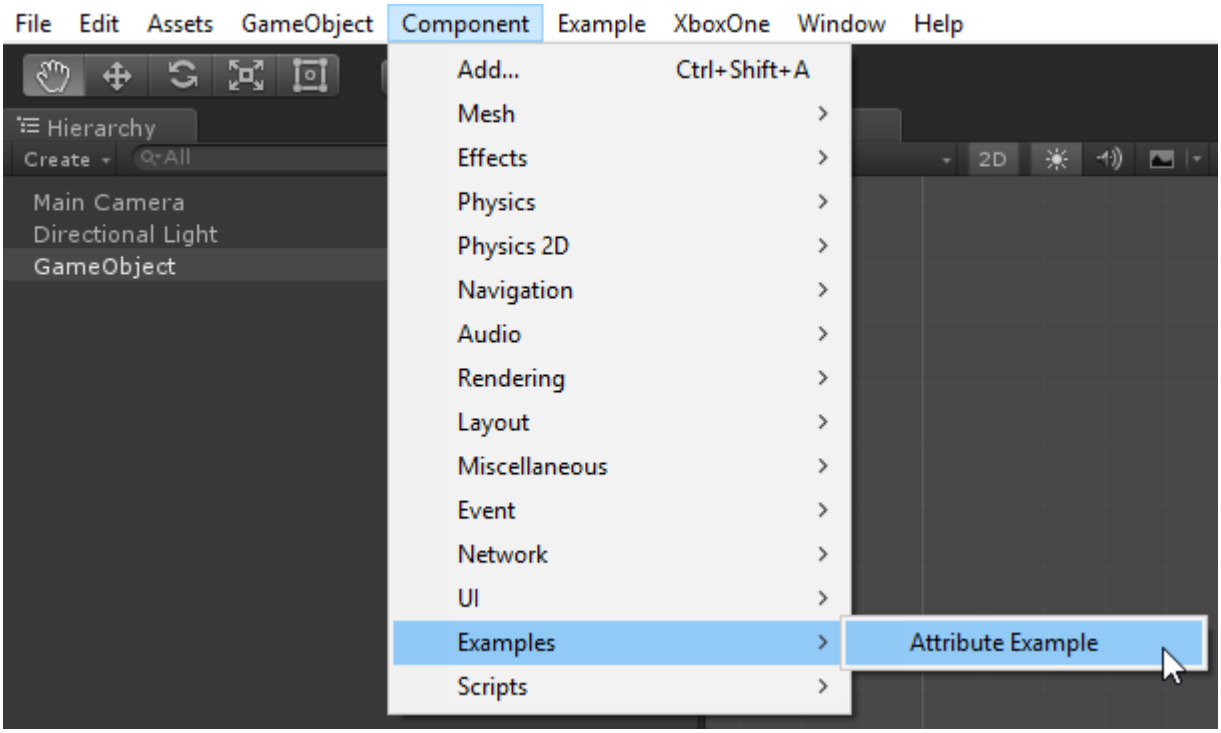
El atributo RuntimeInitializeOnLoadMethod permite llamar a un método de clase de tiempo de ejecución cuando el juego carga el tiempo de ejecución, sin ninguna interacción del usuario.

Puede especificar si desea que el método se invoque antes o después de la carga de la escena (después es predeterminado). El orden de ejecución no está garantizado para los métodos que utilizan este atributo.

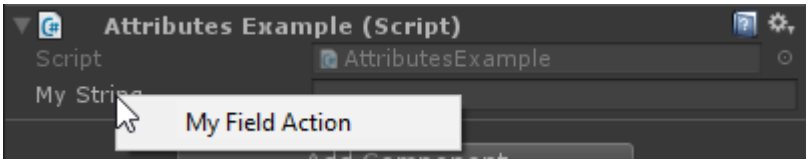
Atributos del menú

```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
{
    [ContextMenuItem( "My Field Action", "MyFieldContextAction" )]
    public string MyString;

    private void MyFieldContextAction()
    {
        [...]
    }

    [ContextMenu( "My Action" )]
    private void MyContextMenuAction()
    {
        [...]
    }
}
```

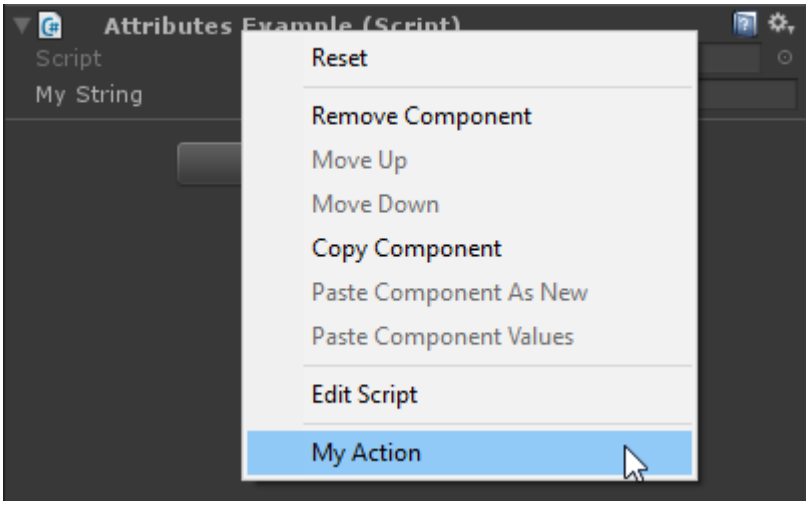
El resultado del atributo [AddComponentMenu]



El resultado del atributo [ContextMenuItem]



El resultado del atributo [ContextMenu]



```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
```

El atributo AddComponentMenu le permite colocar su componente en cualquier lugar del menú Componente en lugar del menú Componente-> Scripts.

```
[ContextMenuItem( "My Field Action", "MyFieldContextAction" )]
public string MyString;
```

```
private void MyFieldContextAction()
{
    [...]
}
```

El atributo ContextMenuItem le permite definir funciones que se pueden agregar al menú contextual de un campo. Estas funciones se ejecutarán tras la selección.

```
[ContextMenu( "My Action" )]
private void MyContextMenuAction()
{
    [...]
}
```

El atributo ContextMenu le permite definir funciones que se pueden agregar al menú contextual del componente.

Atributos del editor

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{

    static AttributesExample()
    {
        [...]
    }

    [InitializeOnLoadMethod]
    private static void Foo()
    {
        [...]
    }
}
```

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{

    static AttributesExample()
    {
        [...]
    }
```

El atributo InitializeOnLoad permite al usuario inicializar una clase sin ninguna interacción del usuario. Esto sucede cada vez que el editor se inicia o en una recompilación. El constructor estático garantiza que esto será llamado antes que cualquier otra función estática.

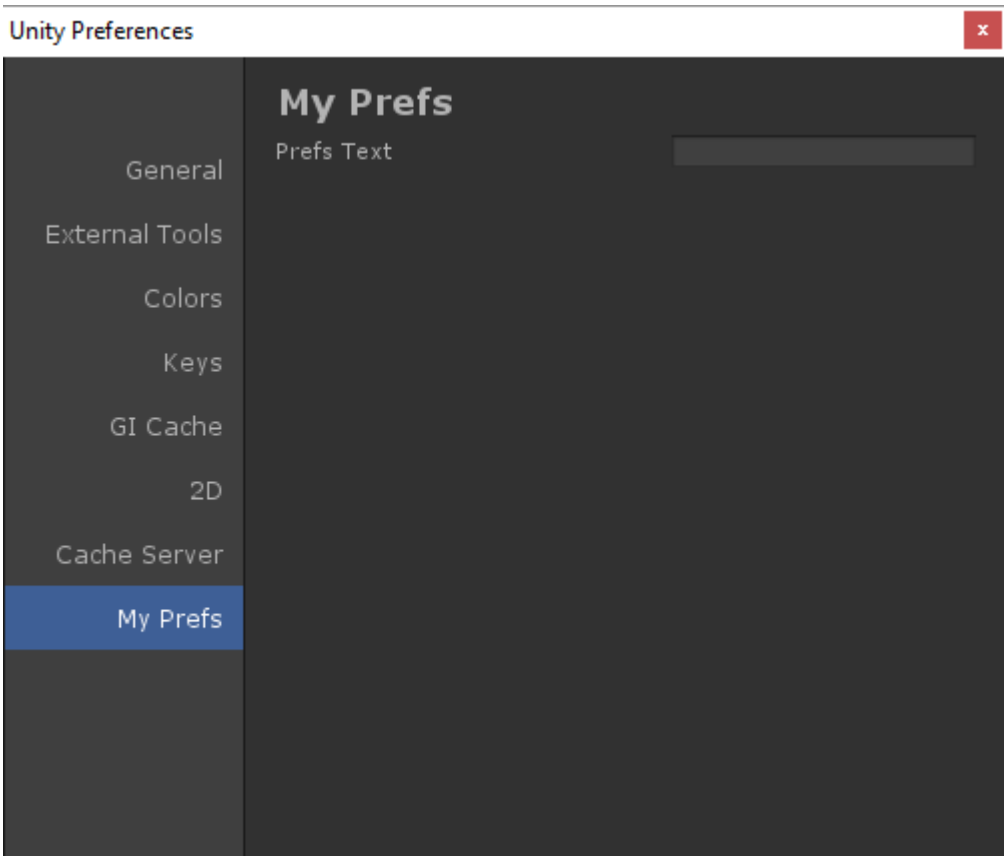
```
[InitializeOnLoadMethod]
private static void Foo()
{
    [...]
```

```
}
```

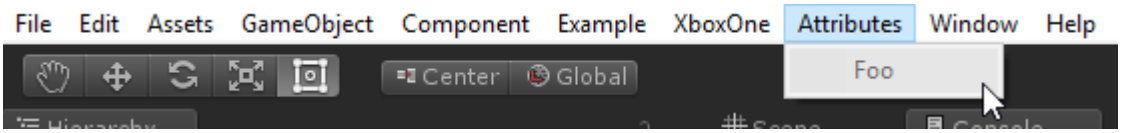
El atributo InitializeOnLoad permite al usuario inicializar una clase sin ninguna interacción del usuario. Esto sucede cada vez que el editor se inicia o en una recompilación. El orden de ejecución no está garantizado para los métodos que utilizan este atributo.

```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
{

    public int MyInt;

    private static string prefsText = "";

    [PreferenceItem( "My Prefs" )]
    public static void PreferencesGUI()
    {
        prefsText = EditorGUILayout.TextField( "Prefs Text", prefsText );
    }

    [MenuItem( "Attributes/Foo" )]
    private static void Foo()
    {
        [...]
    }

    [MenuItem( "Attributes/Foo", true )]
    private static bool FooValidate()
    {
        return false;
    }
}
```

El resultado del atributo [PreferenceItem]



El resultado del atributo [MenuItem]



```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
```

El atributo CanEditMultipleObjects le permite editar valores de su componente sobre múltiples GameObjects. Sin este componente, no verá su componente como normal cuando selecciona múltiples GameObjects, sino que verá el mensaje "No se admite la edición de objetos múltiples"

> Este atributo es para que los editores personalizados admitan la edición múltiple. Los editores no personalizados admiten automáticamente la edición múltiple.

```
[PreferenceItem( "My Prefs" )]
public static void PreferencesGUI()
```

El atributo PreferenceItem te permite crear un elemento adicional en el menú de preferencias de Unity. El método de recepción debe ser estático para que se utilice.

```
[MenuItem( "Attributes/Foo" )]
private static void Foo()
{
    [...]
}
```

```
[MenuItem( "Attributes/Foo", true )]
private static bool FooValidate()
{
    return false;
}
```

El atributo MenuItem le permite crear elementos de menú personalizados para ejecutar funciones. Este ejemplo también utiliza una función de validador (que siempre devuelve falso) para evitar la ejecución de la función.

```
[CustomEditor( typeof( MyComponent ) )]
public class AttributesExample : Editor
{
    [...]
}
```

El atributo CustomEditor le permite crear editores personalizados para sus componentes. Estos editores se utilizarán para dibujar su componente en el inspector y deben derivarse de la clase Editor.

```
[CustomPropertyDrawer( typeof( MyClass ) )]
public class AttributesExample : PropertyDrawer
{
    [...]
}
```

El atributo CustomPropertyDrawer le permite crear un cajón de propiedades personalizado en el inspector. Puede usar estos cajones para sus tipos de datos personalizados para que puedan verse en el inspector.

```
[DrawGizmo( GizmoType.Selected )]
private static void DoGizmo( AttributesExample obj, GizmoType type )
{
    [...]
}
```

El atributo DrawGizmo le permite dibujar artilugios personalizados para sus componentes. Estos gizmos se dibujarán en la vista de escena. Puede decidir cuándo dibujar el gizmo usando el parámetro GizmoType en el atributo DrawGizmo.

> El método de recepción requiere dos parámetros, el primero es el componente para dibujar el gizmo y el segundo es el estado en el que se encuentra el objeto que necesita el gizmo dibujado.

Lea Atributos en línea: https://riptutorial.com/es/unity3d/topic/5535/atributos

Capítulo 6: Capas

Examples

Uso de la capa

Las capas de Unity son similares a las etiquetas, ya que se pueden usar para definir objetos con los que se debe interactuar o se deben comportar de cierta manera; sin embargo, las capas se usan principalmente con funciones en la clase de `Physics` : Documentación de Unity - Física

Las capas están representadas por un entero y se pueden pasar a las funciones de esta manera:

```
using UnityEngine;
class LayerExample {

    public int layer;

    void Start()
    {
        Collider[] colliders = Physics.OverlapSphere(transform.position, 5f, layer);
    }
}
```

El uso de una capa de esta manera incluirá solo Colisionadores cuyos GameObjects tengan la capa especificada en los cálculos realizados. Esto simplifica aún más la lógica y mejora el rendimiento.

Estructura LayerMask

La estructura `LayerMask` es una interfaz que funciona casi exactamente como pasar un entero a la función en cuestión. Sin embargo, su mayor beneficio es permitir al usuario seleccionar la capa en cuestión de un menú desplegable en el inspector.

```
using UnityEngine;
class LayerMaskExample{

    public LayerMask mask;
    public Vector3 direction;

    void Start()
    {
        if(Physics.Raycast(transform.position, direction, 35f, mask))
        {
            Debug.Log("Raycast hit");
        }
    {
}
```

También tiene múltiples funciones estáticas que permiten convertir nombres de capa en índices o índices en nombres de capa.

```
using UnityEngine;
class NameToLayerExample{

    void Start()
    {
        int layerindex = LayerMask.NameToLayer("Obstacle");
    {
}
```

Para facilitar la comprobación de la capa, defina el siguiente método de extensión.

```
public static bool IsInLayerMask(this GameObject @object, LayerMask layerMask)
{
    bool result = (1 << @object.layer & layerMask) == 0;

    return result;
}
```

Este método le permitirá verificar si un objeto de juego está en una tarea (seleccionada en el editor) o no.

Lea Capas en línea: https://riptutorial.com/es/unity3d/topic/4762/capas

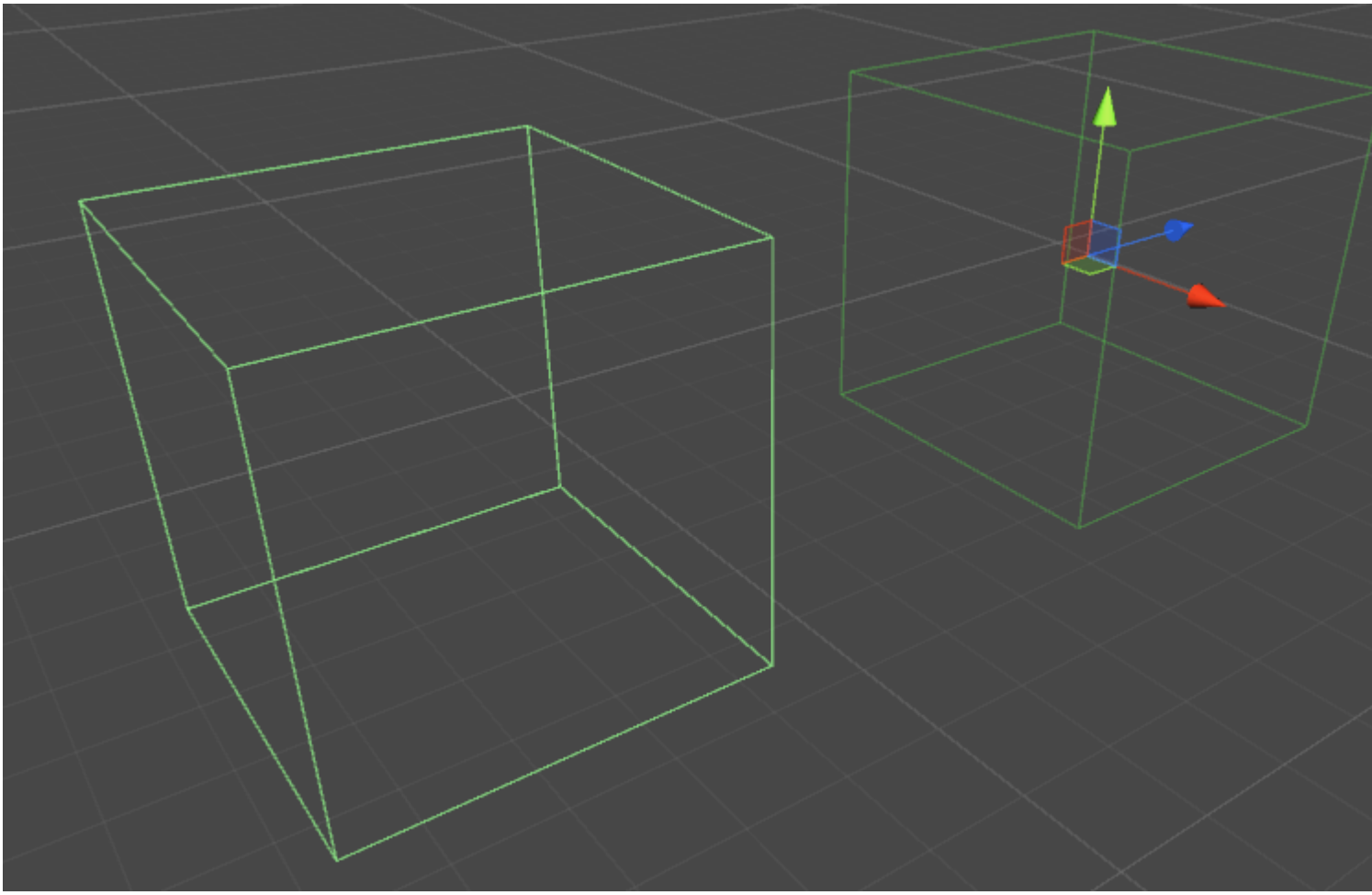
Capítulo 7: Colisión

Examples

Colisionadores

Box Collider

Un coleccionista primitivo con forma de cuboide.



Propiedades

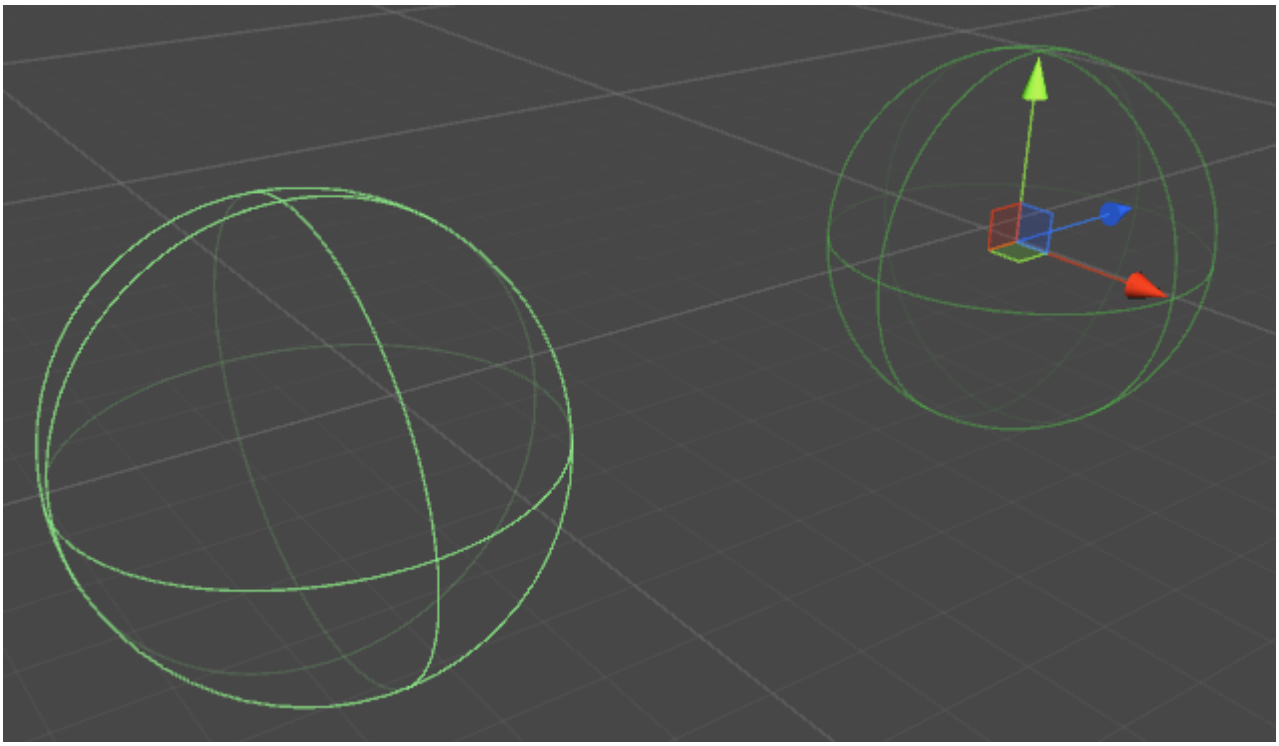
- **Is Trigger** : si está marcado, el Box Collider ignorará la física y se convertirá en un Trigger Collider
- **Material** : una referencia, si se especifica, al material de física del Box Collider.
- **Centro** - La posición central de The Box Collider en el espacio local.
- **Tamaño** : el tamaño del Box Collider medido en el espacio local.

Ejemplo

```
// Add a Box Collider to the current GameObject.
BoxCollider myBC = BoxCollider)myGameObject.gameObject.AddComponent(typeof(BoxCollider));

// Make the Box Collider into a Trigger Collider.
myBC.isTrigger= true;

// Set the center of the Box Collider to the center of the GameObject.
myBC.center = Vector3.zero;

// Make the Box Collider twice as large.
myBC.size = 2;
```

Colisionador de esfera

Un coleccionista primitivo con forma de esfera.



Propiedades

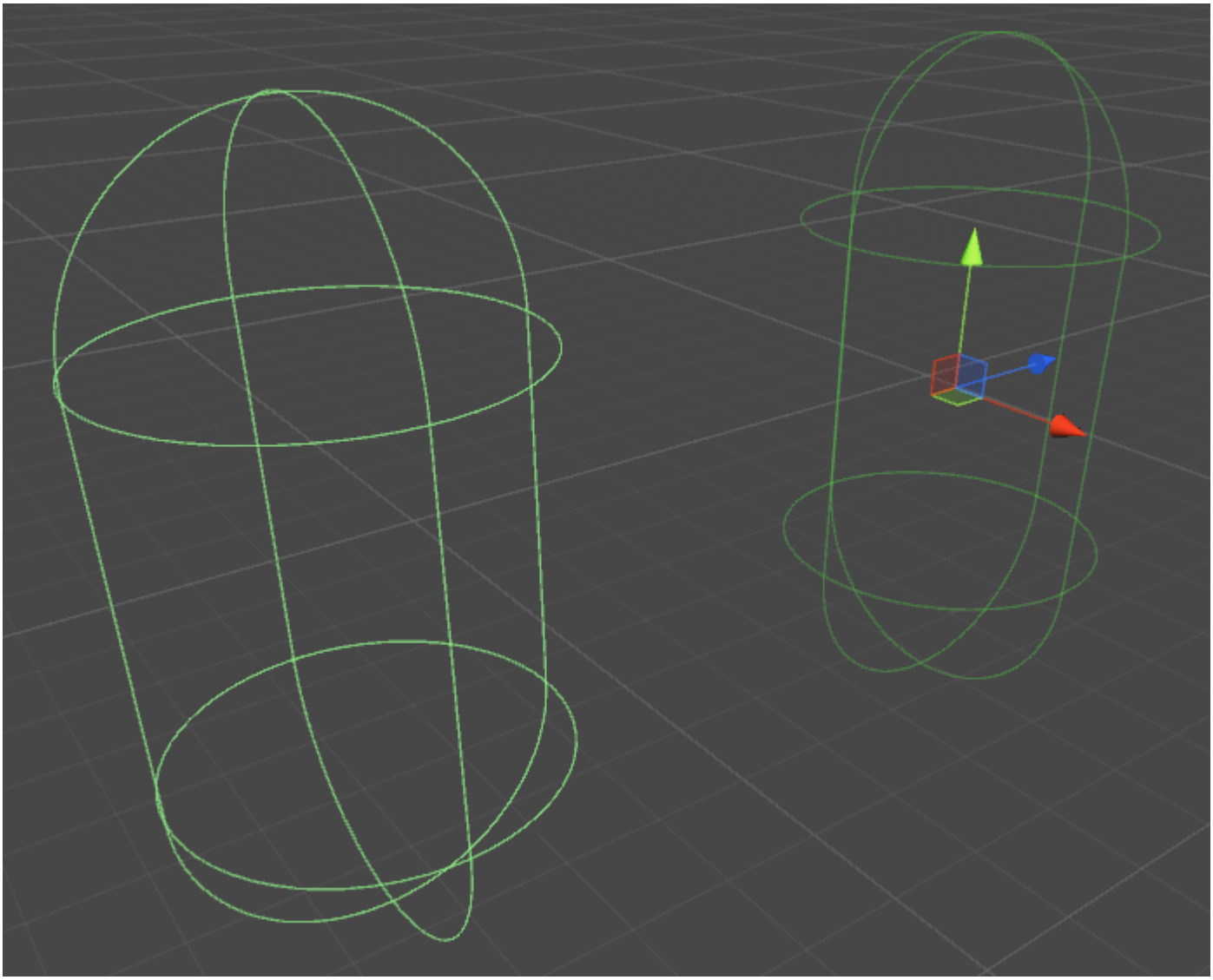
- **Is Trigger** : si está marcado, Sphere Collider ignorará la física y se convertirá en un Trigger Collider.
- **Material** : una referencia, si se especifica, al material de física del Colisionador de Esfera.
- **Centro** - La posición central de The Sphere Collider en el espacio local.
- **Radio** - El radio del Colisionador

Ejemplo

```
// Add a Sphere Collider to the current GameObject.
SphereCollider mySC =
SphereCollider)myGameObject.gameObject.AddComponent(typeof(SphereCollider));

// Make the Sphere Collider into a Trigger Collider.
mySC.isTrigger= true;

// Set the center of the Sphere Collider to the center of the GameObject.
mySC.center = Vector3.zero;

// Make the Sphere Collider twice as large.
mySC.radius = 2;
```

Colisionador de cápsulas

Dos medias esferas unidas por un cilindro.



Propiedades

- **Está activado** : si está marcado, Capsule Collider ignorará la física y se convertirá en un activador.
- **Material** : una referencia, si se especifica, al material de física del Capsule Collider.
- **Centro** - La posición central de Capsule Collider en el espacio local.
- **Radio** - El radio en el espacio local.
- **Altura** - **Altura** total del colisionador
- **Dirección** - El eje de orientación en el espacio local.

Ejemplo

```
// Add a Capsule Collider to the current GameObject.
CapsuleCollider myCC =
CapsuleCollider)myGameObject.gameObject.AddComponent(typeof(CapsuleCollider));

// Make the Capsule Collider into a Trigger Collider.
myCC.isTrigger= true;

// Set the center of the Capsule Collider to the center of the GameObject.
myCC.center = Vector3.zero;

// Make the Sphere Collider twice as tall.
myCC.height= 2;

// Make the Sphere Collider twice as wide.
myCC.radius= 2;

// Set the axis of lengthwise orientation to the X axis.
myCC.direction = 0;

// Set the axis of lengthwise orientation to the Y axis.
myCC.direction = 1;

// Set the axis of lengthwise orientation to the Y axis.
myCC.direction = 2;
```

Colisionador de ruedas

Propiedades

- **Misa** - La masa del colisionador de ruedas
- **Radio** - El radio en el espacio local.

- **Velocidad de** amortiguación de la rueda - Valor de amortiguación para el recolector de ruedas
- **Distancia de suspensión** : extensión máxima a lo largo del eje Y en el espacio local
- **Distancia de punto de aplicación de fuerza** : el punto donde se aplicarán las fuerzas,
- **Centro** - Centro del colisionador de ruedas en el espacio local

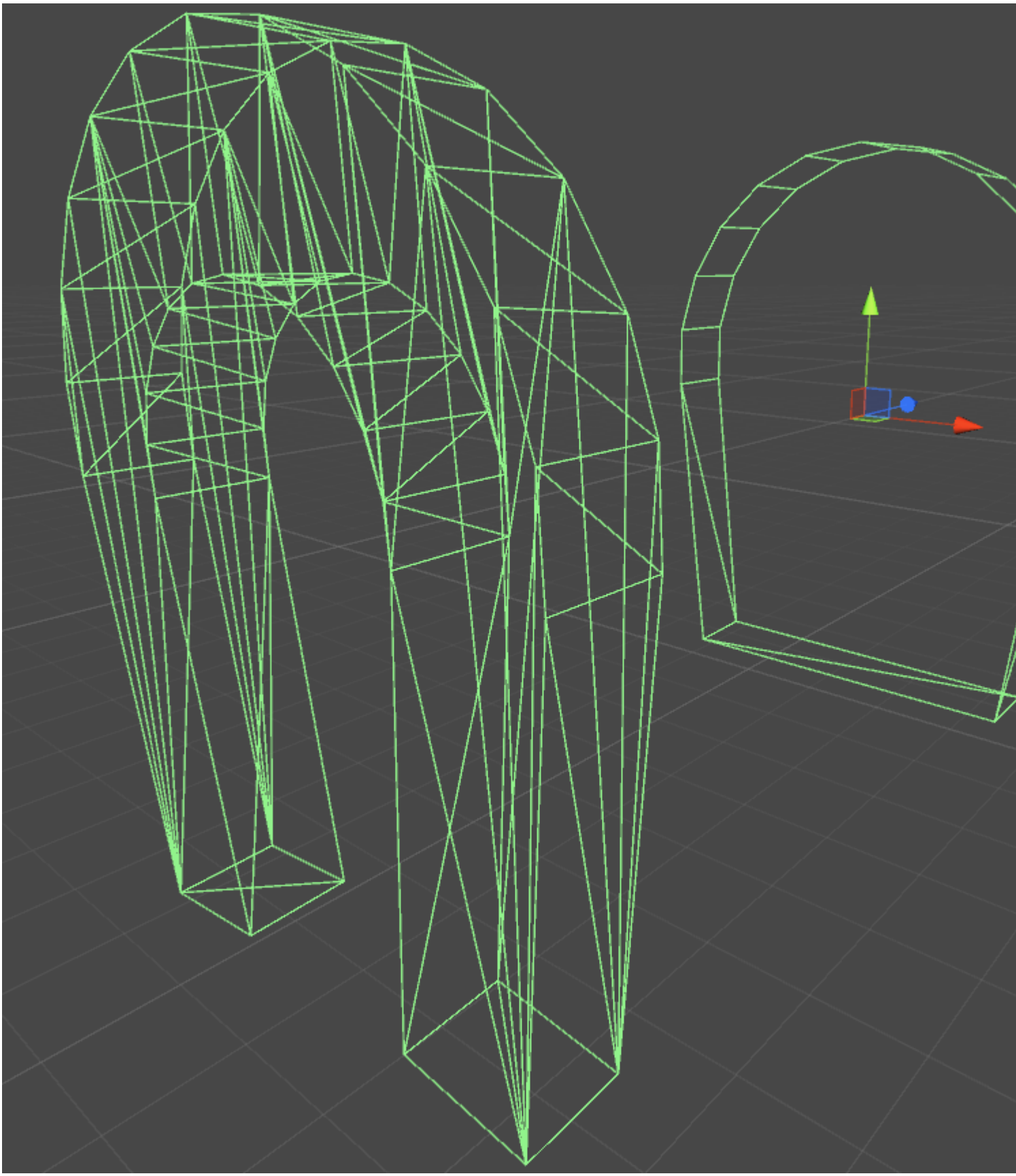
Muelle de suspensión

- **Primavera** : la velocidad a la que la rueda intenta volver a la posición de destino.
- **Amortiguador** : un valor más grande amortigua más la velocidad y la suspensión se mueve más lentamente
- **Posición de destino** : el valor predeterminado es 0.5, en 0 la suspensión es de abajo hacia abajo, en 1 está en la extensión completa
- **Fricción delantera / lateral** : cómo se comporta el neumático al rodar hacia delante o hacia los lados

Ejemplo

Colisionador de malla

Un colisionador basado en un activo de malla.



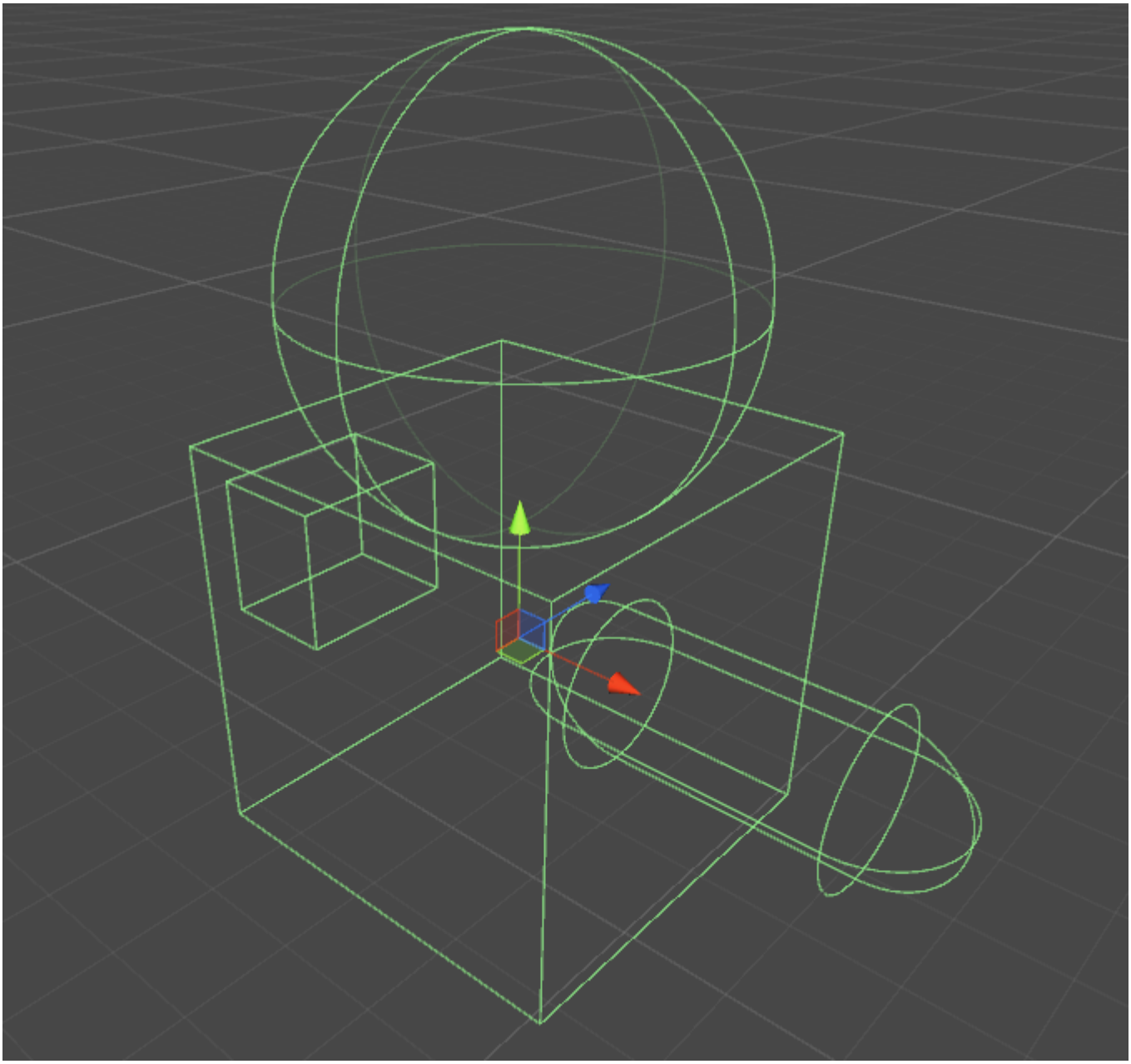
Propiedades

- **Is Trigger** : si está marcado, el Box Collider ignorará la física y se convertirá en un Trigger Collider

- **Material** : una referencia, si se especifica, al material de física del Box Collider.
- **Malla** - Una referencia a la malla en la que se basa el Colisionador
- **Convexo** : los colectores de malla convexos están limitados a 255 polígonos: si está habilitado, este colisionador puede colisionar con otros colectores de malla

Ejemplo

Si aplica más de un Colisionador a un GameObject, lo llamamos un Colisionador Compuesto.



Colisionador de ruedas

El colisionador de ruedas dentro de la unidad se basa en el colisionador de ruedas PhysX de Nvidia y, por lo tanto, comparte muchas propiedades similares. Técnicamente, la unidad es un programa "sin unidad", pero para que todo tenga sentido, se requieren algunas unidades

estándar.

Propiedades básicas

- Masa: el peso de la rueda en Kilogramos, esto se utiliza para el momento de la rueda y el momento de interia al girar.
- Radio - en metros, el radio del colisionador.
- Velocidad de amortiguación de la rueda: ajusta la "respuesta" de las ruedas al par aplicado.
- Distancia de suspensión: distancia total de desplazamiento en metros que la rueda puede recorrer
- Distancia de punto de aplicación de la fuerza: ¿dónde está la fuerza de la suspensión aplicada al cuerpo rígido padre?
- Centro - La posición central de la rueda.

Ajustes de suspensión

- Primavera: esta es la constante de resorte, K, en Newtons / metro en la ecuación:

  Fuerza = Primavera constante * Distancia

Un buen punto de partida para este valor debe ser la masa total de su vehículo, dividida por el número de ruedas, multiplicada por un número entre 50 y 100. Por ejemplo, si tiene un automóvil de 2,000 kg con 4 ruedas, entonces cada rueda deberá soporte 500kg. Multiplica esto por 75, y tu constante de primavera debería ser de 37,500 Newtons / metro.

- Amortiguador: el equivalente a un amortiguador en un automóvil. Las tasas más altas hacen que el suspenso sea "más rígido" y las tasas más bajas lo hacen "más suave" y es más probable que oscile.
  No conozco las unidades o la ecuación para esto, creo que tiene que ver con una ecuación de frecuencia en física.

Configuración de fricción lateral

La curva de fricción en la unidad tiene un valor de deslizamiento determinado por cuánto se desliza la rueda (en m / s) desde la posición deseada en comparación con la posición real.

- Extremum Slip: esta es la cantidad máxima (en m / s) que una rueda puede deslizar antes de que pierda tracción

- Valor extremo: esta es la cantidad máxima de fricción que se debe aplicar a una rueda.

Los valores de Exrtremum Slip deben estar entre .2 y 2m / s para los autos más realistas. 2 m / s es de aproximadamente 6 pies por segundo o 5 mph, que es una gran cantidad de deslizamiento. Si cree que su vehículo necesita tener un valor superior a 2 m / s para el deslizamiento, debe considerar aumentar la fricción máxima (Valor extremo).

Fracción máxima (valor extremo) es el coeficiente de fricción en la ecuación:

> Fuerza de fricción (en newtons) = Coeficiente de fricción * Fuerza a la baja (en newtons)

Esto significa que con un coeficiente de 1, está aplicando toda la fuerza de la cabina + suspensión opuesta a la dirección de deslizamiento. En aplicaciones del mundo real, los valores superiores a 1 son raros, pero no imposibles. Para un neumático sobre asfalto seco, los valores entre .7 y .9 son realistas, por lo que es preferible el valor predeterminado de 1.0.

Este valor no debe ser realísticamente superior a 2.5, ya que comenzará a ocurrir un comportamiento extraño. Por ejemplo, empiezas a girar a la derecha, pero debido a que este valor es tan alto, se aplica una gran fuerza en sentido opuesto a tu dirección y comienzas a deslizarte en el giro en lugar de alejarte.

Si ha maximizado ambos valores, entonces debería comenzar a elevar el deslizamiento y el valor de la asíntota. El deslizamiento de asíntota debe estar entre 0.5 y 2 m / s, y define el coeficiente de fricción para cualquier valor de deslizamiento más allá del deslizamiento de asíntota. Si encuentra que sus vehículos se comportan bien hasta que rompe la tracción, en cuyo punto actúa como si estuviera en el hielo, debe aumentar el valor de Asymptote. Si descubre que su vehículo no puede desviarse, debe disminuir el valor.

Fricción Delantera

La fricción hacia adelante es idéntica a la fricción lateral, con la excepción de que esto define cuánta tracción tiene la rueda en la dirección del movimiento. Si los valores son demasiado bajos, sus vehículos harán quemaduras y solo harán girar los neumáticos antes de avanzar, lentamente. Si es demasiado alto, su vehículo puede tener la tendencia de intentar hacer una voltereta o algo peor.

Notas adicionales

No espere poder crear un clon GTA u otro clon de carreras simplemente ajustando estos valores. En la mayoría de los juegos de conducción, estos valores se cambian constantemente en el script para diferentes velocidades, terrenos y valores de giro. Además, si solo está aplicando un par constante a los colisionadores de ruedas cuando se presiona una tecla, su juego no se comportará de manera realista. En el mundo real, los automóviles tienen curvas de par y transmisiones para cambiar el par aplicado a las ruedas.

Para obtener los mejores resultados, debe ajustar estos valores hasta que el automóvil responda razonablemente bien y luego realice cambios en el par de ruedas, el ángulo de giro máximo y los valores de fricción en el script.

Puede encontrar más información sobre los colisionadores de ruedas en la documentación de Nvidia:
http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Vehicles.html

Colisionadores de gatillo

Métodos

- `OnTriggerEnter()`
- `OnTriggerStay()`

- `OnTriggerExit()`

Puede convertir un Colisionador en un **Disparador** para utilizar los `OnTriggerEnter()` , `OnTriggerStay()` y `OnTriggerExit()` . Un Trigger Collider no reaccionará físicamente a las colisiones, otros GameObjects simplemente lo atravesarán. Son útiles para detectar cuándo otro GameObject está en un área determinada o no, por ejemplo, cuando recopilamos un elemento, es posible que deseamos poder ejecutarlo pero detectar cuándo sucede esto.

Trigger Collider Scripting

Ejemplo

El método a continuación es un ejemplo de un oyente de activación que detecta cuando otro colisionador ingresa al colector de un GameObject (como un jugador). Los métodos de activación se pueden agregar a cualquier secuencia de comandos asignada a un GameObject.

```
void OnTriggerEnter(Collider other)
{
    //Check collider for specific properties (Such as tag=item or has component=item)
}
```

Lea Colisión en línea: https://riptutorial.com/es/unity3d/topic/4405/colision

Capítulo 8: Cómo utilizar paquetes de activos

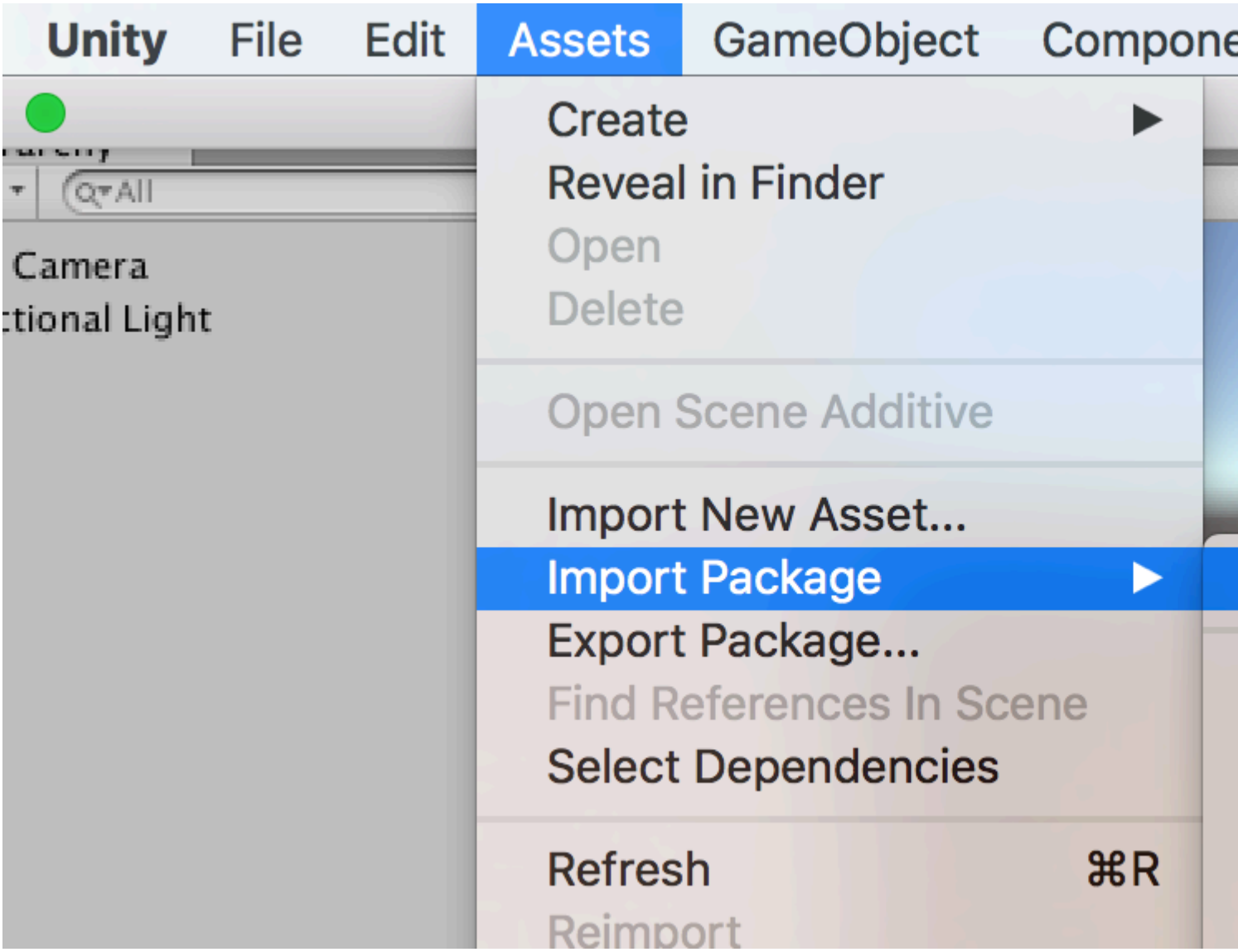
Examples

Paquetes de activos

Los Paquetes de Activos (con el formato de archivo de `.unitypackage`) son una forma comúnmente utilizada para distribuir proyectos de Unity a otros usuarios. Cuando trabaje con periféricos que tienen sus propios SDK (por ejemplo, Oculus), se le puede solicitar que descargue e importe uno de estos paquetes.

Importando un paquete .unity

Para importar un paquete, vaya a la barra de menú de Unity y haga clic en `Assets > Import Package > Custom Package...` , luego navegue hasta el archivo `.unitypackage` en el Explorador de archivos que aparece.



Lea Cómo utilizar paquetes de activos en línea: https://riptutorial.com/es/unity3d/topic/4491/como-utilizar-paquetes-de-activos

Capítulo 9: Comunicación con el servidor

Examples

Obtener

Obtener es obtener datos del servidor web. y `new WWW("https://urlexample.com");` con un url pero sin un segundo parámetro está haciendo un **Get** .

es decir

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    public string url = "http://google.com";

    IEnumerator Start()
    {
        WWW www = new WWW(url); // One get.
        yield return www;
        Debug.Log(www.text); // The data of the url.
    }
}
```

Post simple (Campos Post)

Cada instancia de **WWW** con un segundo parámetro es una *publicación* .

Aquí hay un ejemplo para publicar *el ID de usuario* y la *contraseña* en el servidor.

```
void Login(string id, string pwd)
{
    WWWForm dataParameters = new WWWForm();    // Create a new form.
    dataParameters.AddField("username", id);
    dataParameters.AddField("password", pwd);   // Add fields.
    WWW www = new WWW(url+"/account/login",dataParameters);
    StartCoroutine("PostdataEnumerator", www);
}

IEnumerator PostdataEnumerator(WWW www)
{
    yield return www;
    if (!string.IsNullOrEmpty(www.error))
    {
        Debug.Log(www.error);
    }
    else
    {
        Debug.Log("Data Submitted");
    }
}
```

Publicar (subir un archivo)

Subir un archivo al servidor es también una publicación. Puede cargar fácilmente un archivo a través de **WWW** , como el siguiente:

Subir un archivo zip al servidor

```
string mainUrl = "http://server/upload/";
string saveLocation;

void Start()
{
    saveLocation = "ftp:///home/xxx/x.zip"; // The file path.
    StartCoroutine(PrepareFile());
}

// Prepare The File.
IEnumerator PrepareFile()
{
    Debug.Log("saveLoacation = " + saveLocation);

    // Read the zip file.
    WWW loadTheZip = new WWW(saveLocation);

    yield return loadTheZip;

    PrepareStepTwo(loadTheZip);
}

void PrepareStepTwo(WWW post)
{
    StartCoroutine(UploadTheZip(post));
}

// Upload.
IEnumerator UploadTheZip(WWW post)
{
    // Create a form.
    WWWForm form = new WWWForm();

    // Add the file.
    form.AddBinaryData("myTestFile.zip",post.bytes,"myFile.zip","application/zip");

    // Send POST request.
    string url = mainUrl;
    WWW POSTZIP = new WWW(url,form);

    Debug.Log("Sending zip...");
    yield return POSTZIP;
    Debug.Log("Zip sent!");
}
```

En este ejemplo, usa la **rutina** para preparar y cargar el archivo. Si quieres saber más acerca de Unity coroutines, visita Coroutines .

Enviando una solicitud al servidor.

Hay muchas formas de comunicarse con los servidores utilizando Unity como cliente (algunas metodologías son mejores que otras según su propósito). Primero, uno debe determinar la necesidad del servidor para poder enviar operaciones de manera efectiva hacia y desde el servidor. Para este ejemplo, enviaremos algunos datos a nuestro servidor para validarlos.

Lo más probable es que el programador haya configurado algún tipo de controlador en su servidor para recibir eventos y responder al cliente en consecuencia, sin embargo, eso está fuera del alcance de este ejemplo.

DO#:

```
using System.Net;
using System.Text;

public class TestCommunicationWithServer
{
    public string SendDataToServer(string url, string username, string password)
    {
        WebClient client = new WebClient();

        // This specialized key-value pair will store the form data we're sending to the
server
        var loginData = new System.Collections.Specialized.NameValueCollection();
        loginData.Add("Username", username);
        loginData.Add("Password", password);

        // Upload client data and receive a response
        byte[] opBytes = client.UploadValues(ServerIpAddress, "POST", loginData);

        // Encode the response bytes into a proper string
        string opResponse = Encoding.UTF8.GetString(opBytes);

        return opResponse;
    }
```

Lo primero que hay que hacer es tirar sus declaraciones de uso que nos permiten usar las clases WebClient y NameValueCollection.

Para este ejemplo, la función SendDataToServer toma 3 parámetros de cadena (opcionales):

1. Url del servidor con el que nos estamos comunicando.
2. Primer dato
3. Segunda pieza de datos que estamos enviando al servidor.

El nombre de usuario y la contraseña son los datos opcionales que estoy enviando al servidor. Para este ejemplo, lo estamos utilizando para validarlo luego desde una base de datos o cualquier otro almacenamiento externo.

Ahora que hemos configurado nuestra estructura, crearemos una instancia de un nuevo WebClient que se utilizará para enviar nuestros datos. Ahora necesitamos cargar nuestros datos en nuestra colección NameValue y cargar los datos en el servidor.

La función UploadValues toma en 3 parámetros necesarios también:

1. Dirección IP del servidor
2. Método HTTP
3. Datos que está enviando (el nombre de usuario y la contraseña en nuestro caso)

Esta función devuelve una matriz de bytes de la respuesta del servidor. Necesitamos codificar la matriz de bytes devuelta en una cadena adecuada para poder manipular y diseccionar la respuesta.

Uno podría hacer algo como esto:

```
if(opResponse.Equals(ReturnMessage.Success))
{
    Debug.Log("Unity client has successfully sent and validated data on server.");
}
```

Ahora es posible que todavía esté confundido, así que supongo que le daré una breve explicación de cómo manejar un servidor de respuesta.

Para este ejemplo usaré PHP para manejar la respuesta del cliente. Recomiendo usar PHP como su lenguaje de scripts de back-end porque es súper versátil, fácil de usar y, sobre todo, rápido. Definitivamente hay otras formas de manejar una respuesta en un servidor, pero en mi opinión, PHP es, con mucho, la implementación más sencilla y sencilla en Unity.

PHP:

```
// Check to see if the unity client send the form data
if(!isset($_REQUEST['Username']) || !isset($_REQUEST['Password']))
{
    echo "Empty";
}
else
{
    // Unity sent us the data - its here so do whatever you want

    echo "Success";

}
```

Así que esta es la parte más importante - el 'eco'. Cuando nuestro cliente carga los datos al servidor, el cliente guarda la respuesta (o recurso) en esa matriz de bytes. Una vez que el cliente tiene la respuesta, usted sabe que los datos se han validado y puede continuar en el cliente una vez que ese evento haya ocurrido. También debe pensar qué tipo de datos está enviando (hasta cierto punto) y cómo minimizar la cantidad que está enviando.

Así que esta es solo una forma de enviar / recibir datos de Unity: hay otras formas que pueden ser más efectivas para usted dependiendo de su proyecto.

Lea Comunicación con el servidor en línea:
https://riptutorial.com/es/unity3d/topic/5578/comunicacion-con-el-servidor

Capítulo 10: Coroutines

Sintaxis

- Coroutine StartCoroutine pública (rutina IEnumerator);
- public Coroutine StartCoroutine (string methodName, valor del objeto = nulo);
- public void StopCoroutine (string methodName);
- public void StopCoroutine (rutina IEnumerator);
- public void StopAllCoroutines ();

Observaciones

Consideraciones de rendimiento

Es mejor usar coroutines con moderación ya que la flexibilidad viene con un costo de rendimiento.

- Coroutines en gran número exige más de la CPU que los métodos de actualización estándar.
- Hay un problema en algunas versiones de Unity donde las rutinas producen basura en cada ciclo de actualización debido a que Unity `MoveNext` valor de retorno de `MoveNext` . Esto fue observado por última vez en 5.4.0b13. (Informe de error)

Reducir la basura mediante el almacenamiento en caché de instrucciones de rendimiento

Un truco común para reducir la basura generada en las corrutinas es almacenar en caché la `YieldInstruction` .

```
IEnumerator TickEverySecond()
{
    var wait = new WaitForSeconds(1f); // Cache
    while(true)
    {
        yield return wait; // Reuse
    }
}
```

El rendimiento `null` no produce basura adicional.

Examples

Coroutines

Primero, es esencial comprender que los motores de juego (como Unity) funcionan en un

paradigma "basado en marcos".

El código se ejecuta durante cada fotograma.

Eso incluye el propio código de Unity, y tu código.

Al pensar en los marcos, es importante comprender que no hay ***absolutamente*** ninguna garantía de cuándo suceden los marcos. ***No*** suceden en un ritmo regular. Los espacios entre cuadros podrían ser, por ejemplo, 0.02632, luego 0.021167, luego 0.029778 y así sucesivamente. En el ejemplo, todos están "aproximadamente" a 1/50 de segundo, pero todos son diferentes. Y en cualquier momento, puede obtener un marco que tarda mucho más tiempo o más corto; y su código puede ejecutarse en cualquier momento dentro del marco.

Teniendo esto en cuenta, puede preguntar: ¿cómo accede a estos marcos en su código, en Unity?

Sencillamente, usa la llamada Update () o una coroutina. (De hecho, son exactamente lo mismo: permiten que el código se ejecute en cada fotograma).

El propósito de una coroutina es que:

puede ejecutar algún código, y luego, "detenerse y esperar" ***hasta algún marco futuro*** .

Puede esperar hasta **el siguiente fotograma** , puede esperar **un número de fotogramas** , o puede esperar un tiempo ***aproximado*** en segundos en el futuro.

Por ejemplo, puede esperar "aproximadamente un segundo", lo que significa que esperará aproximadamente un segundo, y luego colocar su código en algún marco aproximadamente un segundo a partir de ahora. (Y de hecho, dentro de ese marco, el código podría ejecutarse en cualquier momento, en cualquier caso). Para repetir: no será exactamente un segundo. La sincronización precisa no tiene sentido en un motor de juego.

Dentro de una coroutina:

Para esperar un fotograma:

```
// do something
yield return null;  // wait until next frame
// do something
```

Para esperar tres cuadros:

```
// do something
yield return null;  // wait until three frames from now
yield return null;
yield return null;
// do something
```

Para esperar **aproximadamente** medio segundo:

```
// do something
yield return new WaitForSeconds (0.5f); // wait for a frame in about .5 seconds
// do something
```

Haz algo en cada cuadro individual:

```
while (true)
{
    // do something
    yield return null;  // wait until the next frame
}
```

Ese ejemplo es literalmente idéntico a simplemente poner algo dentro de la llamada "Actualización" de Unity: el código en "hacer algo" se ejecuta en cada fotograma.

Ejemplo

Adjuntar Ticker a un `GameObject` . Mientras ese objeto del juego esté activo, el tick se ejecutará. Tenga en cuenta que la secuencia de comandos detiene cuidadosamente la rutina, cuando el objeto del juego se vuelve inactivo; Este suele ser un aspecto importante de la correcta ingeniería de uso de coroutine.

```
using UnityEngine;
using System.Collections;

public class Ticker:MonoBehaviour {

    void OnEnable()
    {
        StartCoroutine(TickEverySecond());
    }

    void OnDisable()
    {
        StopAllCoroutines();
    }

    IEnumerator TickEverySecond()
    {
        var wait = new WaitForSeconds(1f); // REMEMBER: IT IS ONLY APPROXIMATE
        while(true)
        {
            Debug.Log("Tick");
            yield return wait;  // wait for a frame, about 1 second from now
        }
    }
}
```

Acabando con una coroutine

A menudo se diseña coroutines para terminar naturalmente cuando se cumplen ciertos objetivos.

```
IEnumerator TickFiveSeconds()
{
    var wait = new WaitForSeconds(1f);
    int counter = 1;
    while(counter < 5)
    {
        Debug.Log("Tick");
        counter++;
        yield return wait;
    }
    Debug.Log("I am done ticking");
}
```

Para detener una coroutine desde "adentro" de la coroutine, no puede simplemente "regresar" como lo haría antes de salir de una función ordinaria. En su lugar, utiliza la `yield break` .

```
IEnumerator ShowExplosions()
{
    ... show basic explosions
    if(player.xp < 100) yield break;
    ... show fancy explosions
}
```

También puede forzar la detención de todas las coroutines lanzadas por el script antes de terminar.

```
void OnDisable()
{
    // Stops all running coroutines
    StopAllCoroutines();
}
```

El método para detener una llamada coroutine *específica* de la persona que llama varía dependiendo de cómo lo inició.

Si empezaste una coroutine por nombre de cadena:

```
StartCoroutine("YourAnimation");
```

luego puede detenerlo llamando a StopCoroutine con el mismo nombre de cadena:

```
StopCoroutine("YourAnimation");
```

Alternativamente, se puede mantener una referencia *a* la `IEnumerator` devuelto por el método de co-rutina, *o* la `Coroutine` objeto devuelto por `StartCoroutine` , y llamar a `StopCoroutine` en cualquiera de los:

```
public class SomeComponent : MonoBehaviour
{
    Coroutine routine;

    void Start () {
        routine = StartCoroutine(YourAnimation());
```

```
    }

    void Update () {
        // later, in response to some input...
        StopCoroutine(routine);
    }

    IEnumerator YourAnimation () { /* ... */ }
}
```

MonoBehaviour métodos que pueden ser los coroutines.

Existen tres métodos de MonoBehaviour que se pueden realizar en las coroutinas.

1. Comienzo()
2. OnBecameVisible ()
3. OnLevelWasLoaded ()

Esto se puede usar para crear, por ejemplo, secuencias de comandos que se ejecutan solo cuando el objeto es visible para una cámara.

```
using UnityEngine;
using System.Collections;

public class RotateObject : MonoBehaviour
{
    IEnumerator OnBecameVisible()
    {
        var tr = GetComponent<Transform>();
        while (true)
        {
            tr.Rotate(new Vector3(0, 180f * Time.deltaTime));
            yield return null;
        }
    }

    void OnBecameInvisible()
    {
        StopAllCoroutines();
    }
}
```

Encadenamiento de coroutines

Los coroutines pueden ceder dentro de sí mismos y esperar a **otros coroutines** .

Por lo tanto, puede encadenar secuencias - "una tras otra".

Esto es muy fácil, y es una técnica básica, básica, en Unity.

Es absolutamente natural en los juegos que ciertas cosas tengan que suceder "en orden". Casi cada "ronda" de un juego comienza con una cierta serie de eventos que suceden, en un espacio de tiempo, en algún orden. Así es como puedes comenzar un juego de carreras de autos:

```
IEnumerator BeginRace()
{
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
}
```

Entonces, cuando llamas BeginRace ...

```
 StartCoroutine(BeginRace());
```

Se ejecutará su rutina de "preparar carrera". (Tal vez, encendiendo algunas luces y ejecutando algo de ruido de la multitud, reajustando las puntuaciones y así sucesivamente.) Cuando termine, ejecutará su secuencia de "cuenta atrás", donde quizás anime una cuenta atrás en la interfaz de usuario. Cuando haya terminado, ejecutará su código de inicio de carrera, donde quizás ejecute efectos de sonido, inicie algunos controladores AI, mueva la cámara de una determinada manera, y así sucesivamente.

Para mayor claridad, entendamos que las tres llamadas.

```
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
```

Deben **estar** ellos mismos **en** una coroutine. Es decir, deben estar en una función del tipo `IEnumerator` . Así que en nuestro ejemplo es `IEnumerator BeginRace` . Por lo tanto, desde el código "normal", inicie esa rutina con la llamada `StartCoroutine` .

```
 StartCoroutine(BeginRace());
```

Para entender mejor el encadenamiento, aquí hay una función que encadena a las coroutinas. Pasas en una serie de coroutines. La función ejecuta tantas corrutinas como pases, en orden, una tras otra.

```
// run various routines, one after the other
IEnumerator OneAfterTheOther( params IEnumerator[] routines )
{
    foreach ( var item in routines )
    {
        while ( item.MoveNext() ) yield return item.Current;
    }

    yield break;
}
```

Así es como llamaría a eso ... digamos que tiene tres funciones. Recordemos que todos deben ser `IEnumerator` :

```
IEnumerator PrepareRace()
{
    // codesay, crowd cheering and camera pan around the stadium
```

```
    yield break;
}

IEnumerator Countdown()
{
    // codesay, animate your countdown on UI
    yield break;
}

IEnumerator StartRace()
{
    // codesay, camera moves and light changes and launch the AIs
    yield break;
}
```

Lo llamarías así

```
StartCoroutine( MultipleRoutines( PrepareRace(), Countdown(), StartRace() ) );
```

o tal vez así

```
IEnumerator[] routines = new IEnumerator[] {
     PrepareRace(),
     Countdown(),
     StartRace() };
StartCoroutine( MultipleRoutines( routines ) );
```

Para repetir, uno de los requisitos más básicos en los juegos es que ciertas cosas suceden una tras otra "en una secuencia" a lo largo del tiempo. Lo logras en Unity de manera muy simple, con

```
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
```

Maneras de ceder

Puedes esperar hasta el siguiente fotograma.

```
yield return null; // wait until sometime in the next frame
```

Puede tener múltiples de estas llamadas en una fila, para simplemente esperar tantos cuadros como desee.

```
//wait for a few frames
yield return null;
yield return null;
```

Espere ***aproximadamente*** n segundos. Es extremadamente importante entender que esto es solo **un tiempo muy aproximado** .

```
yield return new WaitForSeconds(n);
```

Es absolutamente imposible utilizar la llamada "WaitForSeconds" para cualquier forma de sincronización precisa.

A menudo quieres encadenar acciones. Entonces, haz algo, y cuando eso haya terminado, haz algo más, y cuando hayas terminado haz algo más. Para lograrlo, espera otra coroutina:

```
yield return StartCoroutine(coroutine);
```

Comprende que solo puedes llamar a eso desde una coroutine. Asi que:

```
StartCoroutine(Test());
```

Así es como se inicia una corutina a partir de un código "normal".

Luego, dentro de esa corrida corutina:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

Eso imprimirá A, comenzará el proceso largo e **inmediatamente imprimirá B. No** va a esperar a que el largo proceso para terminar. Por otra parte:

```
Debug.Log("A");
yield return StartCoroutine(LongProcess());
Debug.Log("B");
```

Eso imprimirá A, iniciará el proceso largo, **esperará hasta que termine** y luego imprimirá B.

Siempre vale la pena recordar que los coroutines no tienen absolutamente ninguna conexión, de ninguna manera, con los hilos. Con este código:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

es fácil pensar que es "como" iniciar LongProcess en otro subproceso en segundo plano. Pero eso es absolutamente incorrecto. Es sólo una corutina. Los motores de juego están basados en marcos, y las "rutinas" en Unity simplemente te permiten acceder a los marcos.

Es muy fácil esperar a que se complete una solicitud web.

```
void Start() {
    string url = "http://google.com";
    WWW www = new WWW(url);
    StartCoroutine(WaitForRequest(www));
}

IEnumerator WaitForRequest(WWW www) {
    yield return www;
```

```
    if (www.error == null) {
        //use www.data);
    }
    else {
        //use www.error);
    }
}
```

Para completar: En casos muy raros, utiliza la actualización fija en Unity; hay una llamada `WaitForFixedUpdate()` que normalmente nunca se usaría. Hay una llamada específica (`WaitForEndOfFrame()` en la versión actual de Unity) que se usa en ciertas situaciones en relación con la generación de capturas de pantalla durante el desarrollo. (El mecanismo exacto cambia ligeramente a medida que Unity evoluciona, así que busque la información más reciente si es relevante).

Lea Coroutines en línea: https://riptutorial.com/es/unity3d/topic/3415/coroutines

Capítulo 11: Cuaterniones

Sintaxis

- Quaternion.LookRotation (Vector3 adelante [, Vector3 arriba]);
- Quaternion.AngleAxis (ángulos de flotación, vector3 axisOfRotation);
- ángulo de flotación entre = ángulo de cuaternión (rotación de cuaternión1, rotación de cuaternión2);

Examples

Introducción a Quaternion vs Euler

Los ángulos de Euler son "ángulos de grado" como 90, 180, 45, 30 grados. Los cuaterniones difieren de los ángulos de Euler en que representan un punto en una esfera unitaria (el radio es 1 unidad). Puede pensar en esta esfera como una versión 3D del círculo de unidad que aprende en trigonometría. Los cuaterniones difieren de los ángulos de Euler en que usan números imaginarios para definir una rotación 3D.

Si bien esto puede sonar complicado (y podría decirse que lo es), Unity tiene excelentes funciones integradas que le permiten cambiar entre los ángulos y quaterions de Euler, así como funciones para modificar los cuaterniones, sin saber una sola cosa acerca de las matemáticas detrás de ellos.

Conversión entre Euler y Quaternion

```
// Create a quaternion that represents 30 degrees about X, 10 degrees about Y
Quaternion rotation = Quaternion.Euler(30, 10, 0);

// Using a Vector
Vector3 EulerRotation = new Vector3(30, 10, 0);
Quaternion rotation = Quaternion.Euler(EulerRotation);

// Convert a transfroms Quaternion angles to Euler angles
Quaternion quaternionAngles = transform.rotation;
Vector3 eulerAngles = quaternionAngles.eulerAngles;
```

¿Por qué usar un cuaternión?

Los cuaterniones resuelven un problema conocido como bloqueo de cardán. Esto ocurre cuando el eje primario de rotación se convierte en colineal con el eje terciario de rotación. Aquí hay un ejemplo visual @ 2:09

Quaternion Look Rotation

`Quaternion.LookRotation(Vector3 forward [, Vector3 up])` creará una rotación de Quaternion que mira el vector hacia adelante 'hacia abajo' y tiene el eje Y alineado con el vector 'up'. Si no se

especifica el vector hacia arriba, se utilizará Vector3.up.

Gire este objeto de juego para ver un objeto de juego objetivo

```
// Find a game object in the scene named Target
public Transform target = GameObject.Find("Target").GetComponent<Transform>();

// We subtract our position from the target position to create a
// Vector that points from our position to the target position
// If we reverse the order, our rotation would be 180 degrees off.
Vector3 lookVector = target.position - transform.position;
Quaternion rotation = Quaternion.LookRotation(lookVector);
transform.rotation = rotation;
```

Lea Cuaterniones en línea: https://riptutorial.com/es/unity3d/topic/1782/cuaterniones

Capítulo 12: Desarrollo multiplataforma

Examples

Definiciones del compilador

Las definiciones del compilador ejecutan código específico de la plataforma. Usándolos puedes hacer pequeñas diferencias entre varias plataformas.

- Activa los logros de Game Center en dispositivos Apple y Google Play en dispositivos Android.
- Cambie los iconos en los menús (el logotipo de Windows en Windows, Linux Penguin en Linux).
- Posiblemente disponemos de mecánica específica de la plataforma en función de la plataforma.
- Y mucho más...

```
void Update(){

#if UNITY_IPHONE
    //code here is only called when running on iPhone
#endif

#if UNITY_STANDALONE_WIN && !UNITY_EDITOR
    //code here is only ran in a unity game running on windows outside of the editor
#endif

//other code that will be ran regardless of platform

}
```

Una lista completa de las definiciones del compilador de Unity se puede encontrar aquí

Plataforma organizativa de métodos específicos para clases parciales.

Las clases parciales proporcionan una forma limpia de separar la lógica central de sus scripts de los métodos específicos de la plataforma.

Las clases y los métodos parciales están marcados con la palabra clave `partial` . Esto le indica al compilador que deje la clase "abierta" y busque en otros archivos el resto de la implementación.

```
// ExampleClass.cs
using UnityEngine;

public partial class ExampleClass : MonoBehaviour
{
    partial void PlatformSpecificMethod();

    void OnEnable()
    {
```

```
        PlatformSpecificMethod();
    }
}
```

Ahora podemos crear archivos para los scripts específicos de nuestra plataforma que implementan el método parcial. Los métodos parciales pueden tener parámetros (también `ref`) pero deben devolverse `void` .

```
// ExampleClass.Iphone.cs

#if UNITY_IPHONE
using UnityEngine;

public partial class ExampleClass
{
    partial void PlatformSpecificMethod()
    {
        Debug.Log("I am an iPhone");
    }
}
#endif
```

```
// ExampleClass.Android.cs

#if UNITY_ANDROID
using UnityEngine;

public partial class ExampleClass
{
    partial void PlatformSpecificMethod()
    {
        Debug.Log("I am an Android");
    }
}
#endif
```

Si no se implementa un método parcial, el compilador omitirá la llamada.

> Consejo: Este patrón es útil al crear métodos específicos del Editor también.

Lea Desarrollo multiplataforma en línea: https://riptutorial.com/es/unity3d/topic/4816/desarrollo-multiplataforma

Capítulo 13: Encontrar y coleccionar GameObjects

Sintaxis

- Búsqueda de GameObject estática pública (nombre de cadena);
- público estático GameObject FindGameObjectWithTag (etiqueta de cadena);
- GameObject estático público [] FindGameObjectsWithTag (etiqueta de cadena);
- objeto estático público FindObjectOfType (tipo de tipo);
- objeto estático público [] FindObjectsOfType (tipo de tipo);

Observaciones

Qué método utilizar

Tenga cuidado al buscar GameObjects en tiempo de ejecución, ya que esto puede consumir recursos. Especialmente: no ejecute FindObjectOfType o Find in Update, FixedUpdate o más generalmente en un método llamado una o más veces por fotograma.

- Llame a los métodos de ejecución `FindObjectOfType` y `Find` solo cuando sea necesario
- `FindGameObjectWithTag` tiene un rendimiento muy bueno en comparación con otros métodos basados en cadenas. Unity mantiene pestañas separadas en los objetos etiquetados y las consulta en lugar de toda la escena.
- Para GameObjects "estáticos" (como elementos de UI y prefabs) creados en el editor, use la referencia de GameObject serializable en el editor
- Mantenga sus listas de GameObjects en la Lista o Arreglos que usted mismo administra.
- En general, si crea una gran cantidad de GameObjects del mismo tipo, eche un vistazo a la Agrupación de objetos
- Guarda en caché los resultados de búsqueda para evitar ejecutar los métodos de búsqueda costosos una y otra vez.

Yendo mas profundo

Además de los métodos que vienen con Unity, es relativamente fácil diseñar sus propios métodos de búsqueda y recopilación.

- En el caso de `FindObjectsOfType()` , puede hacer que sus scripts guarden una lista de ellos mismos en una colección `static` . Es mucho más rápido iterar una lista de objetos lista que buscar e inspeccionar objetos de la escena.

- O cree un script que almacene sus instancias en un `Dictionary` basado en cadenas, y usted tiene un sistema de etiquetado simple que puede ampliar.

Examples

Buscando por nombre de GameObject

```
var go = GameObject.Find("NameOfTheObject");
```

Pros	Contras
Fácil de usar	El rendimiento se degrada a lo largo del número de gameobjects en escena
	Las cadenas son *referencias débiles* y sospechosas de errores del usuario

Buscando por las etiquetas de GameObject

```
var go = GameObject.FindGameObjectWithTag("Player");
```

Pros	Contras
Posibilidad de buscar tanto objetos individuales como grupos enteros	Las cadenas son referencias débiles y sospechosas de errores del usuario.
Relativamente rápido y eficiente	El código no es portátil, ya que las etiquetas están codificadas en los scripts.

Insertado a scripts en el modo de edición

```
[SerializeField]
GameObject[] gameObjects;
```

Pros	Contras
Gran actuación	La colección de objetos es estática
Codigo portatil	Solo se puede referir a GameObjects desde la misma escena.

Encontrando guiones de GameObjects by MonoBehaviour

```
ExampleScript script = GameObject.FindObjectOfType<ExampleScript>();
GameObject go = script.gameObject;
```

`FindObjectOfType()` devuelve `null` si no se encuentra ninguno.

Pros	Contras
Fuertemente mecanografiado	El rendimiento se degrada a lo largo del número

Pros	Contras
	de gameobjects necesarios para evaluar
Posibilidad de buscar tanto objetos individuales como grupos enteros	

Encuentra GameObjects por nombre de objetos secundarios

```
Transform tr = GetComponent<Transform>().Find("NameOfTheObject");
GameObject go = tr.gameObject;
```

`Find` devoluciones `null` si no se encuentra ninguna

Pros	Contras
Alcance de búsqueda limitado y bien definido.	Las cuerdas son referencias débiles

Lea Encontrar y coleccionar GameObjects en línea:
https://riptutorial.com/es/unity3d/topic/3793/encontrar-y-coleccionar-gameobjects

Capítulo 14: Etiquetas

Introducción

Una etiqueta es una cadena que se puede aplicar para marcar tipos de `GameObject` . De esta manera, facilita la identificación de objetos `GameObject` particulares a través del código.

Se puede aplicar una etiqueta a uno o más objetos del juego, pero un objeto del juego siempre tendrá una sola etiqueta. De forma predeterminada, la etiqueta *"Sin* etiquetar *"* se usa para representar un `GameObject` que no se ha etiquetado intencionalmente.

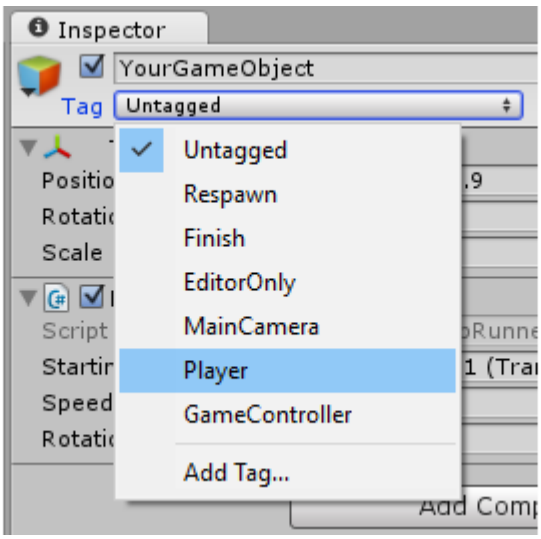
Examples

Creación y aplicación de etiquetas

Las etiquetas se aplican típicamente a través del editor; Sin embargo, también puede aplicar etiquetas a través de script. Cualquier etiqueta personalizada debe crearse a través de la ventana *Etiquetas y Capas* antes de aplicarse a un objeto de juego.

Estableciendo etiquetas en el editor

Con uno o más objetos de juego seleccionados, puede seleccionar una etiqueta del inspector. Los objetos del juego siempre llevarán una sola etiqueta; de forma predeterminada, los objetos del juego se etiquetarán como *"Sin* etiquetar *"* . También puede moverse a la ventana *Etiquetas y capas* seleccionando *"Agregar etiqueta ..."* ; sin embargo, es importante tener en cuenta que esto solo lo lleva a la ventana *Etiquetas y capas* . Cualquier etiqueta que crees *no* se aplicará automáticamente al objeto del juego.



Configuración de etiquetas a través de secuencias de

comandos

Puede cambiar directamente una etiqueta de objetos del juego a través del código. Es importante tener en cuenta que *debe* proporcionar una etiqueta de la lista de etiquetas actuales; Si proporciona una etiqueta que aún no se ha creado, se producirá un error.

Como se detalla en otros ejemplos, el uso de una serie de variables de `static string` en lugar de escribir manualmente cada etiqueta puede garantizar la consistencia y confiabilidad.

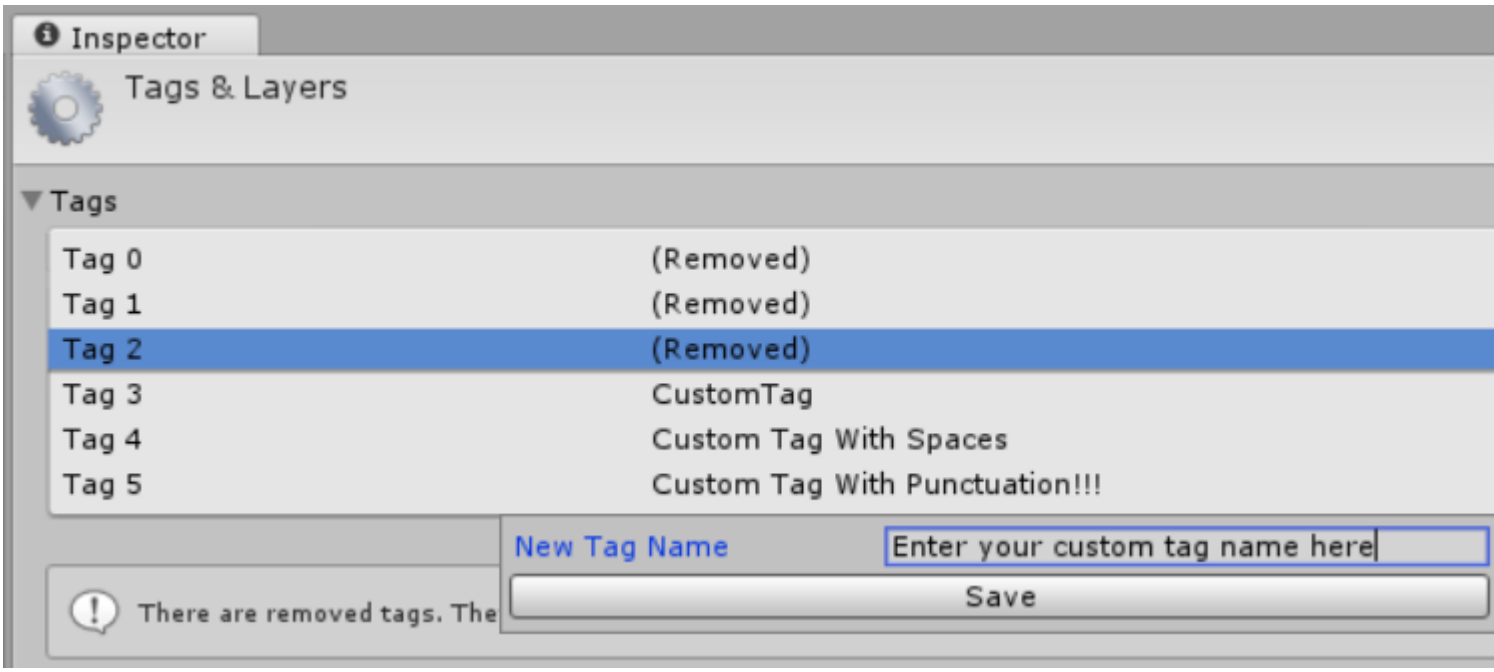
La siguiente secuencia de comandos demuestra cómo podemos cambiar una serie de etiquetas de objetos del juego, utilizando referencias de `static string` para garantizar la coherencia. Tenga en cuenta la suposición de que cada `static string` representa una etiqueta que ya se ha creado en la ventana *Etiquetas y capas* .

```
using UnityEngine;

public class Tagging : MonoBehaviour
{
    static string tagUntagged = "Untagged";
    static string tagPlayer = "Player";
    static string tagEnemy = "Enemy";

    /// <summary>Represents the player character. This game object should
    /// be linked up via the inspector.</summary>
    public GameObject player;
    /// <summary>Represents all the enemy characters. All enemies should
    /// be added to the array via the inspector.</summary>
    public GameObject[] enemy;

    void Start ()
    {
        // We ensure that the game object this script is attached to
        // is left untagged by using the default "Untagged" tag.
        gameObject.tag = tagUntagged;

        // We ensure the player has the player tag.
        player.tag = tagUntagged;

        // We loop through the enemy array to ensure they are all tagged.
        for(int i = 0; i < enemy.Length; i++)
        {
            enemy[i].tag = tagEnemy;
        }
    }
}
```

Creación de etiquetas personalizadas

Independientemente de si establece etiquetas a través del Inspector, o mediante un script, las etiquetas *deben* declararse a través de la ventana *Etiquetas y Capas* antes de usarlas. Puede acceder a esta ventana seleccionando *"Agregar etiquetas ..."* en el menú desplegable de

etiquetas de objetos del juego. Alternativamente, puede encontrar la ventana en **Editar> Configuración del proyecto> Etiquetas y capas** .



Simplemente seleccione el botón `+` , ingrese el nombre deseado y seleccione `Guardar` para crear una etiqueta. Al seleccionar el botón `-` eliminará la etiqueta resaltada en ese momento. Tenga en cuenta que de esta manera, la etiqueta se mostrará inmediatamente como *"(Eliminada)"* , y se eliminará por completo cuando se vuelva a cargar el proyecto.

Seleccionar el engranaje / engranaje desde la parte superior derecha de la ventana le permitirá restablecer todas las opciones personalizadas. Esto eliminará de inmediato todas las etiquetas personalizadas, junto con cualquier capa personalizada que pueda tener en *"Ordenar capas"* y *"Capas"* .

Encontrar GameObjects por etiqueta:

Las etiquetas hacen que sea particularmente fácil localizar objetos específicos del juego. Podemos buscar un solo objeto de juego, o buscar múltiples.

Encontrar un solo objeto de `GameObject`

Podemos usar la función estática `GameObject.FindGameObjectWithTag(string tag)` para buscar objetos individuales del juego. Es importante tener en cuenta que, de esta manera, los objetos del juego no se consultan en ningún orden en particular. Si busca una etiqueta que se utiliza en varios objetos del juego en la escena, esta función no podrá garantizar *qué* objeto del juego se devuelve. Como tal, es más apropiado cuando sabemos que solo *un* objeto del juego usa dicha etiqueta, o cuando no nos preocupa la instancia exacta de `GameObject` que se devuelve.

```
///<summary>We create a static string to allow us consistency.</summary>
string playerTag = "Player"
```

```
///<summary>We can now use the tag to reference our player GameObject.</summary>
GameObject player = GameObject.FindGameObjectWithTag(playerTag);
```

Encontrar una matriz de instancias de `GameObject`

Podemos usar la función estática `GameObject.FindGameObjectsWithTag(string tag)` para buscar *todos los* objetos del juego que usan una etiqueta en particular. Esto es útil cuando queremos iterar a través de un grupo de objetos del juego en particular. Esto también puede ser útil si queremos encontrar un *solo* objeto de juego, pero podemos tener *varios* objetos de juego usando la misma etiqueta. Como no podemos garantizar la instancia exacta devuelto por `GameObject.FindGameObjectWithTag(string tag)` , en lugar debemos recuperar una matriz de todas las posibles `GameObject` instancias con `GameObject.FindGameObjectsWithTag(string tag)` , y analizar más a fondo la matriz resultante para encontrar la instancia somos buscando.

```
///<summary>We create a static string to allow us consistency.</summary>
string enemyTag = "Enemy";

///<summary>We can now use the tag to create an array of all enemy GameObjects.</summary>
GameObject[] enemies = GameObject.FindGameObjectsWithTag(enemyTag );

// We can now freely iterate through our array of enemies
foreach(GameObject enemy in enemies)
{
    // Do something to each enemy (link up a reference, check for damage, etc.)
}
```

Comparando etiquetas

Al comparar dos GameObjects por etiquetas, se debe tener en cuenta que lo siguiente causaría una sobrecarga de Garbage Collector como una cadena que se crea cada vez que:

```
if (go.Tag == "myTag")
{
    //Stuff
}
```

Al realizar esas comparaciones dentro de Update () y la devolución de llamada de Unity (o un bucle) de Unity normal, debe usar este método sin asignación de almacenamiento dinámico:

```
if (go.CompareTag("myTag")
{
    //Stuff
}
```

Además, es más fácil mantener tus etiquetas en una clase estática.

```
public static class Tags
{
    public const string Player = "Player";
```

```
    public const string MyCustomTag = "MyCustomTag";
}
```

Entonces puedes comparar con seguridad

```
if (go.CompareTag(Tags.MyCustomTag)
{
    //Stuff
}
```

De esta manera, las cadenas de etiquetas se generan en el momento de la compilación, y usted limita las implicaciones de los errores de ortografía.

Al igual que mantener las etiquetas en una clase estática, también es posible almacenarlas en una enumeración:

```
public enum Tags
{
    Player, Ennemies, MyCustomTag;
}
```

y luego puedes compararlo usando el método enum `toString()` :

```
if (go.CompareTag(Tags.MyCustomTag.toString())
{
    //Stuff
}
```

Lea Etiquetas en línea: https://riptutorial.com/es/unity3d/topic/5534/etiquetas

Capítulo 15: Extendiendo el Editor

Sintaxis

- [MenuItem (string itemName)]
- [MenuItem (string itemName, bool isValidateFunction)]
- [MenuItem (string itemName, bool isValidateFunction, int prioridad)]
- [ContextMenu (nombre de cadena)]
- [ContextMenuItem (nombre de cadena, función de cadena)]
- [DrawGizmo (GizmoType gizmo)]
- [DrawGizmo (GizmoType gizmo, Type drawnGizmoType)]

Parámetros

Parámetro	Detalles
MenuCommand	MenuCommand se usa para extraer el contexto de un MenuItem
MenuCommand.context	El objeto que es el objetivo del comando de menú.
MenuCommand.userData	Un int para pasar información personalizada a un elemento de menú

Examples

Inspector personalizado

El uso de un inspector personalizado le permite cambiar la forma en que se dibuja un script en el Inspector. A veces desea agregar información adicional en el inspector para su script que no es posible hacer con un cajón de propiedades personalizadas.

A continuación se muestra un ejemplo simple de un objeto personalizado que al usar un inspector personalizado puede mostrar información más útil.

```
using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif

public class InspectorExample : MonoBehaviour {

    public int Level;
    public float BaseDamage;

    public float DamageBonus {
        get {
            return Level / 100f * 50;
```

```
        }
    }

    public float ActualDamage {
        get {
            return BaseDamage + DamageBonus;
        }
    }
}

#if UNITY_EDITOR
[CustomEditor( typeof( InspectorExample ) )]
public class CustomInspector : Editor {

    public override void OnInspectorGUI() {
        base.OnInspectorGUI();

        var ie = (InspectorExample)target;

        EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );
        EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
    }
}
#endif
```

Primero definimos nuestro comportamiento personalizado con algunos campos.

```
public class InspectorExample : MonoBehaviour {
    public int Level;
    public float BaseDamage;
}
```

Los campos que se muestran arriba se dibujan automáticamente (sin el inspector personalizado) cuando está viendo el script en la ventana del Inspector.

```
public float DamageBonus {
    get {
        return Level / 100f * 50;
    }
}

public float ActualDamage {
    get {
        return BaseDamage + DamageBonus;
    }
}
```

Estas propiedades no son dibujadas automáticamente por Unity. Para mostrar estas propiedades en la vista Inspector, tenemos que utilizar nuestro Inspector personalizado.

Primero tenemos que definir a nuestro inspector personalizado como este.

```
[CustomEditor( typeof( InspectorExample ) )]
public class CustomInspector : Editor {
```

El inspector personalizado tiene que derivar del *Editor* y necesita el atributo *CustomEditor* . El

parámetro del atributo es el tipo de objeto para el que debe utilizarse este inspector personalizado.

El siguiente es el método OnInspectorGUI. Este método se llama siempre que el script se muestra en la ventana del inspector.

```
public override void OnInspectorGUI() {
    base.OnInspectorGUI();
}
```

Hacemos una llamada a *base.OnInspectorGUI ()* para permitir que Unity maneje los otros campos que están en el script. Si no lo llamáramos, tendríamos que hacer más trabajo nosotros mismos.

A continuación están nuestras propiedades personalizadas que queremos mostrar

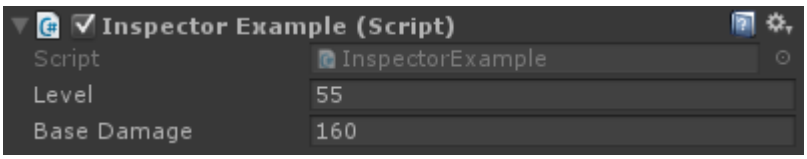
```
var ie = (InspectorExample)target;

EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );
EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
```

Tenemos que crear una variable temporal que contenga el destino convertido en nuestro tipo personalizado (el destino está disponible porque derivamos del Editor).

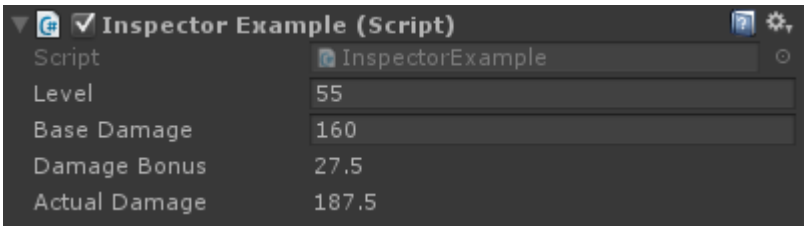
A continuación, podemos decidir cómo dibujar nuestras propiedades, en este caso, dos campos de trabajo son suficientes, ya que solo queremos mostrar los valores y no poder editarlos.

Resultado

antes de



Después



Cajón de propiedad personalizada

A veces, tiene objetos personalizados que contienen datos pero no se derivan de MonoBehaviour. Agregar estos objetos como un campo en una clase que es MonoBehaviour no tendrá ningún efecto visual a menos que escriba su propio cajón de propiedades personalizadas para el tipo de objeto.

A continuación se muestra un ejemplo simple de un objeto personalizado, agregado a MonoBehaviour, y un cajón de propiedades personalizadas para el objeto personalizado.

```
public enum Gender {
    Male,
    Female,
    Other
}

// Needs the Serializable attribute otherwise the CustomPropertyDrawer wont be used
[Serializable]
public class UserInfo {
    public string Name;
    public int Age;
    public Gender Gender;
}

// The class that you can attach to a GameObject
public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UInfo;
}

[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {

    public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
        // The 6 comes from extra spacing between the fields (2px each)
        return EditorGUIUtility.singleLineHeight * 4 + 6;
    }

    public override void OnGUI( Rect position, SerializedProperty property, GUIContent label )
{
        EditorGUI.BeginProperty( position, label, property );

        EditorGUI.LabelField( position, label );

        var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
        var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
        var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );

        EditorGUI.indentLevel++;

        EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
        EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
        EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

        EditorGUI.indentLevel--;

        EditorGUI.EndProperty();
    }
}
```

En primer lugar definimos el objeto personalizado con todos sus requisitos. Solo una clase simple que describe a un usuario. Esta clase se usa en nuestra clase PropertyDrawerExample que podemos agregar a un GameObject.

```
public enum Gender {
    Male,
```

```
    Female,
    Other
}

[Serializable]
public class UserInfo {
    public string Name;
    public int Age;
    public Gender Gender;
}

public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UInfo;
}
```

> La clase personalizada necesita el atributo Serializable; de lo contrario, no se utilizará CustomPropertyDrawer

El siguiente es el CustomPropertyDrawer

Primero tenemos que definir una clase que derive de PropertyDrawer. La definición de clase también necesita el atributo CustomPropertyDrawer. El parámetro pasado es el tipo de objeto para el que desea utilizar este cajón.

```
[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {
```

A continuación, anulamos la función GetPropertyHeight. Esto nos permite definir una altura personalizada para nuestra propiedad. En este caso, sabemos que nuestra propiedad tendrá cuatro partes: etiqueta, nombre, edad y género. Por lo tanto, usamos *EditorGUIUtility.singleLineHeight * 4* , agregamos otros 6 píxeles porque queremos espaciar cada campo con dos píxeles entre ellos.

```
public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
    return EditorGUIUtility.singleLineHeight * 4 + 6;
}
```

El siguiente es el método de OnGUI real. Comenzamos con *EditorGUI.BeginProperty ([...])* y *terminamos* la función con *EditorGUI.EndProperty ()* . Hacemos esto de modo que si esta propiedad formara parte de una prefabulación, la lógica de reemplazo de prefabricación real funcionaría para todo lo que se encuentre entre esos dos métodos.

```
public override void OnGUI( Rect position, SerializedProperty property, GUIContent label ) {
    EditorGUI.BeginProperty( position, label, property );
```

Después de eso mostramos una etiqueta que contiene el nombre del campo y ya definimos los rectángulos para nuestros campos.

```
EditorGUI.LabelField( position, label );

var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
```

```
var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );
```

> Cada campo está espaciado por 16 + 2 píxeles y la altura es 16 (que es lo mismo que EditorGUIUtility.singleLineHeight)

A continuación, sangramos la interfaz de usuario con una pestaña para un diseño un poco más agradable, mostramos las propiedades, *deshicimos* la GUI y *terminamos* con *EditorGUI.EndProperty* .

```
EditorGUI.indentLevel++;

EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

EditorGUI.indentLevel--;

EditorGUI.EndProperty();
```

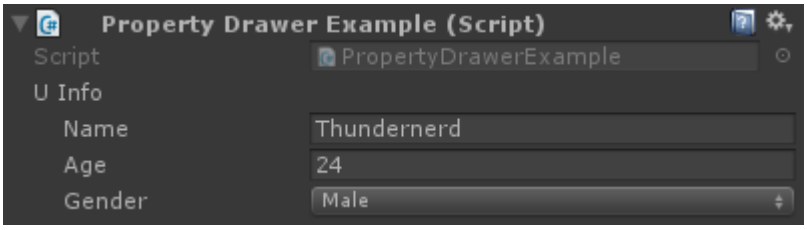
Mostramos los campos usando *EditorGUI.PropertyField* que requiere un rectángulo para la posición y una propiedad Serialized para que la propiedad se muestre. *Adquirimos* la propiedad llamando a *FindPropertyRelative ("...")* en la propiedad aprobada en la función *OnGUI* . Tenga en cuenta que estas propiedades no distinguen entre mayúsculas y minúsculas y no se pueden encontrar.

> Para este ejemplo no estoy guardando el retorno de propiedades de property.FindPropertyRelative ("..."). Debe guardar estos en campos privados en la clase para evitar llamadas innecesarias

Resultado

antes de

Después



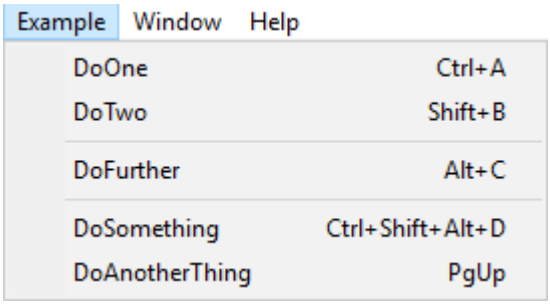
Elementos de menú

Los elementos del menú son una excelente manera de agregar acciones personalizadas al editor. Puede agregar elementos de menú a la barra de menús, tenerlos como clics de contexto en componentes específicos, o incluso como clics de contexto en los campos de sus scripts.

A continuación se muestra un ejemplo de cómo puede aplicar elementos de menú.

```
public class MenuItemsExample : MonoBehaviour {

    [MenuItem( "Example/DoSomething %#&d" )]
    private static void DoSomething() {
        // Execute some code
    }

    [MenuItem( "Example/DoAnotherThing", true )]
    private static bool DoAnotherThingValidator() {
        return Selection.gameObjects.Length > 0;
    }

    [MenuItem( "Example/DoAnotherThing _PGUP", false )]
    private static void DoAnotherThing() {
        // Execute some code
    }

    [MenuItem( "Example/DoOne %a", false, 1 )]
    private static void DoOne() {
        // Execute some code
    }

    [MenuItem( "Example/DoTwo #b", false, 2 )]
    private static void DoTwo() {
        // Execute some code
    }

    [MenuItem( "Example/DoFurther &c", false, 13 )]
    private static void DoFurther() {
        // Execute some code
    }

    [MenuItem( "CONTEXT/Camera/DoCameraThing" )]
    private static void DoCameraThing( MenuCommand cmd ) {
        // Execute some code
    }

    [ContextMenu( "ContextSomething" )]
    private void ContentSomething() {
        // Execute some code
    }

    [ContextMenuItem( "Reset", "ResetDate" )]
    [ContextMenuItem( "Set to Now", "SetDateToNow" )]
    public string Date = "";

    public void ResetDate() {
        Date = "";
    }

    public void SetDateToNow() {
        Date = DateTime.Now.ToString();
    }
}
```

Que se parece a esto



Vamos a repasar el elemento básico del menú. Como puede ver a continuación, debe definir una función estática con un atributo *MenuItem* , que pasa una cadena como título para el elemento del menú. Puede poner su elemento de menú en varios niveles de profundidad agregando un / en el nombre.

```
[MenuItem( "Example/DoSomething %#&d" )]
private static void DoSomething() {
    // Execute some code
}
```

> No puede tener un elemento de menú en el nivel superior. ¡Sus elementos de menú deben estar en un submenú!

Los caracteres especiales al final del nombre de MenuItem son para las teclas de acceso directo, no son un requisito.

Hay caracteres especiales que puedes usar para tus teclas de acceso directo, estos son:

- % - Ctrl en Windows, Cmd en OS X
- # - Cambio
- & - Alt

Eso significa que el acceso directo *% # & d* significa ctrl + shift + alt + D en Windows, y cmd + shift + alt + D en OS X.

Si desea usar un atajo sin ninguna tecla especial, por ejemplo, solo la tecla 'D', puede anteponer el carácter _ (guión bajo) a la tecla de atajo que desea usar.

Hay algunas otras teclas especiales que son compatibles, que son:

- IZQUIERDA, DERECHA, ARRIBA, ABAJO - para las teclas de flecha
- F1..F12 - para las teclas de función
- INICIO, FIN, PGUP, PGDN - para las teclas de navegación

> Las teclas de acceso directo deben separarse de cualquier otro texto con un espacio

A continuación están los elementos del menú validador. Los elementos del menú del validador permiten que los elementos del menú se deshabiliten (en gris, no se puede hacer clic) cuando no se cumple la condición. Un ejemplo de esto podría ser que su elemento de menú actúa en la selección actual de GameObjects, que puede verificar en el elemento de menú del validador.

```
[MenuItem( "Example/DoAnotherThing", true )]
private static bool DoAnotherThingValidator() {
    return Selection.gameObjects.Length > 0;
}

[MenuItem( "Example/DoAnotherThing _PGUP", false )]
private static void DoAnotherThing() {
    // Execute some code
}
```

Para que un elemento del menú del validador funcione, necesita crear dos funciones estáticas, tanto con el atributo MenuItem como con el mismo nombre (la tecla de acceso directo no importa). La diferencia entre ellos es que los está marcando como una función de validación o no al pasar un parámetro booleano.

También puede definir el orden de los elementos del menú agregando una prioridad. La prioridad está definida por un entero que se pasa como el tercer parámetro. Cuanto más pequeño sea el número, más arriba en la lista, más grande será el número más bajo en la lista. Puede agregar un separador entre dos elementos del menú asegurándose de que haya al menos 10 dígitos entre la prioridad de los elementos del menú.

```
[MenuItem( "Example/DoOne %a", false, 1 )]
private static void DoOne() {
    // Execute some code
}

[MenuItem( "Example/DoTwo #b", false, 2 )]
private static void DoTwo() {
    // Execute some code
}

[MenuItem( "Example/DoFurther &c", false, 13 )]
private static void DoFurther() {
    // Execute some code
}
```

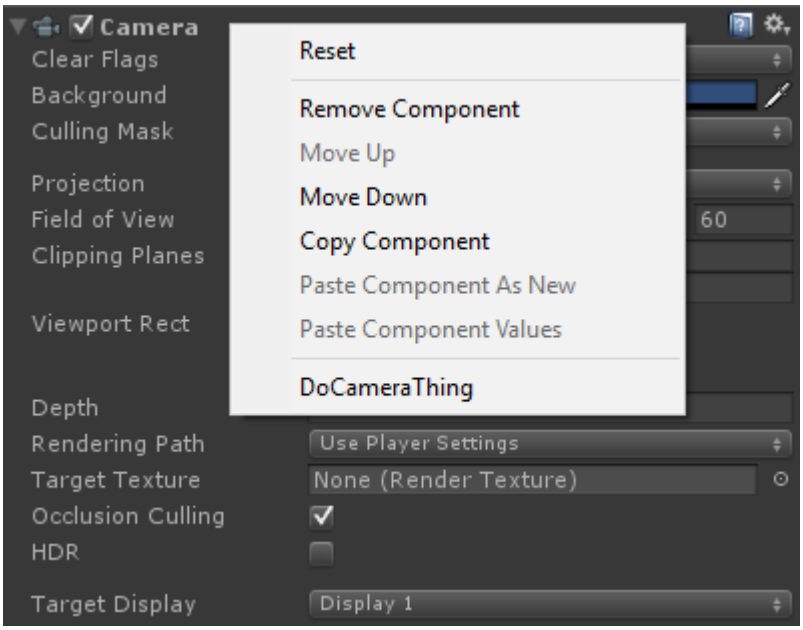
> Si tiene una lista de menú que tiene una combinación de elementos priorizados y no priorizados, los no priorizados se separarán de los elementos priorizados.

Lo siguiente es agregar un elemento de menú al menú contextual de un componente ya existente. Debe iniciar el nombre de MenuItem con CONTEXT (distingue entre mayúsculas y minúsculas) y hacer que su función tome un parámetro MenuCommand.

El siguiente fragmento de código agregará un elemento del menú contextual al componente Cámara.

```
[MenuItem( "CONTEXT/Camera/DoCameraThing" )]
private static void DoCameraThing( MenuCommand cmd ) {
    // Execute some code
}
```

Que se parece a esto

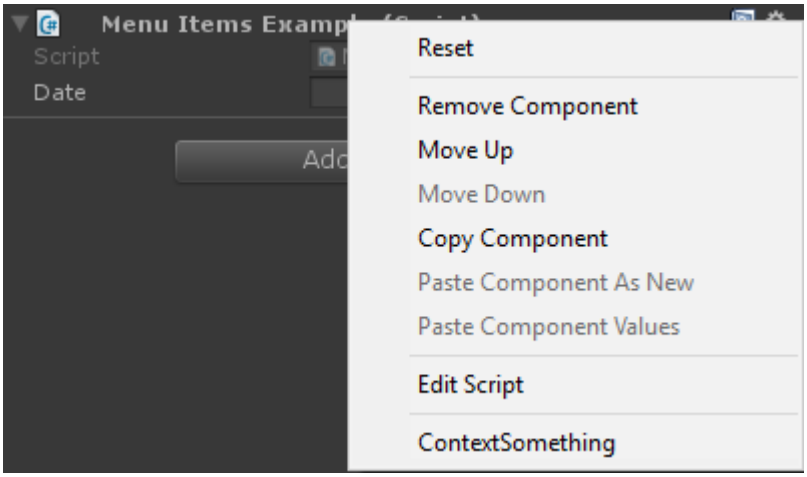


El parámetro MenuCommand le da acceso al valor del componente y cualquier dato de usuario que se envíe con él.

También puede agregar un elemento del menú contextual a sus propios componentes utilizando el atributo ContextMenu. Este atributo solo toma un nombre, ninguna validación o prioridad, y tiene que ser parte de un método no estático.

```
[ContextMenu( "ContextSomething" )]
private void ContentSomething() {
    // Execute some code
}
```

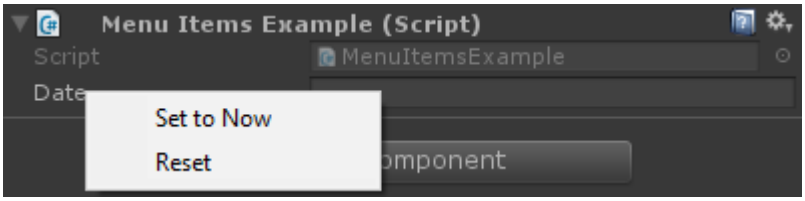
Que se parece a esto



También puede agregar elementos del menú de contexto a los campos en su propio componente. Estos elementos del menú aparecerán cuando haga clic contextual en el campo al que pertenecen y pueda ejecutar los métodos que haya definido en ese componente. De esta manera puede agregar, por ejemplo, valores predeterminados, o la fecha actual, como se muestra a continuación.

```
[ContextMenuItem( "Reset", "ResetDate" )]
[ContextMenuItem( "Set to Now", "SetDateToNow" )]
public string Date = "";

public void ResetDate() {
    Date = "";
}

public void SetDateToNow() {
    Date = DateTime.Now.ToString();
}
```

Que se parece a esto



Gizmos

Los gizmos se utilizan para dibujar formas en la vista de escena. Puede utilizar estas formas para obtener información adicional sobre sus GameObjects, por ejemplo, el frustum que tienen o el rango de detección.

A continuación hay dos ejemplos de cómo hacer esto.

Ejemplo uno

Este ejemplo utiliza los *métodos OnDrawGizmos* y *OnDrawGizmosSelected* (magic).

```
public class GizmoExample : MonoBehaviour {

    public float GetDetectionRadius() {
        return 12.5f;
    }

    public float GetFOV() {
        return 25f;
    }

    public float GetMaxRange() {
        return 6.5f;
    }

    public float GetMinRange() {
        return 0;
    }

    public float GetAspect() {
        return 2.5f;
    }
```

```
    public void OnDrawGizmos() {
        var gizmoMatrix = Gizmos.matrix;
        var gizmoColor = Gizmos.color;

        Gizmos.matrix = Matrix4x4.TRS( transform.position, transform.rotation,
transform.lossyScale );
        Gizmos.color = Color.red;
        Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect()
);

        Gizmos.matrix = gizmoMatrix;
        Gizmos.color = gizmoColor;
    }

    public void OnDrawGizmosSelected() {
        Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
    }
}
```

En este ejemplo tenemos dos métodos para dibujar artilugios, uno que dibuja cuando el objeto está activo (OnDrawGizmos) y uno para cuando el objeto está seleccionado en la jerarquía (OnDrawGizmosSelected).

```
public void OnDrawGizmos() {
    var gizmoMatrix = Gizmos.matrix;
    var gizmoColor = Gizmos.color;

    Gizmos.matrix = Matrix4x4.TRS( transform.position, transform.rotation,
transform.lossyScale );
    Gizmos.color = Color.red;
    Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect() );

    Gizmos.matrix = gizmoMatrix;
    Gizmos.color = gizmoColor;
}
```

Primero guardamos la matriz de gizmo y el color porque lo vamos a cambiar y queremos revertirlo cuando hayamos terminado para no afectar a ningún otro dibujo de gizmo.

A continuación, queremos dibujar el frustum que tiene nuestro objeto, sin embargo, necesitamos cambiar la matriz de Gizmos para que coincida con la posición, la rotación y la escala. También establecemos el color de Gizmos en rojo para enfatizar el frustum. Cuando se hace esto, podemos llamar a *Gizmos.DrawFrustum* para dibujar el frustum en la vista de escena.

Cuando hayamos terminado de dibujar lo que queremos dibujar, restablecemos la matriz y el color de Gizmos.

```
public void OnDrawGizmosSelected() {
    Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
}
```

También queremos dibujar un rango de detección cuando seleccionamos nuestro GameObject. Esto se hace a través de la clase *Handles* ya que la clase *Gizmos* no tiene ningún método para discos.

Usando esta forma de dibujar los resultados de gizmos en la salida que se muestra a continuación.

Ejemplo dos

Este ejemplo utiliza el atributo *DrawGizmo* .

```
public class GizmoDrawerExample {

    [DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]
    public static void DrawGizmo( GizmoExample obj, GizmoType type ) {
        var gizmoMatrix = Gizmos.matrix;
        var gizmoColor = Gizmos.color;

        Gizmos.matrix = Matrix4x4.TRS( obj.transform.position, obj.transform.rotation,
obj.transform.lossyScale );
        Gizmos.color = Color.red;
        Gizmos.DrawFrustum( Vector3.zero, obj.GetFOV(), obj.GetMaxRange(), obj.GetMinRange(),
obj.GetAspect() );

        Gizmos.matrix = gizmoMatrix;
        Gizmos.color = gizmoColor;

        if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {
            Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius()
);
        }
    }
}
```

De esta manera le permite separar las llamadas de gizmo de su script. La mayoría de esto usa el mismo código que el otro ejemplo, excepto por dos cosas.

```
[DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]
public static void DrawGizmo( GizmoExample obj, GizmoType type ) {
```

Debe usar el atributo DrawGizmo, que toma la enumeración GizmoType como primer parámetro y un Tipo como segundo parámetro. El Tipo debe ser el tipo que desea utilizar para dibujar el gizmo.

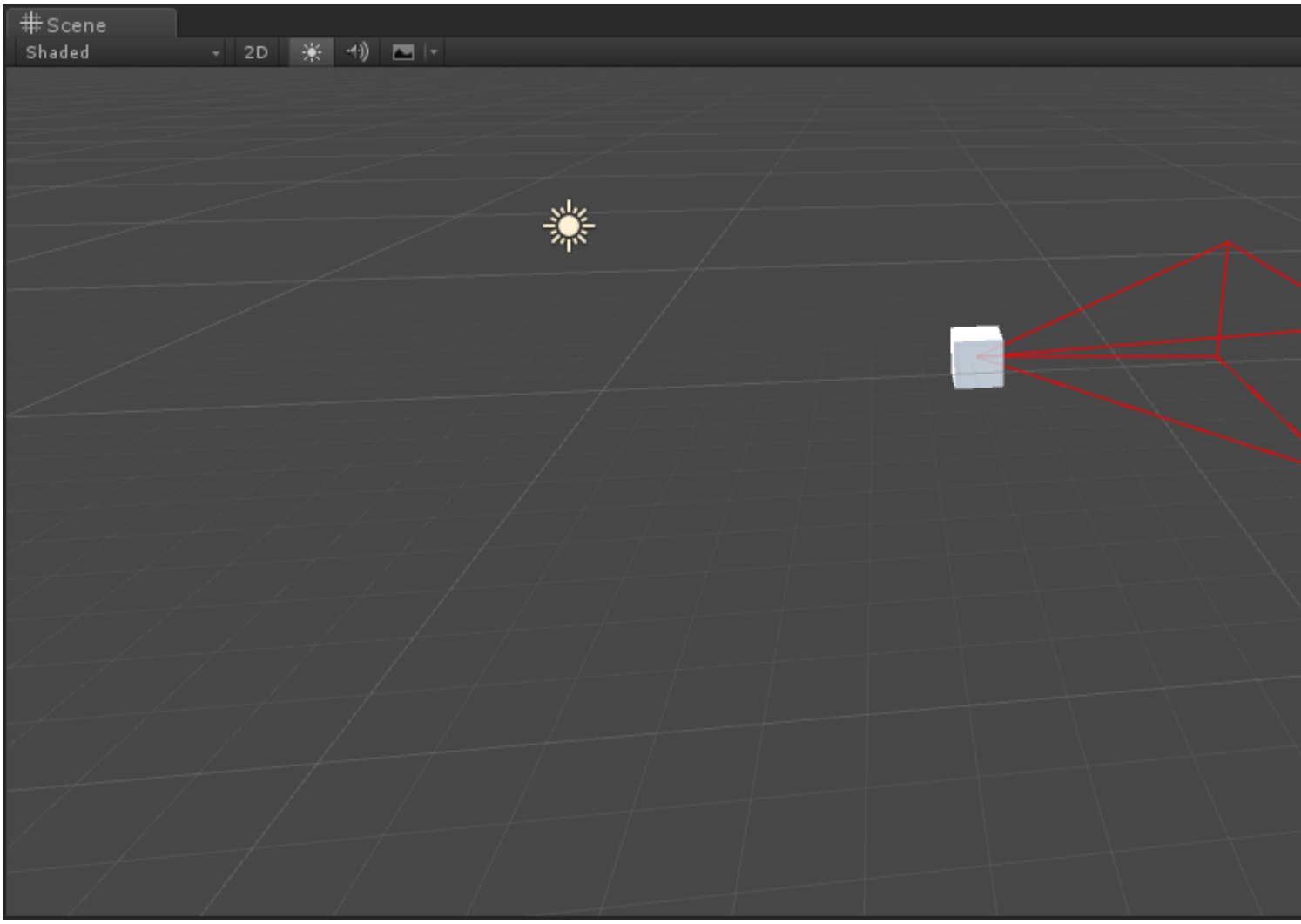
El método para dibujar el gizmo debe ser estático, público o no público, y se puede nombrar como se desee. El primer parámetro es el tipo, que debe coincidir con el tipo pasado como el segundo parámetro en el atributo, y el segundo parámetro es la enumeración GizmoType que describe el estado actual de su objeto.

```
if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {
    Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius() );
}
```

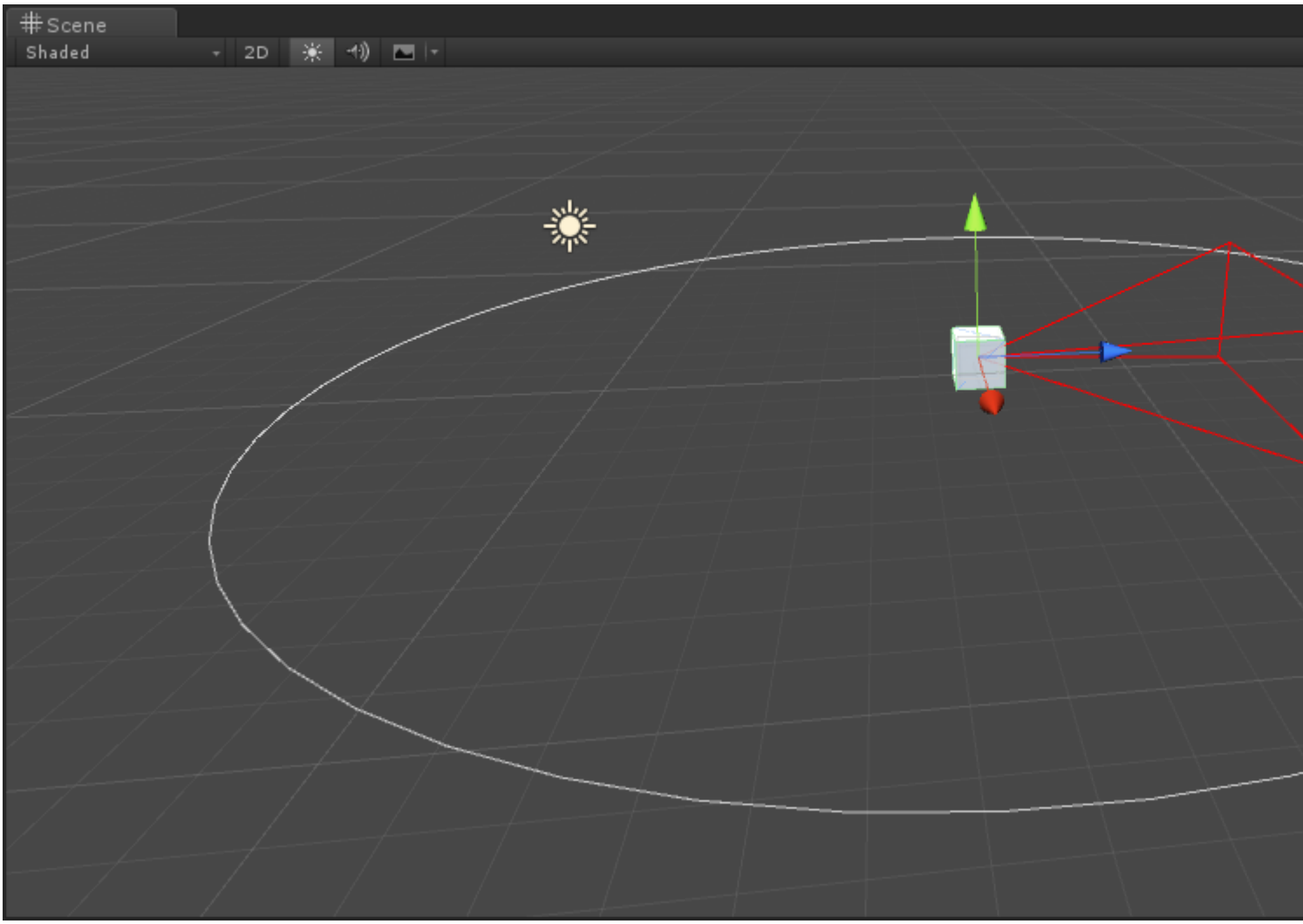
La otra diferencia es que para verificar cuál es el GizmoType del objeto, debe hacer un AND en el parámetro y el tipo que desea.

Resultado

No seleccionado



Seleccionado



Ventana del editor

¿Por qué una ventana de editor?

Como puede haber visto, puede hacer muchas cosas en un inspector personalizado (si no sabe qué es un inspector personalizado, consulte el ejemplo aquí: http://www.riptutorial.com/unity3d/topic/2506 / extender-el-editor . Pero en un momento es posible que desee implementar un panel de configuración o una paleta de recursos personalizada. En esos casos, va a utilizar una ventana de Editor . La interfaz de usuario de Unity se compone de Editor de Windows; puede abrirlos (generalmente a través de la barra superior), tabúelos, etc.

Crear un Editor de Windows básico

Ejemplo simple

Crear una ventana de editor personalizada es bastante simple. Todo lo que necesita hacer es extender la clase EditorWindow y usar los métodos Init () y OnGUI (). Aquí hay un ejemplo simple:

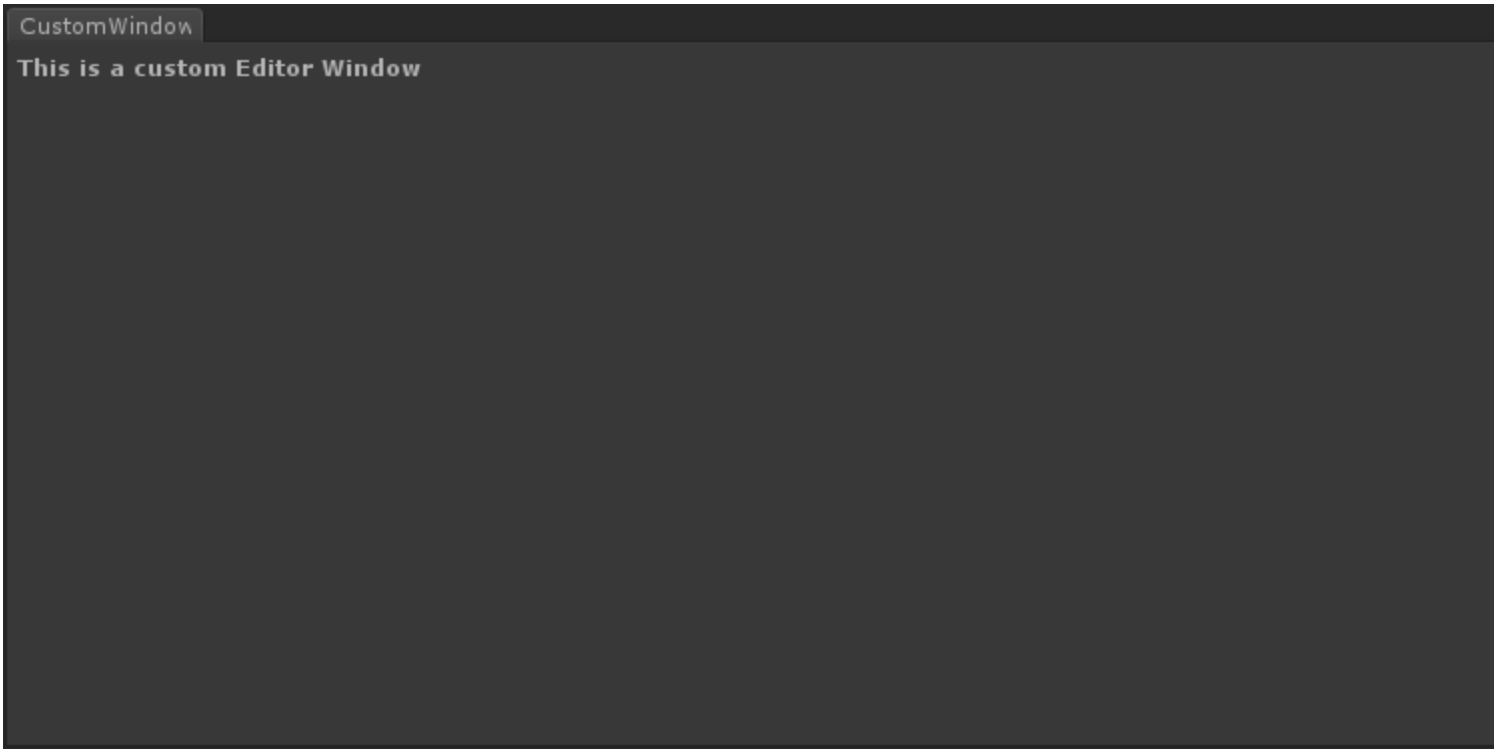
```
using UnityEngine;
using UnityEditor;
```

```
public class CustomWindow : EditorWindow
{
    // Add menu named "Custom Window" to the Window menu
    [MenuItem("Window/Custom Window")]
    static void Init()
    {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
        window.Show();
    }

    void OnGUI()
    {
        GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);
    }
}
```

Los 3 puntos importantes son:

1. No te olvides de extender EditorWindow
2. Utilice el Init () como se proporciona en el ejemplo. EditorWindow.GetWindow está comprobando si ya se ha creado una CustomWindow. Si no, creará una nueva instancia. Al usar esto, se asegura de no tener varias instancias de su ventana al mismo tiempo
3. Use OnGUI () como de costumbre para mostrar información en su ventana

El resultado final se verá así:



Yendo mas profundo

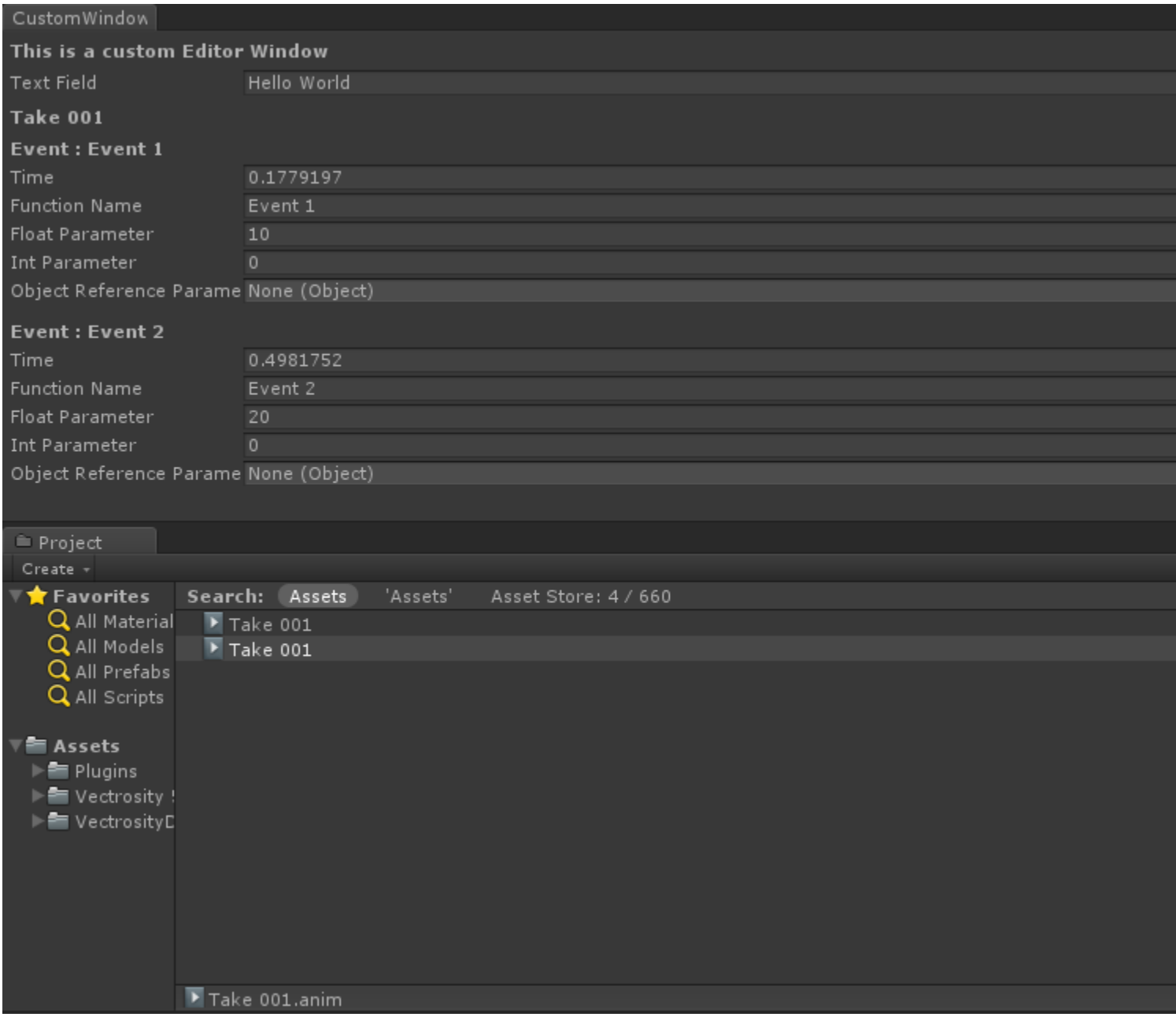
Por supuesto, probablemente querrá administrar o modificar algunos activos usando esta ventana de Editor. Aquí hay un ejemplo que usa la clase de Selección (para obtener la Selección activa) y la modificación de las propiedades del activo seleccionado a través de SerializedObject y

SerializedProperty .

```
    using System.Linq;
    using UnityEngine;
    using UnityEditor;

    public class CustomWindow : EditorWindow
    {
        private AnimationClip _animationClip;
        private SerializedObject _serializedClip;
        private SerializedProperty _events;

        private string _text = "Hello World";

        // Add menu named "Custom Window" to the Window menu
        [MenuItem("Window/Custom Window")]
        static void Init()
        {
            // Get existing open window or if none, make a new one:
            CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
            window.Show();
        }

        void OnGUI()
        {
            GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);

            // You can use EditorGUI, EditorGUILayout and GUILayout classes to display
anything you want
            // A TextField example
            _text = EditorGUILayout.TextField("Text Field", _text);

            // Note that you can modify an asset or a gameobject using an EditorWindow. Here
is a quick example with an AnimationClip asset
            // The _animationClip, _serializedClip and _events are set in OnSelectionChange()

            if (_animationClip == null || _serializedClip == null || _events == null) return;

            // We can modify our serializedClip like we would do in a Custom Inspector. For
example we can grab its events and display their information

            GUILayout.Label(_animationClip.name, EditorStyles.boldLabel);

            for (var i = 0; i < _events.arraySize; i++)
            {
                EditorGUILayout.BeginVertical();

                EditorGUILayout.LabelField(
                    "Event : " +
_events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName").stringValue,
                    EditorStyles.boldLabel);

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("time"),
true,
                    GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName"),

                    true, GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("floatParameter"),
```

```
                    true, GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("intParameter"),

                    true, GUILayout.ExpandWidth(true));
                EditorGUILayout.PropertyField(

_events.GetArrayElementAtIndex(i).FindPropertyRelative("objectReferenceParameter"), true,
                    GUILayout.ExpandWidth(true));

                EditorGUILayout.Separator();
                EditorGUILayout.EndVertical();
            }

            // Of course we need to Apply the modified properties. We don't our changes won't
be saved
            _serializedClip.ApplyModifiedProperties();
        }

        /// This Message is triggered when the user selection in the editor changes. That's
when we should tell our Window to Repaint() if the user selected another AnimationClip
        private void OnSelectionChange()
        {
            _animationClip =
                Selection.GetFiltered(typeof(AnimationClip),
SelectionMode.Assets).FirstOrDefault() as AnimationClip;
            if (_animationClip == null) return;

            _serializedClip = new SerializedObject(_animationClip);
            _events = _serializedClip.FindProperty("m_Events");
            Repaint();
        }
    }
```

Aquí está el resultado:



Temas avanzados

Puede hacer algunas cosas realmente avanzadas en el editor, y la clase EditorWindow es perfecta para mostrar gran cantidad de información. La mayoría de los activos avanzados en Unity Asset Store (como NodeCanvas o PlayMaker) usan EditorWindow para mostrar vistas personalizadas.

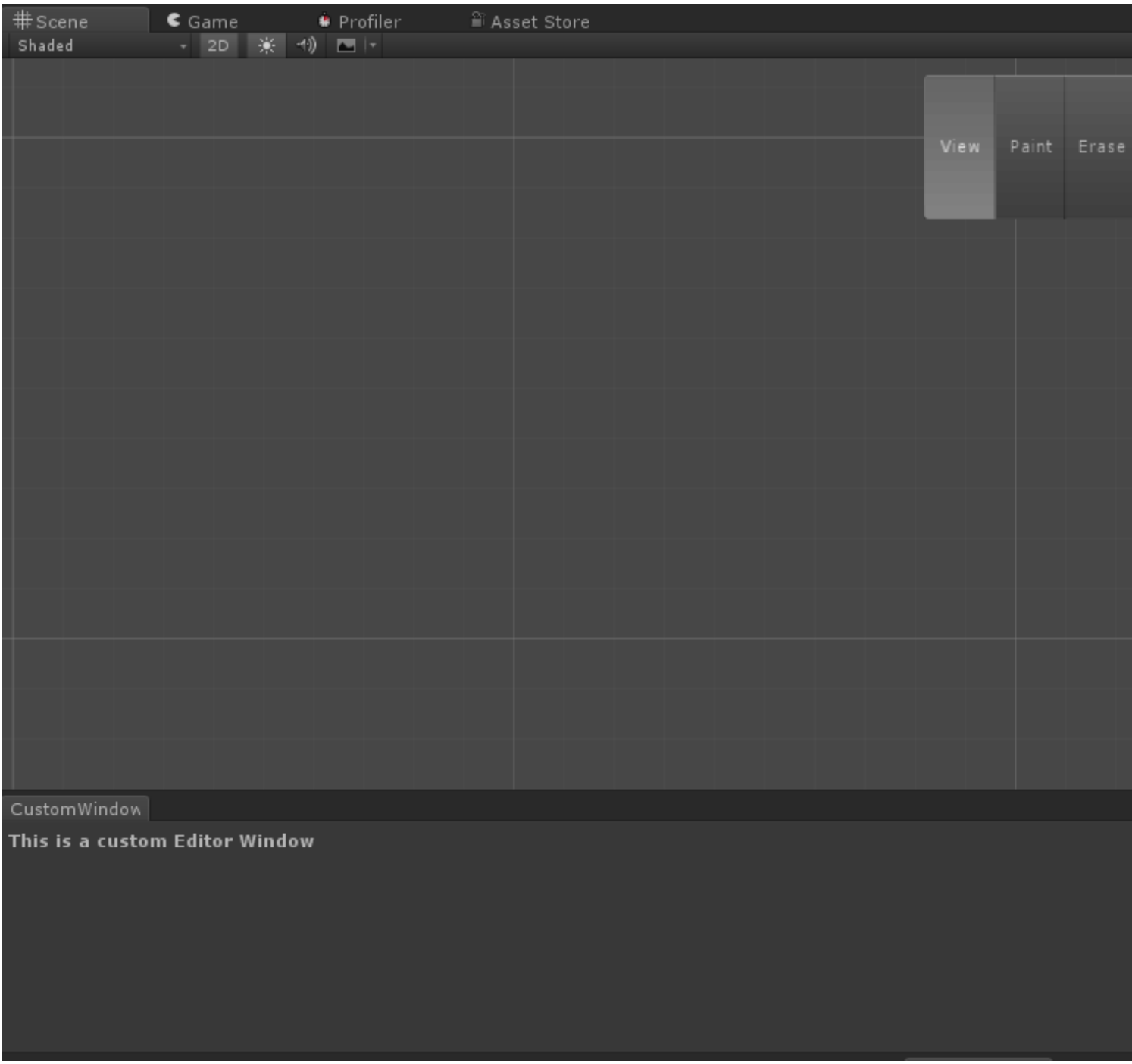
Dibujando en el SceneView

Una cosa interesante que hacer con un EditorWindow es mostrar información directamente en su SceneView. De esta manera, puede crear un editor de mapas / mundos totalmente personalizado, por ejemplo, utilizando su EditorWindow personalizado como una paleta de activos y escuchando los clics en SceneView para crear una instancia de nuevos objetos. Aquí hay un ejemplo :

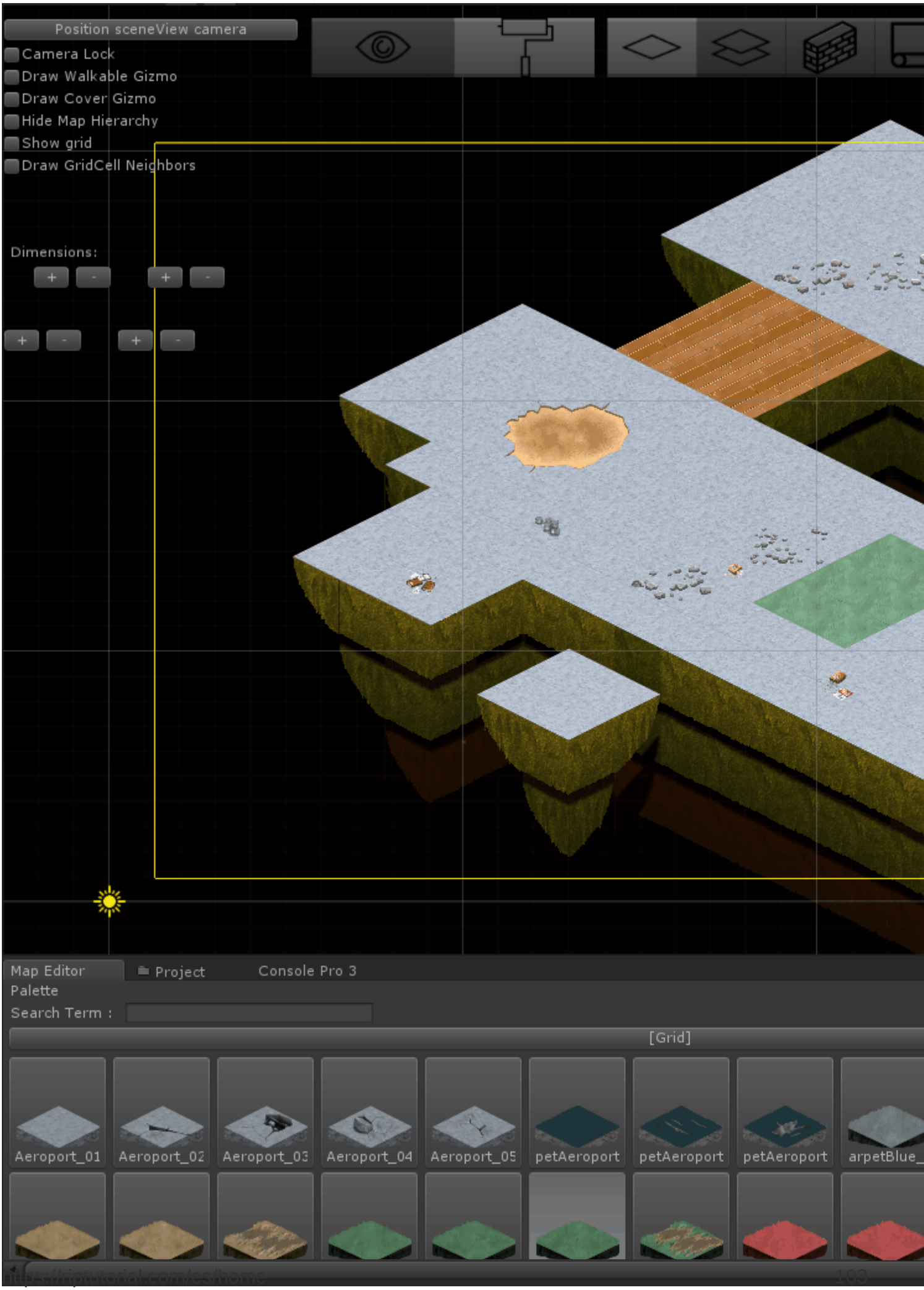
```
using UnityEngine;
```

```
using System;
using UnityEditor;

public class CustomWindow : EditorWindow {

    private enum Mode {
        View = 0,
        Paint = 1,
        Erase = 2
    }

    private Mode CurrentMode = Mode.View;

    [MenuItem ("Window/Custom Window")]
    static void Init () {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow)EditorWindow.GetWindow (typeof (CustomWindow));
        window.Show();
    }

    void OnGUI () {
        GUILayout.Label ("This is a custom Editor Window", EditorStyles.boldLabel);
    }

    void OnEnable() {
        SceneView.onSceneGUIDelegate = SceneViewGUI;
        if (SceneView.lastActiveSceneView) SceneView.lastActiveSceneView.Repaint();
    }

    void SceneViewGUI(SceneView sceneView) {
        Handles.BeginGUI();
        // We define the toolbars' rects here
        var ToolBarRect = new Rect((SceneView.lastActiveSceneView.camera.pixelRect.width / 6),
10, (SceneView.lastActiveSceneView.camera.pixelRect.width * 4 / 6) ,
SceneView.lastActiveSceneView.camera.pixelRect.height / 5);
        GUILayout.BeginArea(ToolBarRect);
        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
         CurrentMode = (Mode) GUILayout.Toolbar(
            (int) CurrentMode,
            Enum.GetNames(typeof(Mode)),
            GUILayout.Height(ToolBarRect.height));
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();
        GUILayout.EndArea();
        Handles.EndGUI();
    }
}
```

Esto mostrará la barra de herramientas directamente en tu SceneView



Aquí tiene un vistazo rápido de lo lejos que puede llegar:


Position sceneView camera
Camera Lock
Draw Walkable Gizmo
Draw Cover Gizmo
Hide Map Hierarchy
Show grid
Draw GridCell Neighbors
Dimensions:
Map Editor
Project
Console Pro 3
Palette
Search Term :
[Grid]
Aeroport_01
Aeroport_02
Aeroport_03
Aeroport_04
Aeroport_05
petAeroport
petAeroport
petAeroport
arpetBlue_

https://riptutorial.com/es/unity3d/topic/2506/extendiendo-el-editor

Capítulo 16: Física

Examples

Cuerpos rígidos

Visión general

El componente Rigidbody le da a GameObject una *presencia física* en la escena en el sentido de que es capaz de responder a las fuerzas. Puedes aplicar fuerzas directamente a GameObject o permitir que reaccione a fuerzas externas como la gravedad u otro Rigidbody que lo golpee.

Añadiendo un componente de Rigidbody

Puede agregar un Rigidbody haciendo clic en **Componente> Física> Rigidbody**

Moviendo un objeto Rigidbody

Se recomienda que si aplica un Rigidbody a un GameObject, use fuerzas o torque para moverlo en lugar de manipular su Transform. Use los `AddForce()` o `AddTorque()` para esto:

```
// Add a force to the order of myForce in the forward direction of the Transform.
GetComponent<Rigidbody>().AddForce(transform.forward * myForce);

// Add torque about the Y axis to the order of myTurn.
GetComponent<Rigidbody>().AddTorque(transform.up * torque * myTurn);
```

Masa

Puedes alterar la masa de un objeto de juego de Rigidbody para afectar cómo reacciona con otros cuerpos y fuerzas de Rigidbody. Una masa más alta significa que GameObject tendrá más influencia sobre otros GameObjects basados en la física, y requerirá una mayor fuerza para moverse. Los objetos de diferente masa caerán a la misma velocidad si tienen los mismos valores de arrastre. Para alterar la masa en el código:

```
GetComponent<Rigidbody>().mass = 1000;
```

Arrastrar

Cuanto mayor sea el valor de arrastre, más se ralentizará un objeto mientras se mueve. Piénsalo como una fuerza opuesta. Para alterar el arrastre en el código:

```
GetComponent<Rigidbody>().drag = 10;
```

---

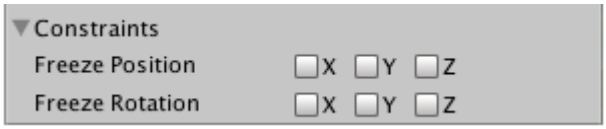
es cinemático

Si marca un Rigidbody como **Cinemático,** entonces no puede ser afectado por otras fuerzas, pero puede afectar a otros GameObjects. Para alterar en el código:

```
GetComponent<Rigidbody>().isKinematic = true;
```

Restricciones

También es posible agregar restricciones a cada eje para congelar la posición o rotación del Rigidbody en el espacio local. El valor predeterminado es `RigidbodyConstraints.None` como se muestra aquí:



Un ejemplo de restricciones en el código:

```
// Freeze rotation on all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeRotation

// Freeze position on all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePosition

// Freeze rotation and motion an all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeAll
```

Puede utilizar el operador bit a bit o `|` para combinar múltiples restricciones como tal:

```
// Allow rotation on X and Y axes and motion on Y and Z axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePositionZ |
    RigidbodyConstraints.FreezeRotationX;
```

Colisiones

Si quieres un GameObject con un Rigidbody en él para responder a las colisiones, también necesitarás agregarle un colisionador. Los tipos de colisionador son:

- Box Collider
- Colisionador de esfera
- Colisionador de cápsulas
- Colisionador de ruedas
- Colisionador de malla

Si aplica más de un colisionador a un GameObject, lo llamamos un colisionador compuesto.

Puede convertir a Collider en un **Trigger** para utilizar los `OnTriggerEnter()` , `OnTriggerStay()` y `OnTriggerExit()` . Un colisionador desencadenante no reaccionará físicamente a las colisiones, otros GameObjects simplemente lo atravesarán. Son útiles para detectar cuándo otro GameObject está en un área determinada o no, por ejemplo, cuando recopilamos un elemento, es posible que deseamos poder ejecutarlo pero detectar cuándo sucede esto.

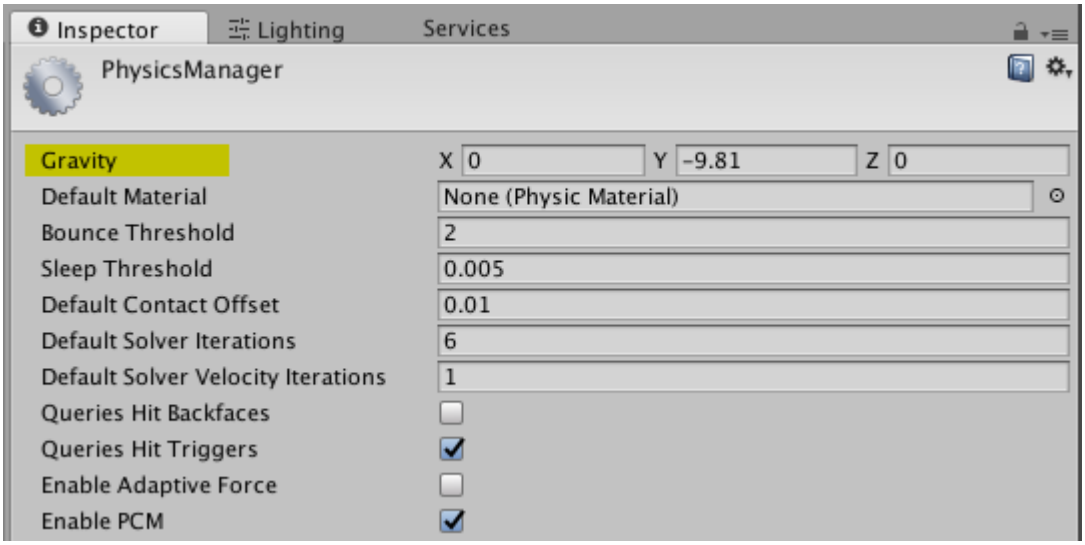
Gravedad en Cuerpo Rigido

La propiedad `useGravity` de un `RigidBody` controla si la gravedad la afecta o no. Si se establece en `false` `RigidBody` se comportará como si estuviera en el espacio exterior (sin que se le aplique una fuerza constante en alguna dirección).

```
GetComponent<RigidBody>().useGravity = false;
```

Es muy útil en las situaciones en las que necesita todas las demás propiedades de `RigidBody` excepto el movimiento controlado por la gravedad.

Cuando está habilitado, `RigidBody` se verá afectado por una fuerza gravitacional, configurada en `Physics Settings` :



La gravedad se define en unidades de mundo por segundo al cuadrado, y se ingresa aquí como un vector tridimensional: lo que significa que con los ajustes en la imagen de ejemplo, todos los `RigidBodies` con la propiedad `useGravity` establecida en `True` experimentarán una fuerza de 9.81

unidades mundiales por segundo *por segundo* en la dirección hacia abajo (como valores Y negativos en el sistema de coordenadas de Unity apuntan hacia abajo).

Lea Física en línea: https://riptutorial.com/es/unity3d/topic/3680/fisica

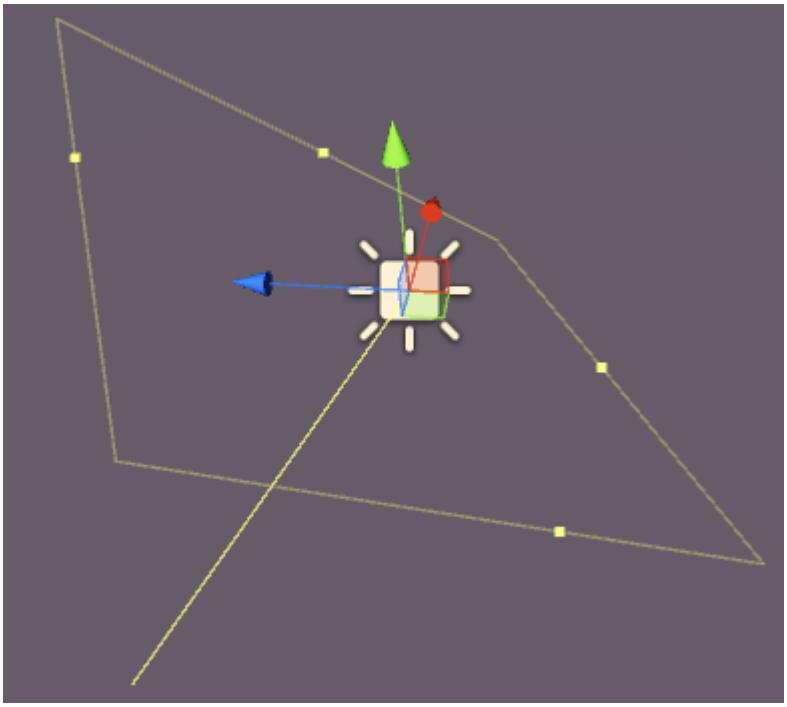
Capítulo 17: Iluminación de la unidad

Examples

Tipos de luz

Luz de área

La luz se emite a través de la superficie de un área rectangular. Solo están horneados, lo que significa que no podrás ver el efecto hasta que cocines la escena.

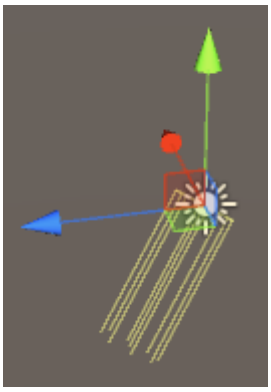


Las luces de área tienen las siguientes propiedades:

- **Ancho** - Ancho del área de luz.
- **Altura** - Altura del área de luz.
- **Color** - Asigna el color de la luz.
- **Intensidad** : cuán poderosa es la luz de 0 a 8.
- **Intensidad de rebote** : cuán poderosa es la luz *indirecta* de 0 a 8.
- **Dibujar halo** : dibujará un halo alrededor de la luz.
- **Destello** : le permite asignar un efecto de destello a la luz.
- **Modo de procesamiento** : automático, importante, no importante.
- **Máscara de eliminación** : le permite iluminar selectivamente partes de una escena.

Luz direccional

Las luces direccionales emiten luz en una sola dirección (como el sol). No importa en qué lugar de la escena se coloca el GameObject real, ya que la luz está "en todas partes". La intensidad de la luz no disminuye como los otros tipos de luz.

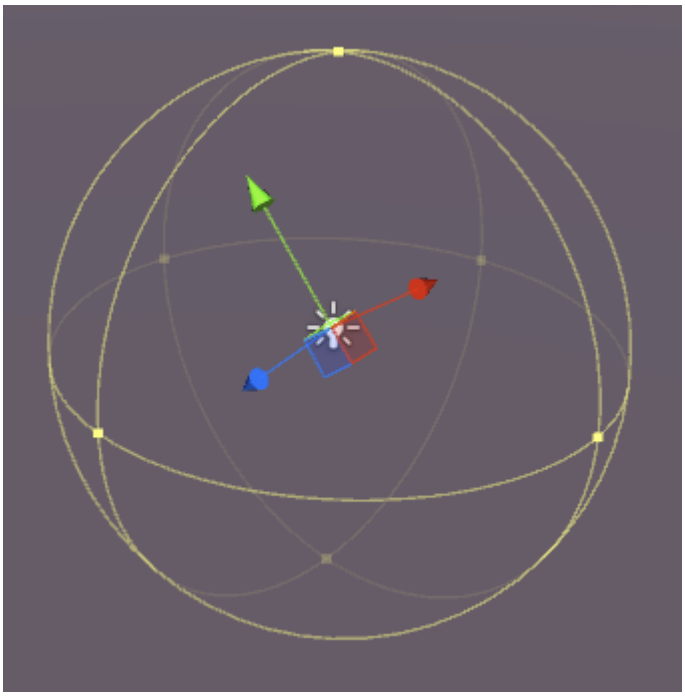


Una luz direccional tiene las siguientes propiedades:

- **Hornear** - En tiempo real, al horno o mixto.
- **Color** - Asigna el color de la luz.
- **Intensidad** : cuán poderosa es la luz de 0 a 8.
- **Intensidad de rebote** : cuán poderosa es la luz *indirecta* de 0 a 8.
- **Tipo de sombra** : sin sombras, sombras duras o sombras suaves.
- **Cookie** - Le permite asignar una cookie para la luz.
- **Tamaño de cookie** : el tamaño de la cookie asignada.
- **Dibujar halo** : dibujará un halo alrededor de la luz.
- **Destello** : le permite asignar un efecto de destello a la luz.
- **Modo de procesamiento** : automático, importante, no importante.
- **Máscara de eliminación** : le permite iluminar selectivamente partes de una escena.

Luz puntual

Una luz puntual emite luz desde un punto en el espacio en todas las direcciones. Cuanto más lejos del punto de origen, menos intensa es la luz.

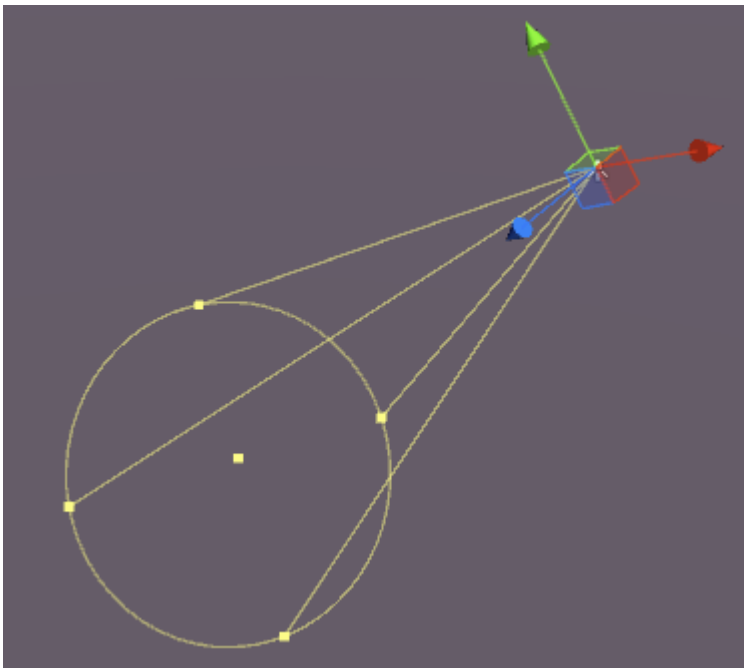


Las luces de punto tienen las siguientes propiedades:

- **Hornear** - En tiempo real, al horno o mixto.
- **Rango** : la distancia desde el punto donde la luz ya no alcanza.
- **Color** - Asigna el color de la luz.
- **Intensidad** : cuán poderosa es la luz de 0 a 8.
- **Intensidad de rebote** : cuán poderosa es la luz *indirecta* de 0 a 8.
- **Tipo de sombra** : sin sombras, sombras duras o sombras suaves.
- **Cookie** - Le permite asignar una cookie para la luz.
- **Dibujar halo** : dibujará un halo alrededor de la luz.
- **Destello** : le permite asignar un efecto de destello a la luz.
- **Modo de procesamiento** : automático, importante, no importante.
- **Máscara de eliminación** : le permite iluminar selectivamente partes de una escena.

Destacar

Una luz puntual es muy parecida a una luz puntual, pero la emisión está restringida a un ángulo. El resultado es un "cono" de luz, útil para los faros de los automóviles o los reflectores.



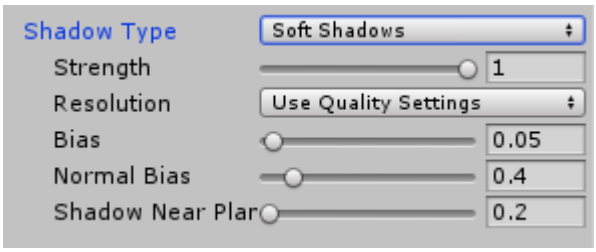
Spot Lights tiene las siguientes propiedades:

- **Hornear** - En tiempo real, al horno o mixto.
- **Rango** : la distancia desde el punto donde la luz ya no alcanza.
- **Ángulo puntual** - El ángulo de emisión de luz.
- **Color** - Asigna el color de la luz.
- **Intensidad** : cuán poderosa es la luz de 0 a 8.
- **Intensidad de rebote** : cuán poderosa es la luz *indirecta* de 0 a 8.
- **Tipo de sombra** : sin sombras, sombras duras o sombras suaves.
- **Cookie** - Le permite asignar una cookie para la luz.
- **Dibujar halo** : dibujará un halo alrededor de la luz.
- **Destello** : le permite asignar un efecto de destello a la luz.
- **Modo de procesamiento** : automático, importante, no importante.
- **Máscara de eliminación** : le permite iluminar selectivamente partes de una escena.

Nota sobre las sombras

Si selecciona Sombras duras o Suaves, las siguientes opciones estarán disponibles en el inspector:

- **Fuerza** : cuán oscuras son las sombras de 0 a 1.
- **Resolución** : cómo son las sombras detalladas.
- **Sesgo** : el grado en que las superficies de proyección de sombra se alejan de la luz.
- **Sesgo normal** : el grado en que las superficies de proyección de sombra se empujan hacia adentro a lo largo de sus normales.
- **Sombra cerca del avión** - 0.1 - 10.



Emisión

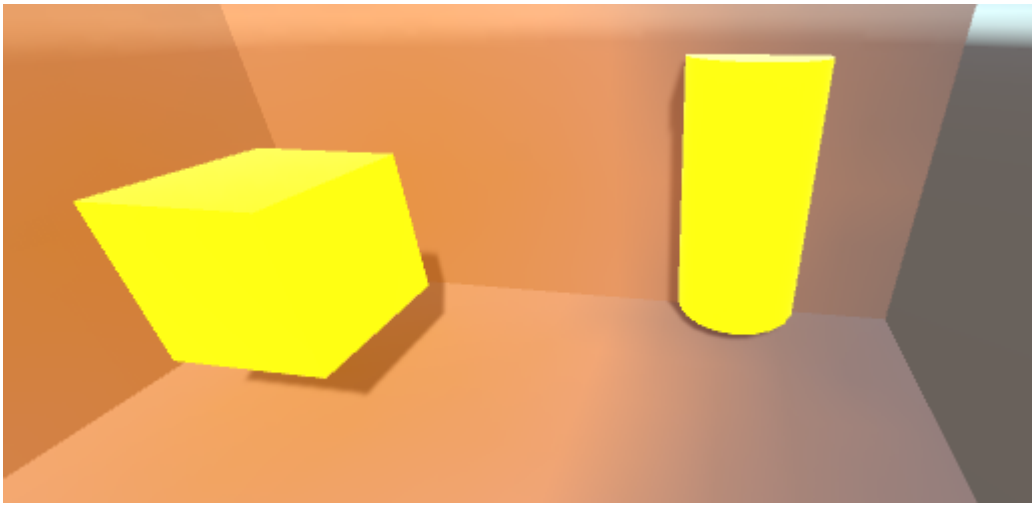
La emisión es cuando una superficie (o más bien un material) emite luz. En el panel de inspección para un material en un objeto estático que utiliza el sombreador estándar, hay una propiedad de emisión:

Si cambia esta propiedad a un valor superior al valor predeterminado de 0, puede establecer el color de emisión o asignar un **mapa de emisión** al material. Cualquier textura asignada a esta ranura permitirá que la emisión utilice sus propios colores.

También hay una opción de iluminación global que le permite establecer si la emisión se agrega a objetos estáticos cercanos o no:

- **Al horno** : la emisión se incluirá en la escena.
- **Tiempo real** - La emisión afectará a objetos dinámicos.
- **Ninguno** - La emisión no afectará a objetos cercanos.

Si el objeto *no* está configurado como estático, el efecto seguirá haciendo que el objeto parezca que "brille" pero no se emite ninguna luz. El cubo aquí es estático, el cilindro no es:



Puedes configurar el color de emisión en código como este:

```
Renderer renderer = GetComponent<Renderer>();
Material mat = renderer.material;
mat.SetColor("_EmissionColor", Color.yellow);
```

La luz emitida caerá a una velocidad cuadrática y solo se mostrará contra los materiales estáticos en la escena.

Lea Iluminación de la unidad en línea: https://riptutorial.com/es/unity3d/topic/7884/iluminacion-de-la-unidad

Capítulo 18: Implementación de la clase MonoBehaviour.

Examples

No hay métodos anulados

La razón por la que no tiene que anular `Awake` , `Start` , `Update` y otro método es porque no son métodos virtuales definidos en una clase base.

La primera vez que se accede a su script, el tiempo de ejecución de scripting busca en el script para ver si se han definido algunos métodos. Si lo son, esa información se almacena en caché y los métodos se agregan a su lista respectiva. Estas listas simplemente se repiten en diferentes momentos.

La razón por la que estos métodos no son virtuales se debe al rendimiento. Si todos los scripts tuvieran `Awake` , `Start` , `OnEnable` , `OnDisable` , `Update` , `LateUpdate` y `FixedUpdate` , entonces todos estos se agregarían a sus listas, lo que significaría que todos estos métodos se ejecutarán. Normalmente, esto no sería un gran problema, sin embargo, todas estas llamadas de métodos son desde el lado nativo (C ++) al lado administrado (C #), lo que conlleva un costo de rendimiento.

Ahora imagine esto, todos estos métodos están en sus listas y algunos / la mayoría de ellos incluso pueden no tener un cuerpo de método real. Esto significaría que hay una gran cantidad de rendimiento desperdiciado en los métodos de llamada que ni siquiera hacen nada. Para evitar esto, Unity optó por dejar de usar métodos virtuales e hizo un sistema de mensajería que se asegura de que solo se llame a estos métodos cuando realmente estén definidos, ahorrando llamadas de método innecesarias.

Puede leer más sobre el tema en un blog de Unity aquí: 10000 Update () Llamadas y más sobre IL2CPP aquí: Una introducción a los internos de IL2CPP

Lea Implementación de la clase MonoBehaviour. en línea: https://riptutorial.com/es/unity3d/topic/2304/implementacion-de-la-clase-monobehaviour-

Capítulo 19: Importadores y (Post) Procesadores

Sintaxis

- AssetPostprocessor.OnPreprocessTexture ()

Observaciones

Use `String.Contains()` para procesar solo los activos que tienen una cadena dada en sus rutas de activos.

```
if (assetPath.Contains("ProcessThisFolder"))
{
    // Process asset
}
```

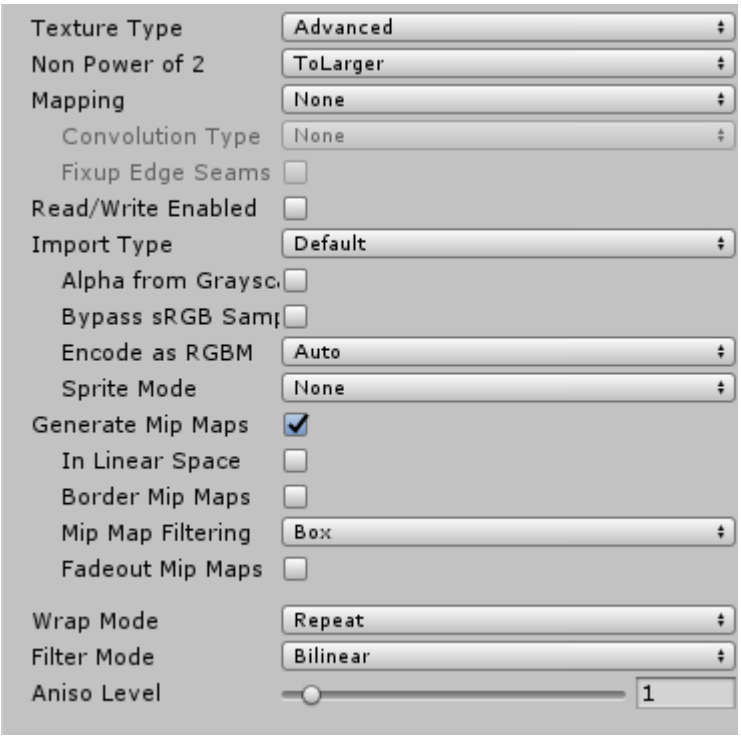
Examples

Postprocesador de texturas

Cree el archivo `TexturePostProcessor.cs` en cualquier lugar de la carpeta de **Activos** :

```
using UnityEngine;
using UnityEditor;

public class TexturePostProcessor : AssetPostprocessor
{
    void OnPostprocessTexture(Texture2D texture)
    {
        TextureImporter importer = assetImporter as TextureImporter;
        importer.anisoLevel = 1;
        importer.filterMode = FilterMode.Bilinear;
        importer.mipmapEnabled = true;
        importer.npotScale = TextureImporterNPOTScale.ToLarger;
        importer.textureType = TextureImporterType.Advanced;
    }
}
```

Ahora, cada vez que Unity importa una textura tendrá los siguientes parámetros:



Si usa el posprocesador, no puede cambiar los parámetros de textura manipulando la **configuración de importación** en el editor.
Cuando presiona el botón **Aplicar,** la textura volverá a importarse y el código del postprocesador se ejecutará nuevamente.

Un importador básico

Suponga que tiene un archivo personalizado para el que desea crear un importador. Podría ser un archivo .xls o lo que sea. En este caso, vamos a utilizar un archivo JSON porque es fácil, pero vamos a elegir una extensión personalizada para que sea más fácil saber qué archivos son los nuestros.

Asumamos que el formato del archivo JSON es

```
{
  "someValue": 123,
  "someOtherValue": 456.297,
  "someBoolValue": true,
  "someStringValue": "this is a string",
}
```

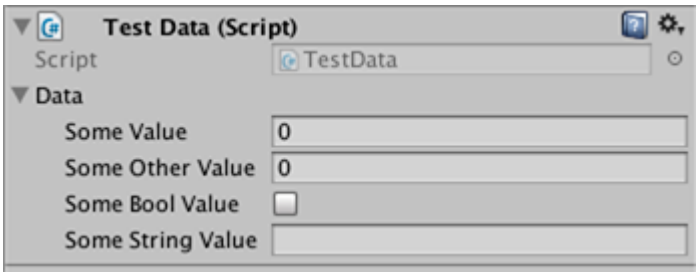
`Example.test` eso como `Example.test` algún lugar *fuera* de los activos por ahora.

A continuación, haga un `MonoBehaviour` con una clase personalizada solo para los datos. La clase personalizada es únicamente para facilitar la deserialización de JSON. NO tiene que usar una clase personalizada, pero hace que este ejemplo sea más corto. Guardaremos esto en `TestData.cs`

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

```
public class TestData : MonoBehaviour {

    [System.Serializable]
    public class Data {
        public int someValue = 0;
        public float someOtherValue = 0.0f;
        public bool someBoolValue = false;
        public string someStringValue = "";
    }

    public Data data = new Data();
}
```

Si agregaras manualmente ese script a un GameObject verías algo como



Luego haga una carpeta de `Editor` algún lugar debajo de `Assets` . Puedo estar en cualquier nivel. Dentro de la carpeta Editor, haga un archivo `TestDataAssetPostprocessor.cs` y póngalo en él.

```
using UnityEditor;
using UnityEngine;
using System.Collections;

public class TestDataAssetPostprocessor : AssetPostprocessor
{
    const string s_extension = ".test";

    // NOTE: Paths start with "Assets/"
    static bool IsFileWeCareAbout(string path)
    {
        return System.IO.Path.GetExtension(path).Equals(
           s_extension,
           System.StringComparison.Ordinal);
    }

    static void HandleAddedOrChangedFile(string path)
    {
        string text = System.IO.File.ReadAllText(path);
        // should we check for error if the file can't be parsed?
        TestData.Data newData = JsonUtility.FromJson<TestData.Data>(text);

        string prefabPath = path + ".prefab";
        // Get the existing prefab
        GameObject existingPrefab =
            AssetDatabase.LoadAssetAtPath(prefabPath, typeof(Object)) as GameObject;
        if (!existingPrefab)
        {
            // If no prefab exists make one
            GameObject newGameObject = new GameObject();
            newGameObject.AddComponent<TestData>();
            PrefabUtility.CreatePrefab(prefabPath,
```

```
                                    newGameObject,
                                    ReplacePrefabOptions.Default);
            GameObject.DestroyImmediate(newGameObject);
            existingPrefab =
                AssetDatabase.LoadAssetAtPath(prefabPath, typeof(Object)) as GameObject;
        }

        TestData testData = existingPrefab.GetComponent<TestData>();
        if (testData != null)
        {
            testData.data = newData;
            EditorUtility.SetDirty(existingPrefab);
        }
    }

    static void HandleRemovedFile(string path)
    {
        // Decide what you want to do here. If the source file is removed
        // do you want to delete the prefab? Maybe ask if you'd like to
        // remove the prefab?
        // NOTE: Because you might get many calls (like you deleted a
        // subfolder full of .test files you might want to get all the
        // filenames and ask all at once ("delete all these prefabs?").
    }

    static void OnPostprocessAllAssets (string[] importedAssets, string[] deletedAssets,
string[] movedAssets, string[] movedFromAssetPaths)
    {
        foreach (var path in importedAssets)
        {
            if (IsFileWeCareAbout(path))
            {
                HandleAddedOrChangedFile(path);
            }
        }

        foreach (var path in deletedAssets)
        {
            if (IsFileWeCareAbout(path))
            {
                HandleRemovedFile(path);
            }
        }

        for (var ii = 0; ii < movedAssets.Length; ++ii)
        {
            string srcStr = movedFromAssetPaths[ii];
            string dstStr = movedAssets[ii];

            // the source was moved, let's move the corresponding prefab
            // NOTE: We don't handle the case if there already being
            // a prefab of the same name at the destination
            string srcPrefabPath = srcStr + ".prefab";
            string dstPrefabPath = dstStr + ".prefab";

            AssetDatabase.MoveAsset(srcPrefabPath, dstPrefabPath);
        }
    }
}
```

Con eso guardado, deberías poder arrastrar y soltar el archivo `Example.test` que creamos

anteriormente en tu carpeta de Unity Assets y deberías ver el prefab correspondiente creado. Si edita `Example.test` , verá que los datos en el prefab se actualizan inmediatamente. Si arrastra el prefab en la jerarquía de la escena, verá que se actualiza así como los cambios de `Example.test` . Si mueve `Example.test` a otra carpeta, el prefab correspondiente se moverá con él. Si cambia un campo en una instancia y luego cambia el archivo `Example.test` , verá que solo se actualizan los campos que no modificó en la instancia.

Mejoras: en el ejemplo anterior, después de arrastrar `Example.test` a su carpeta de `Assets` , verá que hay un `Example.test` y un `Example.test.prefab` . Sería genial saber que funcione más como el trabajo de los importadores de modelos. `AssetBundle` solo veremos `Example.test` y es un `AssetBundle` o algo así. Si sabe cómo, por favor proporcione ese ejemplo

Lea Importadores y (Post) Procesadores en línea:
https://riptutorial.com/es/unity3d/topic/5279/importadores-y--post--procesadores

Capítulo 20: Integración de anuncios

Introducción

Este tema trata sobre la integración de servicios de publicidad de terceros, como Unity Ads o Google AdMob, en un proyecto de Unity.

Observaciones

Esto se aplica a Unity Ads .

> **Asegúrese de que el Modo de prueba para Unity Ads esté habilitado durante el desarrollo**

Usted, como desarrollador, no puede generar impresiones ni instalaciones haciendo clic en los anuncios de su propio juego. Si lo hace, viola el Acuerdo de Términos de Servicio de Unity Ads , y se le prohibirá el acceso a la red de Unity Ads por intento de fraude.

Para obtener más información, lea el acuerdo de términos de servicio de Unity Ads .

Examples

Conceptos básicos de Unity Ads en C

```
using UnityEngine;
using UnityEngine.Advertisements;

public class Example : MonoBehaviour
{
    #if !UNITY_ADS // If the Ads service is not enabled
    public string gameId; // Set this value from the inspector
    public bool enableTestMode = true; // Enable this during development
    #endif

    void InitializeAds () // Example of how to initialize the Unity Ads service
    {
        #if !UNITY_ADS // If the Ads service is not enabled
        if (Advertisement.isSupported) { // If runtime platform is supported
            Advertisement.Initialize(gameId, enableTestMode); // Initialize
        }
        #endif
    }

    void ShowAd () // Example of how to show an ad
    {
        if (Advertisement.isInitialized || Advertisement.IsReady()) { // If the ads are ready
to be shown
            Advertisement.Show(); // Show the default ad placement
        }
    }
}
```

Conceptos básicos de Unity Ads en JavaScript

```
#pragma strict
import UnityEngine.Advertisements;

#if !UNITY_ADS // If the Ads service is not enabled
public var gameId : String; // Set this value from the inspector
public var enableTestMode : boolean = true; // Enable this during development
#endif

function InitializeAds () // Example of how to initialize the Unity Ads service
{
    #if !UNITY_ADS // If the Ads service is not enabled
    if (Advertisement.isSupported) { // If runtime platform is supported
        Advertisement.Initialize(gameId, enableTestMode); // Initialize
    }
    #endif
}

function ShowAd () // Example of how to show an ad
{
    if (Advertisement.isInitialized && Advertisement.IsReady()) { // If the ads are ready to
be shown
        Advertisement.Show(); // Show the default ad placement
    }
}
```

Lea Integración de anuncios en línea: https://riptutorial.com/es/unity3d/topic/9796/integracion-de-anuncios

Capítulo 21: Mejoramiento

Observaciones

1. Si es posible, deshabilite los scripts en los objetos cuando no sean necesarios. Por ejemplo, si tiene un script en un objeto enemigo que los buscadores buscan y dispara al jugador, considere desactivar este script cuando el enemigo está demasiado lejos, por ejemplo, del jugador.

Examples

Cheques rápidos y eficientes

Evite operaciones innecesarias y llamadas a métodos siempre que pueda, especialmente en un método que se llama muchas veces por segundo, como `Update` .

Controles de distancia / rango

Use `sqrMagnitude` lugar de `magnitude` al comparar distancias. Esto evita operaciones `sqrt` innecesarias. Tenga en cuenta que al utilizar `sqrMagnitude` , el lado derecho también debe estar cuadrado.

```
if ((target.position - transform.position).sqrMagnitude < minDistance * minDistance))
```

Cheques de límites

Las intersecciones de objetos pueden verificarse crudamente verificando si sus límites de `Collider` / `Renderer` intersecan. La estructura de `Bounds` también tiene un práctico método `Intersects` que ayuda a determinar si se intersecan dos límites.

`Bounds` también nos ayudan a calcular una aproximación aproximada de la distancia *real* (de superficie a superficie) entre los objetos (consulte `Bounds.SqrDistance`).

Advertencias

La comprobación de límites funciona realmente bien para los objetos convexos, pero las comprobaciones de límites en objetos cóncavos pueden llevar a imprecisiones mucho más altas según la forma del objeto.

`Mesh.bounds` se recomienda usar `Mesh.bounds` ya que devuelve los límites del espacio local. Utilice `MeshRenderer.bounds` en `MeshRenderer.bounds` lugar.

Uso

Si tiene una operación de larga ejecución que se basa en la API de Unity no segura para subprocesos, utilice Coroutines para dividirla en varios marcos y mantener su aplicación receptiva.

Coroutines también ayuda a realizar acciones caras en cada fotograma en lugar de ejecutar esa acción en cada fotograma.

División de rutinas de larga ejecución en varios marcos

Coroutines ayuda a distribuir operaciones de larga duración en varios marcos para ayudar a mantener la velocidad de cuadros de su aplicación.

Las rutinas que pintan o generan terreno por procedimientos o generan ruido son ejemplos que pueden necesitar el tratamiento de Coroutine.

```
for (int y = 0; y < heightmap.Height; y++)
{
    for (int x = 0; x < heightmap.Width; x++)
    {
        // Generate pixel at (x, y)
        // Assign pixel at (x, y)

        // Process only 32768 pixels each frame
        if ((y * heightmap.Height + x) % 32 * 1024) == 0)
            yield return null; // Wait for next frame
    }
}
```

> El código anterior es un ejemplo fácil de entender. En el código de producción que es mejor evitar el cheque por píxel que comprueba cuándo `yield return` (tal vez hacerlo cada 2-3 filas) y comprobar la validez de calcular `for` la longitud del bucle de antemano.

Realización de acciones caras con menos frecuencia

Coroutines lo ayuda a realizar acciones costosas con menos frecuencia, por lo que no es un golpe de rendimiento tan grande como lo sería si se realizara cada fotograma.

Tomando el siguiente ejemplo directamente del Manual :

```
private void ProximityCheck()
{
    for (int i = 0; i < enemies.Length; i++)
    {
        if (Vector3.Distance(transform.position, enemies[i].transform.position) <
dangerDistance)
                return true;
    }
    return false;
}

private IEnumerator ProximityCheckCoroutine()
{
    while(true)
    {
        ProximityCheck();
        yield return new WaitForSeconds(.1f);
    }
}
```

> Las pruebas de proximidad se pueden optimizar aún más utilizando la API de CullingGroup .

Errores comunes

Un error común que cometen los desarrolladores es acceder a los resultados o efectos secundarios de las coroutinas *fuera de* ellas. Coroutines devuelve el control a la persona que llama tan pronto como se encuentra una declaración de `yield return` y el resultado o efecto secundario puede que aún no se haya realizado. Para evitar problemas en los que *tenga* que usar el resultado / efecto secundario fuera de la rutina, marque esta respuesta .

Instrumentos de cuerda

Se podría argumentar que hay un mayor número de recursos en la Unidad que en la humilde cadena, pero es uno de los aspectos más fáciles de solucionar desde el principio.

Las operaciones de cuerdas construyen basura

La mayoría de las operaciones de cadena generan pequeñas cantidades de basura, pero si esas operaciones se llaman varias veces en el transcurso de una sola actualización, se acumula. Con el tiempo, activará la recolección automática de basura, lo que puede resultar en un pico de CPU visible.

Caché tus operaciones de cadena

Considere el siguiente ejemplo.

```
string[] StringKeys = new string[] {
    "Key0",
    "Key1",
    "Key2"
};

void Update()
{
    for (var i = 0; i < 3; i++)
    {
        // Cached, no garbage generated
        Debug.Log(StringKeys[i]);
    }

    for (var i = 0; i < 3; i++)
    {
        // Not cached, garbage every cycle
        Debug.Log("Key" + i);
    }

    // The most memory-efficient way is to not create a cache at all and use literals or
constants.
    // However, it is not necessarily the most readable or beautiful way.
    Debug.Log("Key0");
    Debug.Log("Key1");
    Debug.Log("Key2");
}
```

Puede parecer tonto y redundante, pero si está trabajando con Shaders, puede encontrarse en situaciones como estas. Cachear las llaves marcará la diferencia.

Tenga en cuenta que las cadenas *literales* y las *constantes* no generan basura, ya que se inyectan estáticamente en el espacio de la pila del programa. Si está *generando* cadenas en tiempo de ejecución y está *garantizado* que generará las **mismas** cadenas cada vez que el ejemplo anterior, el almacenamiento en caché definitivamente ayudará.

Para otros casos donde la cadena generada no es la misma cada vez, no hay otra alternativa para generar esas cadenas. Como tal, el pico de memoria con cadenas que se generan manualmente cada vez es generalmente despreciable, a menos que se generen decenas de miles de cadenas a la vez.

La mayoría de las operaciones de cadena son mensajes de depuración

Hacer operaciones de cadena para mensajes de depuración, es decir. `Debug.Log("Object Name: " + obj.name)` está bien y no se puede evitar durante el desarrollo. Sin embargo, es importante asegurarse de que los mensajes de depuración irrelevantes no terminen en el producto publicado.

Una forma es usar el atributo condicional en sus llamadas de depuración. Esto no solo elimina las llamadas de método, sino también todas las operaciones de cadena que entran en él.

```
using UnityEngine;
using System.Collections;

public class ConditionalDebugExample: MonoBehaviour
{
    IEnumerator Start()
    {
        while(true)
        {
            // This message will pop up in Editor but not in builds
            Log("Elapsed: " + Time.timeSinceLevelLoad);
            yield return new WaitForSeconds(1f);
        }
    }

    [System.Diagnostics.Conditional("UNITY_EDITOR")]
    void Log(string Message)
    {
        Debug.Log(Message);
    }
}
```

Este es un ejemplo simplificado. Es posible que desee invertir algo de tiempo en diseñar una rutina de registro más completa.

Comparación de cuerdas

Esta es una optimización menor, pero vale la pena una mención. Comparar cadenas es un poco más complicado de lo que uno podría pensar. El sistema intentará tener en cuenta las diferencias culturales por defecto. Puede optar por utilizar una comparación binaria simple, que funciona más rápido.

```
// Faster string comparison
if (strA.Equals(strB, System.StringComparison.Ordinal)) {...}
// Compared to
if (strA == strB) {...}

// Less overhead
if (!string.IsNullOrEmpty(strA)) {...}
// Compared to
if (strA == "") {...}

// Faster lookups
Dictionary<string, int> myDic = new Dictionary<string, int>(System.StringComparer.Ordinal);
// Compared to
Dictionary<string, int> myDictionary = new Dictionary<string, int>();
```

Referencias de caché

Cache las referencias para evitar las llamadas costosas, especialmente en la función de actualización. Esto se puede hacer almacenando estas referencias en caché al inicio, si están disponibles o cuando estén disponibles, y comprobando que no haya null / bool flat para evitar volver a obtener la referencia.

Ejemplos:

Referencias de componentes de caché

cambio

```
void Update()
{
    var renderer = GetComponent<Renderer>();
    renderer.material.SetColor("_Color", Color.green);
}
```

a

```
private Renderer myRenderer;
void Start()
{
    myRenderer = GetComponent<Renderer>();
}

void Update()
{
    myRenderer.material.SetColor("_Color", Color.green);
}
```

Referencias de objetos de caché

cambio

```
void Update()
{
    var enemy = GameObject.Find("enemy");
    enemy.transform.LookAt(new Vector3(0,0,0));
}
```

a

```
private Transform enemy;

void Start()
{
    this.enemy = GameObject.Find("enemy").transform;
}

void Update()
{
    enemy.LookAt(new Vector3(0, 0, 0));
}
```

Además, caché llamadas costosas como llamadas a Mathf cuando sea posible.

Evitar los métodos de llamada utilizando cadenas

Evite llamar a métodos utilizando cadenas que puedan aceptar métodos. Este enfoque hará uso

de la reflexión que puede ralentizar el juego, especialmente cuando se utiliza en la función de actualización.

Ejemplos:

```
    //Avoid StartCoroutine with method name
    this.StartCoroutine("SampleCoroutine");

    //Instead use the method directly
    this.StartCoroutine(this.SampleCoroutine());

    //Avoid send message
    var enemy = GameObject.Find("enemy");
    enemy.SendMessage("Die");

    //Instead make direct call
    var enemy = GameObject.Find("enemy") as Enemy;
    enemy.Die();
```

Evita los métodos de unidad vacía.

Evita los métodos de unidad vacía. Además de ser un mal estilo de programación, hay una pequeña sobrecarga involucrada en las secuencias de comandos de tiempo de ejecución. En muchos casos, esto puede acumularse y afectar el rendimiento.

```
void Update
{
}

void FixedUpdate
{
}
```

Lea Mejoramiento en línea: https://riptutorial.com/es/unity3d/topic/3433/mejoramiento

Capítulo 22: Patrones de diseño

Examples

Modelo de diseño del controlador de vista (MVC)

El controlador de vista de modelo es un patrón de diseño muy común que ha existido durante bastante tiempo. Este patrón se enfoca en reducir el código de *espagueti* al separar las clases en partes funcionales. Recientemente he estado experimentando con este patrón de diseño en Unity y me gustaría presentar un ejemplo básico.

Un diseño de MVC consta de tres partes principales: Modelo, Vista y Controlador.

Modelo: el modelo es una clase que representa la porción de datos de su objeto. Esto podría ser un jugador, un inventario o un nivel entero. Si está programado correctamente, debería poder tomar este script y usarlo fuera de Unity.

Tenga en cuenta algunas cosas sobre el modelo:

- No debe heredar de Monobehaviour.
- No debe contener código específico de Unity para la portabilidad.
- Ya que estamos evitando las llamadas a la API de Unity, esto puede dificultar cosas como los convertidores implícitos en la clase Modelo (se requieren soluciones)

Jugador.cs

```
using System;

public class Player
{
    public delegate void PositionEvent(Vector3 position);
    public event PositionEvent OnPositionChanged;

    public Vector3 position
    {
        get
        {
            return _position;
        }
        set
        {
            if (_position != value) {
                _position = value;
                if (OnPositionChanged != null) {
                    OnPositionChanged(value);
                }
            }
        }
    }
    private Vector3 _position;
}
```

Vector3.cs

Una clase personalizada de Vector3 para usar con nuestro modelo de datos.

```
using System;

public class Vector3
{
    public float x;
    public float y;
    public float z;

    public Vector3(float x, float y, float z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

Vista: la vista es una clase que representa la parte de visualización vinculada al modelo. Esta es una clase apropiada para derivar de Monobehaviour. Este debe contener código que interactúe directamente con las API específicas de Unity, como `OnCollisinEnter` , `Start` , `Update` , etc.

- Típicamente hereda de Monobehaviour
- Contiene código específico de Unity

PlayerView.cs

```
using UnityEngine;

public class PlayerView : Monobehaviour
{
    public void SetPosition(Vector3 position)
    {
        transform.position = position;
    }
}
```

Controlador: el controlador es una clase que une el modelo y la vista. Los controladores mantienen sincronizados tanto el modelo como la vista, así como la interacción de la unidad. El controlador puede escuchar eventos de cualquiera de los socios y actualizarse en consecuencia.

- Enlaza tanto el modelo como la vista mediante el estado de sincronización
- Puede impulsar la interacción entre socios
- Los controladores pueden o no ser portátiles (es posible que tenga que usar el código de Unity aquí)
- Si decide no hacer que su controlador sea portátil, considere convertirlo en un Monobehaviour para ayudar con la inspección del editor

PlayerController.cs

```
using System;

public class PlayerController
{
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public PlayerController(Player model, PlayerView view)
    {
        this.model = model;
        this.view = view;

        this.model.OnPositionChanged += OnPositionChanged;
    }

    private void OnPositionChanged(Vector3 position)
    {
        // Sync
        Vector3 pos = this.model.position;

        // Unity call required here! (we lost portability)
        this.view.SetPosition(new UnityEngine.Vector3(pos.x, pos.y, pos.z));
    }

    // Calling this will fire the OnPositionChanged event
    private void SetPosition(Vector3 position)
    {
        this.model.position = position;
    }
}
```

Uso final

Ahora que tenemos todas las piezas principales, podemos crear una fábrica que generará las tres partes.

PlayerFactory.cs

```
using System;

public class PlayerFactory
{
    public PlayerController controller { get; private set; }
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public void Load()
    {
        // Put the Player prefab inside the 'Resources' folder
        // Make sure it has the 'PlayerView' Component attached
        GameObject prefab = Resources.Load<GameObject>("Player");
        GameObject instance = GameObject.Instantiate<GameObject>(prefab);
        this.model = new Player();
        this.view = instance.GetComponent<PlayerView>();
        this.controller = new PlayerController(model, view);
    }
}
```

Y finalmente podemos llamar a la fábrica de un gerente ...

Manager.cs

```
using UnityEngine;

public class Manager : Monobehaviour
{
    [ContextMenu("Load Player")]
    private void LoadPlayer()
    {
        new PlayerFactory().Load();
    }
}
```

Adjunte la secuencia de comandos de Manager a un GameObject vacío en la escena, haga clic con el botón derecho en el componente y seleccione "Cargar jugador".

Para una lógica más compleja, puede introducir la herencia con clases base abstractas e interfaces para una arquitectura mejorada.

Lea Patrones de diseño en línea: https://riptutorial.com/es/unity3d/topic/10842/patrones-de-diseno

Capítulo 23: Plataformas moviles

Sintaxis

- public static int Input.touchCount
- Toque público estático Input.GetTouch (índice int)

Examples

Detección de toque

Para detectar un toque en Unity es bastante simple, solo tenemos que usar `Input.GetTouch()` y pasarle un índice.

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        if (Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Began)
        {
            //Do Stuff
        }
    }
}
```

o

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        for(int i = 0; i < Input.touchCount; i++)
        {
            if (Input.GetTouch(i).phase == TouchPhase.Began)
            {
                //Do Stuff
            }
        }
    }
}
```

Estos ejemplos dan el toque del último cuadro de juego.

TouchPhase

Dentro de la enumeración de TouchPhase hay 5 tipos diferentes de TouchPhase

- Comenzó - un dedo tocó la pantalla
- Movido - un dedo movido en la pantalla
- Estacionario: un dedo está en la pantalla pero no se mueve
- Terminado - un dedo fue levantado de la pantalla
- Cancelado: el sistema canceló el seguimiento para el toque

Por ejemplo, para mover el objeto al que se adjunta esta secuencia de comandos a través de la pantalla en función del toque.

```
public class TouchMoveExample : MonoBehaviour
{
    public float speed = 0.1f;

    void Update () {
        if(Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Moved)
        {
            Vector2 touchDeltaPosition = Input.GetTouch(0).deltaPosition;
            transform.Translate(-touchDeltaPosition.x * speed, -touchDeltaPosition.y * speed,
0);
        }
    }
}
```

Lea Plataformas moviles en línea: https://riptutorial.com/es/unity3d/topic/6285/plataformas-moviles

Capítulo 24: Plugins de Android 101 - Una introducción

Introducción

Este tema es la primera parte de una serie sobre cómo crear complementos de Android para Unity. Comience aquí si tiene poca o ninguna experiencia en la creación de complementos y / o el sistema operativo Android.

Observaciones

A través de esta serie, utilizo extensivamente enlaces externos que te invito a leer. Mientras que las versiones parafraseadas del contenido relevante se incluirán aquí, puede haber ocasiones en que la lectura adicional ayude.

Comenzando con los complementos de Android

Actualmente, Unity proporciona dos formas de llamar al código nativo de Android.

1. Escriba el código nativo de Android en Java y llame a estas funciones de Java usando C #
2. Escriba el código C # para llamar directamente a las funciones que forman parte del sistema operativo Android

Para interactuar con el código nativo, Unity proporciona algunas clases y funciones.

- AndroidJavaObject : esta es la clase base que Unity proporciona para interactuar con el código nativo. Casi cualquier objeto devuelto desde el código nativo se puede almacenar como y AndroidJavaObject
- AndroidJavaClass - Herencias de AndroidJavaObject. Esto se utiliza para hacer referencia a las clases en su código nativo
- Obtenga / establezca los valores de una instancia de un objeto nativo y las versiones estáticas GetStatic / SetStatic
- Call / CallStatic para llamar a funciones nativas no estáticas y estáticas

---

Esquema para crear un plugin y terminología.

1. Escribir código Java nativo en Android Studio
2. Exportar el código en un archivo JAR / AAR (pasos aquí para archivos JAR y archivos AAR)
3. Copie el archivo JAR / AAR en su proyecto de Unity en **Assets / Plugins / Android**
4. Escriba el código en Unity (C # siempre ha sido el camino a seguir aquí) para llamar a las

funciones en el complemento

Tenga en cuenta que los primeros tres pasos se aplican SOLAMENTE si desea tener un complemento nativo.

De aquí en adelante, me referiré al archivo JAR / AAR como el **complemento nativo** , y al script C # como el **contenedor C #**

Elegir entre los métodos de creación de plugins.

Inmediatamente es obvio que la primera forma de crear complementos es larga, por lo que elegir su ruta parece ser discutible. Sin embargo, el método 1 es la ÚNICA forma de llamar a un código personalizado. Entonces, ¿cómo se elige?

En pocas palabras, tiene su plugin

1. Involucrar código personalizado - Elija el método 1
2. ¿Solo invocar funciones nativas de Android? - Elija el método 2

NO intente "mezclar" (es decir, una parte del complemento que utiliza el método 1 y la otra que utiliza el método 2) ¡los dos métodos! Si bien es completamente posible, a menudo es poco práctico y doloroso de manejar.

Examples

UnityAndroidPlugin.cs

Cree un nuevo script de C # en Unity y reemplace su contenido con lo siguiente

```
using UnityEngine;
using System.Collections;

public static class UnityAndroidPlugin {

}
```

UnityAndroidNative.java

Cree una nueva clase de Java en Android Studio y reemplace su contenido con lo siguiente

```
package com.axs.unityandroidplugin;
import android.util.Log;
import android.widget.Toast;
import android.app.ActivityManager;
import android.content.Context;
```

```
public class UnityAndroidNative {


}
```

UnityAndroidPluginGUI.cs

Cree un nuevo script de C # en Unity y pegue estos contenidos

```
using UnityEngine;
using System.Collections;

public class UnityAndroidPluginGUI : MonoBehaviour {

    void OnGUI () {

    }

}
```

Lea Plugins de Android 101 - Una introducción en línea:
https://riptutorial.com/es/unity3d/topic/10032/plugins-de-android-101---una-introduccion

Capítulo 25: Prefabricados

Sintaxis

- Public static Object PrefabUtility.InstantiatePrefab (Object target);
- Objeto estático público AssetDatabase.LoadAssetAtPath (string assetPath, Tipo de tipo);
- Objeto estático público Object.Instantiate (Objeto original);
- Recursos de objetos estáticos públicos. Carga (ruta de cadena);

Examples

Introducción

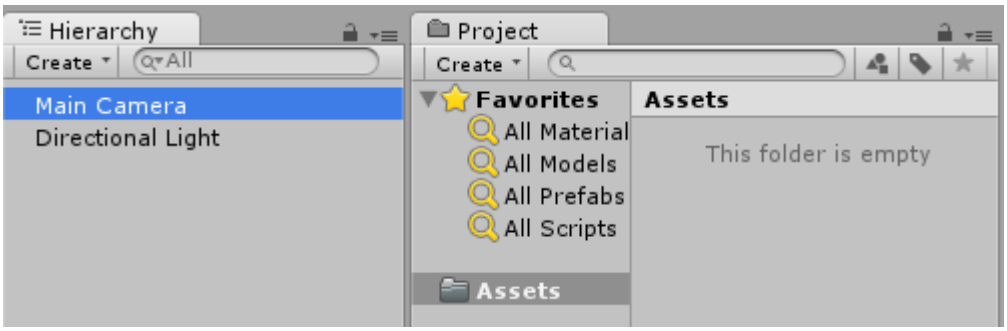
Las casas prefabricadas son un tipo de activo que permite el almacenamiento de un GameObject completo con sus componentes, propiedades, componentes adjuntos y valores de propiedad serializados. Hay muchos escenarios donde esto es útil, incluyendo:

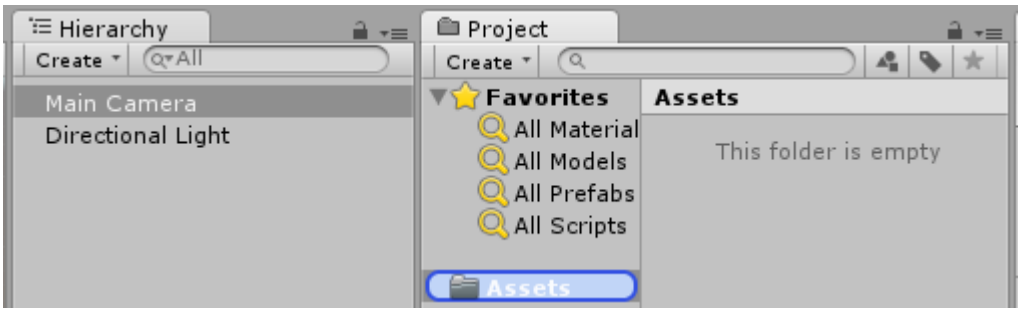
- Duplicar objetos en una escena.
- Compartiendo un objeto común a través de múltiples escenas
- Poder modificar una prefabulación una vez y hacer que los cambios se apliquen a múltiples objetos / escenas
- Crear objetos duplicados con modificaciones menores, mientras que los elementos comunes se pueden editar desde una base prefabricada
- Instalar GameObjects en tiempo de ejecución

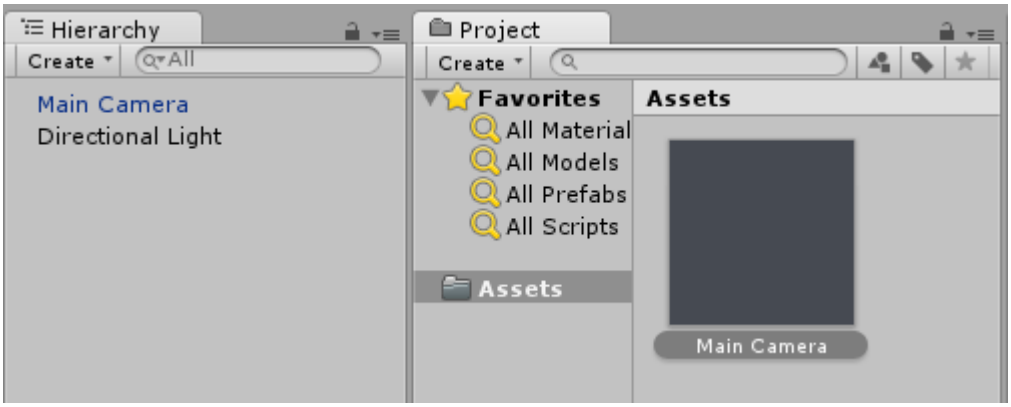
Hay una regla de oro de las clases en la Unidad que dice "todo debería ser Prefabs". Si bien esto probablemente sea exagerado, fomenta la reutilización del código y la creación de GameObjects de una manera reutilizable, que es a la vez eficiente en memoria y buen diseño.

Creación de prefabs

Para crear un prefab, arrastre un objeto del juego desde la jerarquía de la escena a la carpeta o subcarpeta de **Activos** :





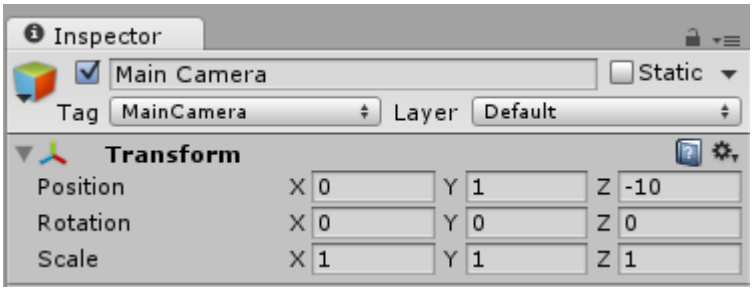


El nombre del objeto del juego se vuelve azul, lo que indica que está **conectado a una prefab** . Ahora este objeto es una **instancia prefabricada** , al igual que una instancia de objeto de una clase.

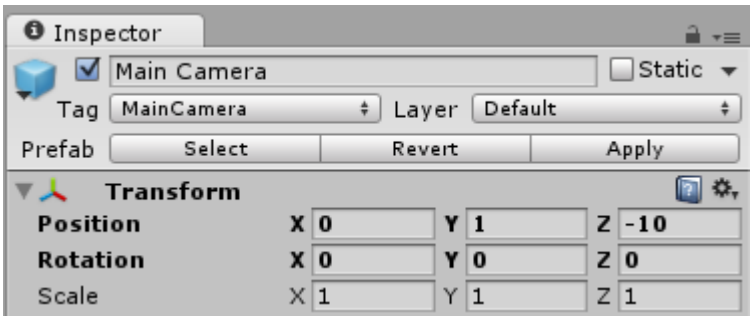
Un prefab puede ser eliminado después de la instanciación. En ese caso, el nombre del objeto del juego previamente conectado se vuelve rojo:

Inspector prefabricado

Si selecciona una prefab en la vista de jerarquía, notará que su inspector se diferencia ligeramente de un objeto de juego normal:



vs



Las propiedades en negrita significan que sus valores difieren de los valores prefabricados. Puede cambiar cualquier propiedad de un prefab instanciado sin afectar los valores prefabricados originales. Cuando se cambia un valor en una instancia prefabricada, se pone en negrita y cualquier cambio posterior del mismo valor en la prefabricación no se reflejará en la instancia modificada.

Puede revertir a los valores prefabricados originales haciendo clic en el botón **Revertir** , que también reflejará los cambios de valor en la instancia. Además, para revertir un valor individual, puede hacer clic con el botón derecho y presionar **Revertir valor en Prefab** . Para revertir un componente, haga clic derecho y presione **Revertir a Prefab** .

Al hacer clic en el botón **Aplicar** se sobrescriben los valores de propiedad prefabricados con los valores de propiedad del objeto del juego actual. No hay un botón "Deshacer" o un cuadro de diálogo de confirmación, así que maneja este botón con cuidado.

Los botones de selección resaltan el prefab conectado en la estructura de carpetas del proyecto.

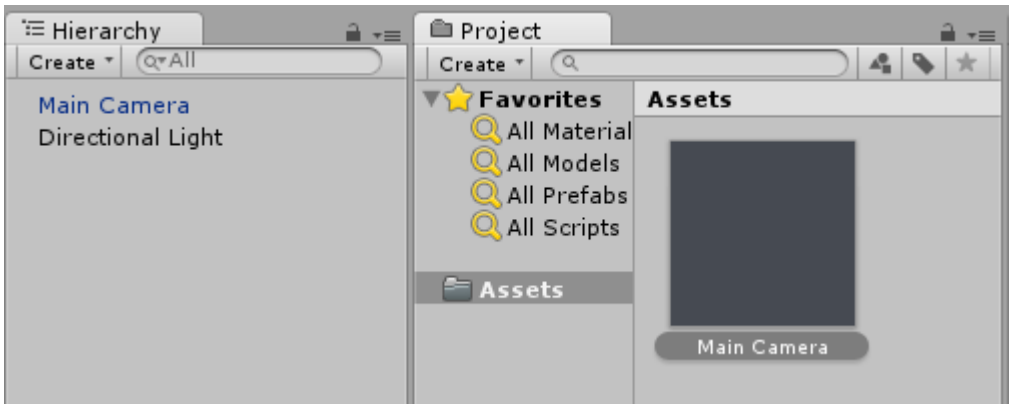
Creación de prefabs

Hay dos formas de crear una instancia de prefabs: durante el **tiempo de diseño** o en **tiempo de ejecución** .

Tiempo de diseño de instanciación

La creación de instancias prefabricadas en el momento del diseño es útil para colocar visualmente múltiples instancias del mismo objeto (por ejemplo, *colocar árboles al diseñar un nivel de su juego*).

- Para crear una instancia visual de un prefab, arrástrelo desde la vista del proyecto a la jerarquía de escenas.



- Si está escribiendo una extensión de editor , también puede crear una instancia de un `PrefabUtility.InstantiatePrefab()` método de `PrefabUtility.InstantiatePrefab()` :

```
GameObject gameObject =
(GameObject)PrefabUtility.InstantiatePrefab(AssetDatabase.LoadAssetAtPath("Assets/MainCamera.pref
typeof(GameObject)));
```

Instanciación en tiempo de ejecución

La creación de instancias prefabricadas en tiempo de ejecución es útil para crear instancias de un objeto de acuerdo con cierta lógica (por ejemplo, *generar un enemigo cada 5 segundos*).

Para crear una instancia de una prefab, necesita una referencia al objeto prefabricado. Esto se puede hacer teniendo un campo de `public GameObject` en su script `MonoBehaviour` (y configurando su valor utilizando el inspector en el Editor de Unity):

```
public class SomeScript : MonoBehaviour {
   public GameObject prefab;
}
```

O colocando el prefab en la carpeta de Recursos y utilizando `Resources.Load` . `Resources.Load` :

```
GameObject prefab = Resources.Load("Assets/Resources/MainCamera");
```

Una vez que tenga una referencia al objeto prefabricado, puede crear una instancia utilizando la función de `Instantiate` en cualquier parte de su código (por ejemplo, *dentro de un bucle para crear múltiples objetos*):

```
GameObject gameObject = Instantiate<GameObject>(prefab, new Vector3(0,0,0),
Quaternion.identity);
```

Nota: El término *prefabricado* no existe en el tiempo de ejecución.

Prefabricados anidados

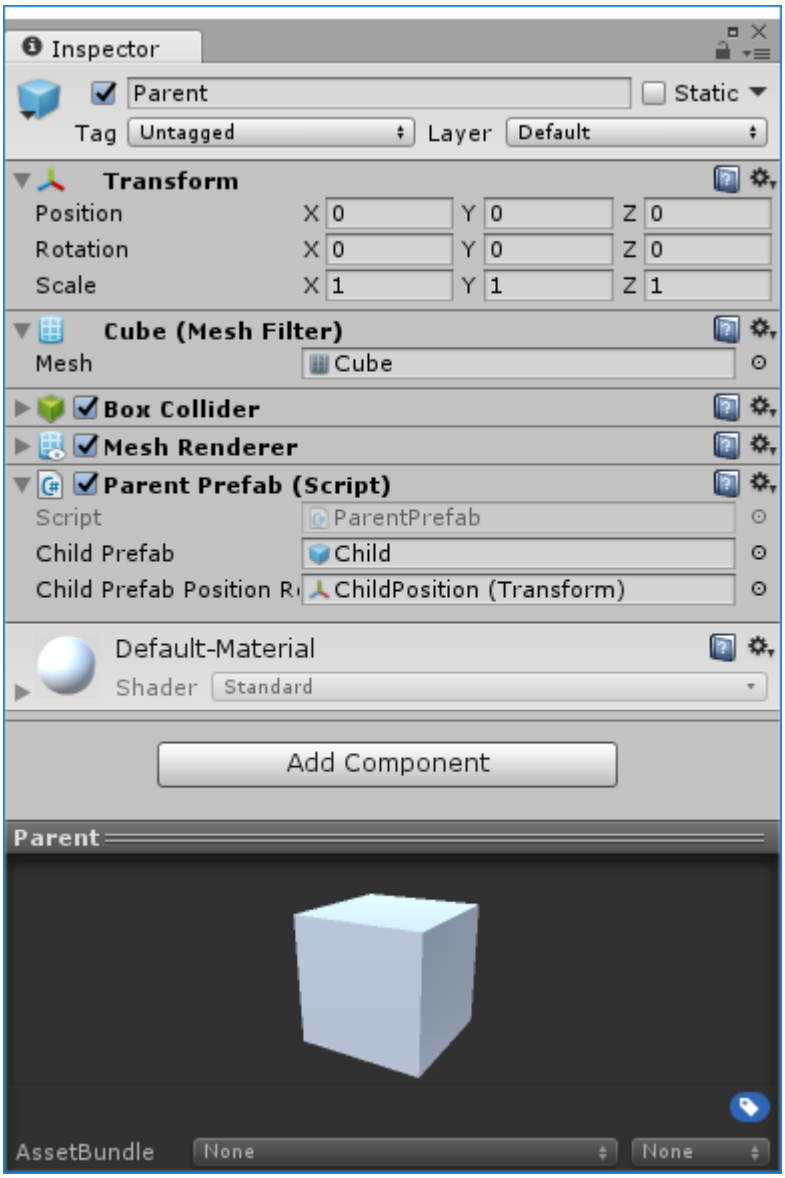
Los prefabs anidados no están disponibles en Unity en este momento. Puede arrastrar una

prefabricada a otra, y aplicar eso, pero cualquier cambio en la prefabricación secundaria no se aplicará a una anidada.

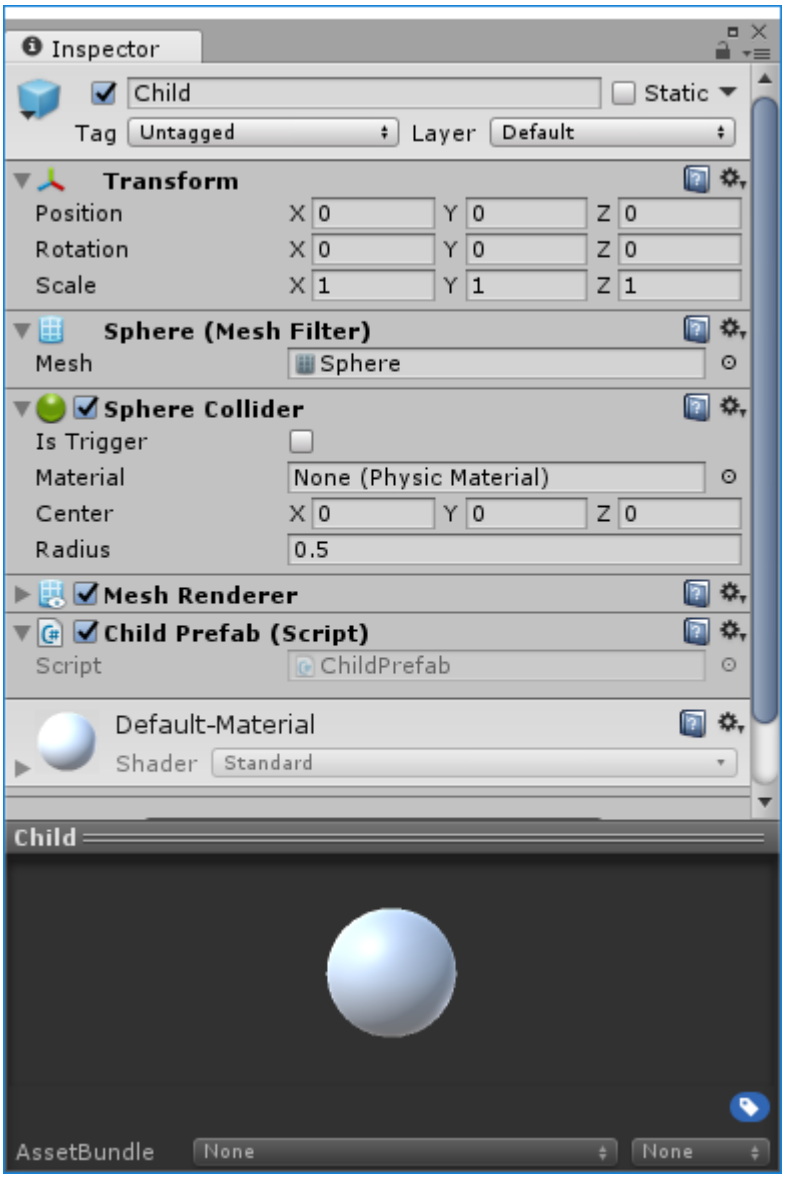
Pero hay una solución simple: **tiene que agregar a la prefab para padres una secuencia de comandos simple, que ejemplificará una para niños.**

```
using UnityEngine;

public class ParentPrefab : MonoBehaviour {

    [SerializeField] GameObject childPrefab;
    [SerializeField] Transform childPrefabPositionReference;

    // Use this for initialization
    void Start () {
        print("Hello, I'm a parent prefab!");
        Instantiate(
            childPrefab,
            childPrefabPositionReference.position,
            childPrefabPositionReference.rotation,
            gameObject.transform
        );
    }
}
```

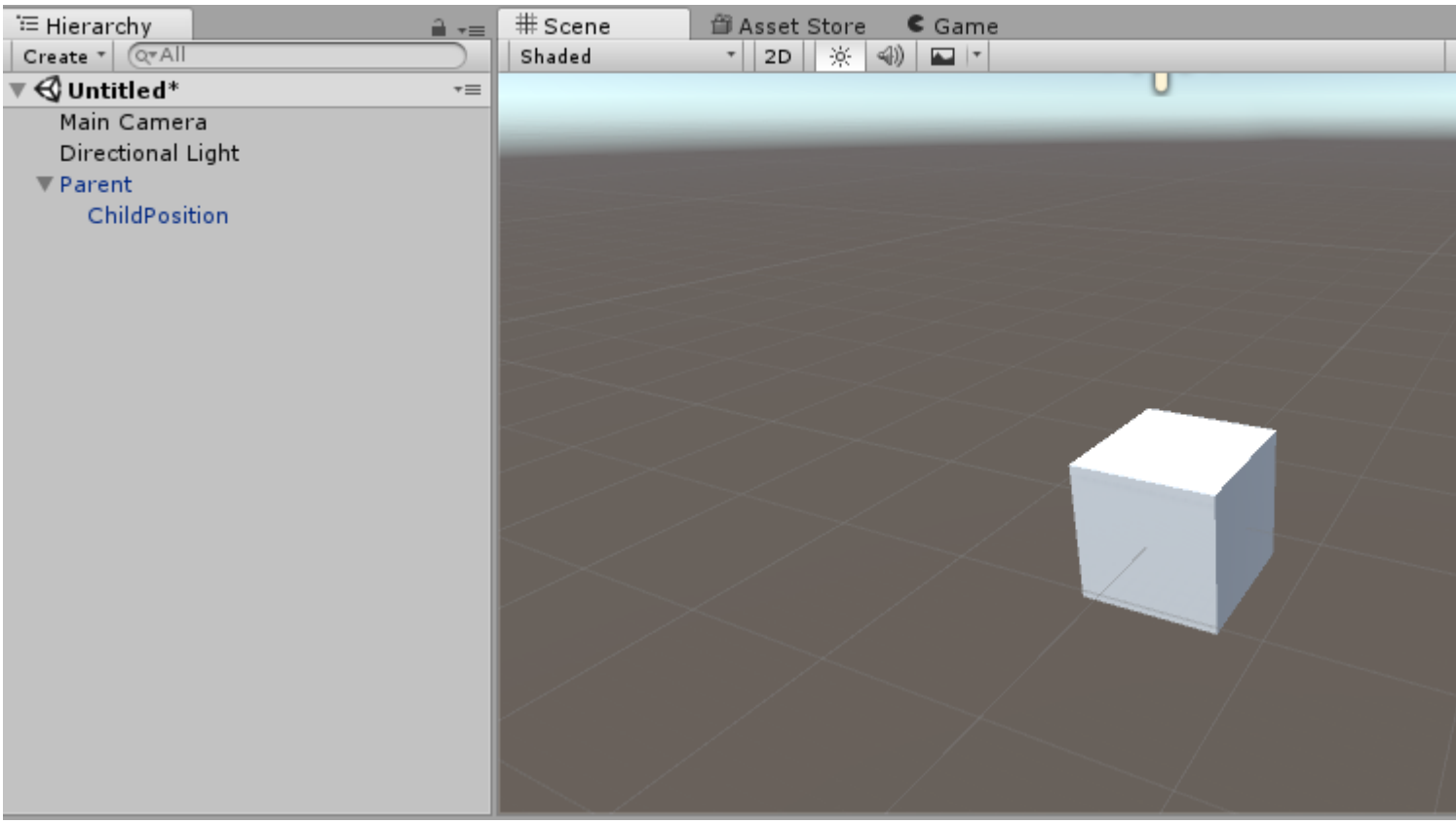
Padre prefabricado:

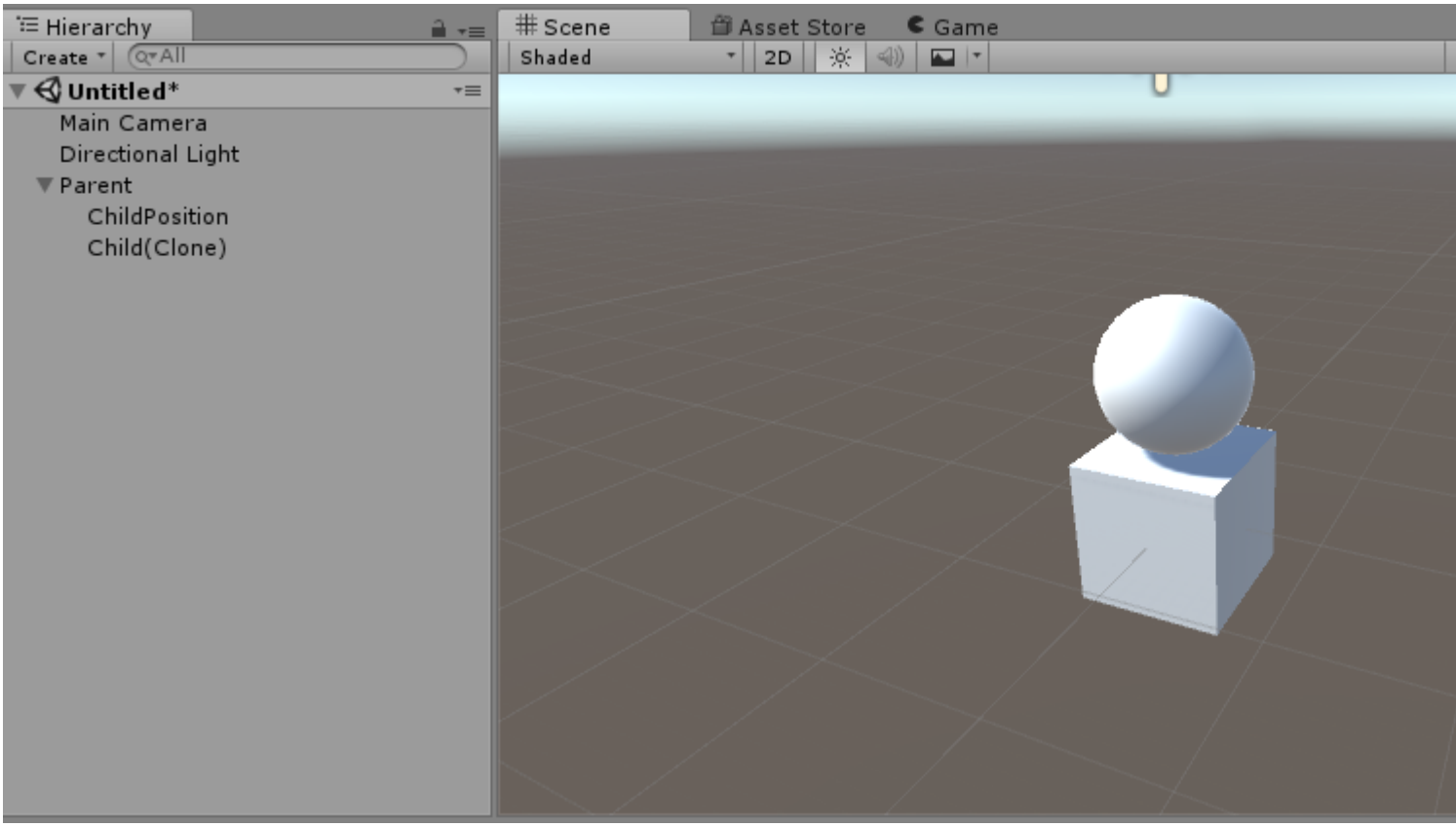


Prefabricado infantil:



Escena antes y después del inicio:





Lea Prefabricados en línea: https://riptutorial.com/es/unity3d/topic/2133/prefabricados

Capítulo 26: Raycast

Parámetros

Parámetro	Detalles
origen	El punto de partida del rayo en coordenadas mundiales.
dirección	La dirección del rayo.
distancia maxima	La distancia máxima que el rayo debe verificar para detectar colisiones.
máscara de capa	Una máscara de capa que se usa para ignorar selectivamente los colisionadores al lanzar un rayo.
queryTriggerInteraction	Especifica donde esta consulta debe golpear Triggers.

Examples

Física Raycast

Esta función proyecta un rayo desde el `origin` del punto en la `direction` de la longitud `maxDistance` contra todos los colisionadores en la escena.

La función toma la `direction` `origin` `maxDistance` y calcula si hay un colisionador frente al GameObject.

```
Physics.Raycast(origin, direction, maxDistance);
```

Por ejemplo, esta función imprimirá `Hello World` en la consola si hay algo dentro de las 10 unidades del `GameObject` adjunto:

```
using UnityEngine;

public class TestPhysicsRaycast: MonoBehaviour
{
    void FixedUpdate()
    {
        Vector3 fwd = transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast(transform.position, fwd, 10))
            print("Hello World");
    }
}
```

Physics2D Raycast2D

Puedes usar raycasts para verificar si un ai puede caminar sin caerse del borde de un nivel.

```
using UnityEngine;

public class Physics2dRaycast: MonoBehaviour
    {
        public LayerMask LineOfSightMask;
        void FixedUpdate()
        {
            RaycastHit2D hit = Physics2D.Raycast(raycastRightPart, Vector2.down, 0.6f *
heightCharacter, LineOfSightMask);
            if(hit.collider != null)
            {
                //code when the ai can walk
            }
            else
            {
                //code when the ai cannot walk
            }
    }
}
```

En este ejemplo, la dirección es correcta. La variable raycastRightPart es la parte correcta del personaje, por lo que el raycast ocurrirá en la parte derecha del personaje. La distancia es de 0,6 veces la altura del personaje, por lo que el raycast no dará un impacto cuando toque el suelo que está muy por debajo del suelo en el que se encuentra en este momento. Asegúrese de que Layermask esté configurado solo en el suelo, de lo contrario también detectará otros tipos de objetos.

> RaycastHit2D en sí mismo es una estructura y no una clase, por lo que el hit no puede ser nulo; esto significa que debe verificar el colisionador de una variable RaycastHit2D.

Encapsulando llamadas Raycast

Hacer que sus scripts llamen a `Raycast` directamente puede ocasionar problemas si necesita cambiar las matrices de colisión en el futuro, ya que tendrá que rastrear cada campo de `LayerMask` para adaptarse a los cambios. Dependiendo del tamaño de su proyecto, esto puede convertirse en una gran empresa.

Encapsular `Raycast` llamadas de `Raycast` puede hacer que tu vida sea más fácil en el futuro.

Mirándolo desde un principio de SoC , un objeto de juego realmente no debería saber o preocuparse por LayerMasks. Solo necesita un método para escanear sus alrededores. Si el resultado de raycast devuelve esto o eso no debería importar al objeto de juego. Solo debe actuar sobre la información que recibe y no hacer suposiciones sobre el entorno en el que existe.

Una forma de abordar esto es mover el valor de LayerMask a las instancias de ScriptableObject y usarlas como una forma de servicios de raycast que inyecta en sus scripts.

```
// RaycastService.cs
using UnityEngine;
```

```
[CreateAssetMenu(menuName = "StackOverflow")]
public class RaycastService : ScriptableObject
{
    [SerializeField]
    LayerMask layerMask;

    public RaycastHit2D Raycast2D(Vector2 origin, Vector2 direction, float distance)
    {
        return Physics2D.Raycast(origin, direction, distance, layerMask.value);
    }

    // Add more methods as needed

}
```

```
// MyScript.cs
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [SerializeField]
    RaycastService raycastService;

    void FixedUpdate()
    {
        RaycastHit2D hit = raycastService.Raycast2D(Vector2.zero, Vector2.down, 1f);
    }
}
```

Esto le permite realizar una serie de servicios de emisión de rayos, todos con diferentes combinaciones de LayerMask para diferentes situaciones. Podría tener uno que golpee solo colisionadores de tierra, y otro que golpee colisionadores de tierra y plataformas unidireccionales.

Si alguna vez necesita realizar cambios drásticos en sus configuraciones de LayerMask, solo necesita actualizar estos activos RaycastService.

Otras lecturas

- Inversión de control
- Inyección de dependencia

Lea Raycast en línea: https://riptutorial.com/es/unity3d/topic/2826/raycast

Capítulo 27: Realidad Virtual (VR)

Examples

Plataformas VR

Hay dos plataformas principales en la realidad virtual, una es la plataforma móvil, como **Google Cardboard** , **Samsung GearVR** , la otra es la plataforma para PC, como **HTC Vive, Oculus, PS VR** ...

Unity es oficialmente compatible con **Oculus Rift** , **Google Carboard** , **Steam VR** , **Playstation VR** , **Gear VR** y **Microsoft Hololens** .

La mayoría de las plataformas tienen su propio soporte y SDK. Por lo general, es necesario descargar el SDK como una extensión en primer lugar para la unidad.

SDKs:

- Google Cartón
- Plataforma de ensueño
- Samsung GearVR (integrado desde Unity 5.3)
- Oculus Rift
- HTC Vive / Open VR
- Microsoft Hololens

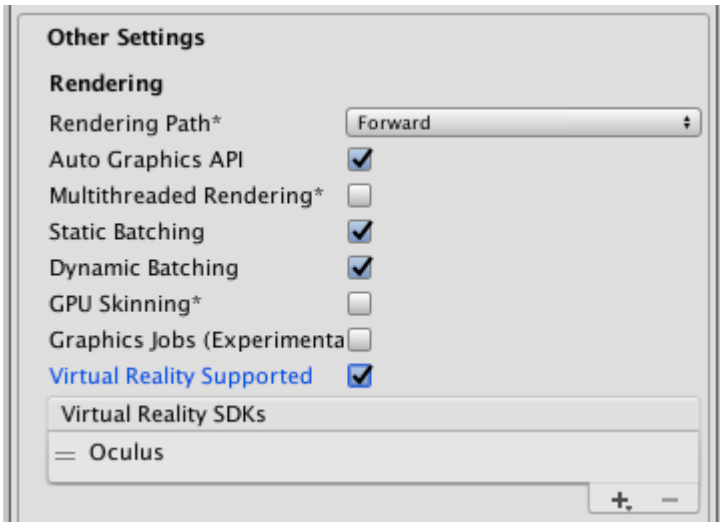
Documentación:

- Google Cardboard / Daydream
- Samsung GearVR
- Oculus Rift
- HTC Vive
- Microsoft Hololens

Habilitar el soporte de VR

En Unity Editor, abra **Configuración del reproductor** (Editar> Configuración del proyecto> Jugador).

En **Otras configuraciones** , marque *Virtual Reality Supported* .



Agregue o elimine dispositivos VR para cada destino de compilación en la lista de *Realkity SDKs* debajo de la casilla de verificación.

Hardware

Existe una dependencia de hardware necesaria para una aplicación de VR, que generalmente depende de la plataforma para la que está construyendo. Existen 2 categorías amplias para dispositivos de hardware basadas en sus capacidades de movimiento:

1. 3 DOF (Grados de Libertad)
2. 6 DOF (Grados de Libertad)

3 DOF significa que el movimiento de la pantalla montada en la cabeza (HMD) está restringido para operar en 3 dimensiones que gira alrededor de los tres ejes ortogonales centrados en el centro de gravedad HMD: los ejes longitudinal, vertical y horizontal. El movimiento en torno al eje longitudinal se denomina balanceo, el movimiento en torno al eje lateral se denomina inclinación y el movimiento en torno al eje perpendicular se denomina guiñada, principios similares que gobiernan el movimiento de cualquier objeto en movimiento como un avión o un automóvil, lo que significa que aunque sea capaz de ver en todas las direcciones X, Y, Z mediante el movimiento de su HMD en el entorno virtual, pero no podría mover ni tocar nada (el movimiento de un controlador Bluetooth adicional no es el mismo).

Sin embargo, 6 DOF permite una experiencia a escala de habitación en la que también puede moverse alrededor de los ejes X, Y y Z, aparte de los movimientos de balanceo, inclinación y giro alrededor de su centro de gravedad, por lo tanto, el grado de libertad de 6 grados.

Actualmente, un VR a escala de sala para 6 DOF requiere un alto rendimiento de cómputo con una tarjeta gráfica de gama alta y RAM que probablemente no obtendrá de sus computadoras portátiles estándar y requerirá una computadora de escritorio con un rendimiento óptimo y al menos 6 pies x 6 pies espacio libre, mientras que una experiencia de 3 DOF se puede lograr con solo un teléfono inteligente estándar con un giro incorporado (incorporado en la mayoría de los teléfonos inteligentes modernos que cuestan alrededor de $ 200 o más).

Algunos dispositivos comunes disponibles en el mercado hoy en día son:

- Oculus Rift (6 DOF)
- HTC Vive (6 DOF)
- Ensueño (3 dof)
- Gear VR accionado por Oculus (3 DOF)
- Google Cartón (3 DOF)

Lea Realidad Virtual (VR) en línea: https://riptutorial.com/es/unity3d/topic/5787/realidad-virtual--vr-

Capítulo 28: Recursos

Examples

Introducción

Con la clase de Recursos es posible cargar dinámicamente los activos que no forman parte de la escena. Es muy útil cuando tiene que usar activos a pedido, por ejemplo, localizar audios en varios idiomas, textos, etc.

Los activos se deben colocar en una carpeta llamada **Recursos** . Es posible tener varias carpetas de Recursos distribuidas en la jerarquía del proyecto. `Resources` clase de `Resources` inspeccionará todas las carpetas de recursos que pueda tener.

Cada activo colocado en Recursos se incluirá en la compilación, incluso si no se hace referencia en su código. Por lo tanto, no inserte activos en Recursos indiscriminadamente.

```
//Example of how to load language specific audio from Resources

[RequireComponent(typeof(AudioSource))]
public class loadIntroAudio : MonoBehaviour {
    void Start () {
        string language = Application.systemLanguage.ToString();
        AudioClip ac = Resources.Load(language + "/intro") as AudioClip; //loading intro.mp3
specific for user's language (note the file file extension should not be used)
        if (ac==null)
        {
            ac = Resources.Load("English/intro") as AudioClip; //fallback to the english
version for any unsupported language
        }
        transform.GetComponent<AudioSource>().clip = ac;
        transform.GetComponent<AudioSource>().Play();
    }
}
```

Recursos 101

Introducción

Unity tiene algunas carpetas 'especialmente nombradas' que permiten una variedad de usos. Una de estas carpetas se llama 'Recursos'

La carpeta 'Recursos' es una de las DOS formas de cargar activos en tiempo de ejecución en Unity (la otra es AssetBundles (Unity Docs)

La carpeta 'Recursos' puede residir en cualquier lugar dentro de su carpeta de Activos, y puede tener varias carpetas llamadas Recursos. El contenido de todas las carpetas de 'Recursos' se combinan durante el tiempo de compilación.

La forma principal de cargar un activo desde una carpeta de Recursos es usar la función Resources.Load . Esta función toma un parámetro de cadena que le permite especificar la ruta del archivo en **relación** con la carpeta Recursos. Tenga en cuenta que NO necesita especificar extensiones de archivo mientras carga un activo

```
public class ResourcesSample : MonoBehaviour {

    void Start () {
        //The following line will load a TextAsset named 'foobar' which was previously place
under 'Assets/Resources/Stackoverflow/foobar.txt'
        //Note the absence of the '.txt' extension! This is important!

        var text = Resources.Load<TextAsset>("Stackoverflow/foobar").text;
        Debug.Log(string.Format("The text file had this in it :: {0}", text));
    }
}
```

Los objetos que se componen de varios objetos también se pueden cargar desde Recursos. Los ejemplos son tales objetos son modelos 3D con texturas al horno, o un sprite múltiple.

```
//This example will load a multiple sprite texture from Resources named "A_Multiple_Sprite"
var sprites = Resources.LoadAll("A_Multiple_Sprite") as Sprite[];
```

Poniendolo todo junto

Esta es una de mis clases de ayuda que utilizo para cargar todos los sonidos de cualquier juego. Puede adjuntar esto a cualquier GameObject en una escena y cargará los archivos de audio especificados desde la carpeta 'Recursos / Sonidos'

```
public class SoundManager : MonoBehaviour {

    void Start () {

        //An array of all sounds you want to load
        var filesToLoad = new string[] { "Foo", "Bar" };

        //Loop over the array, attach an Audio source for each sound clip and assign the
        //clip property.
        foreach(var file in filesToLoad) {
            var soundClip = Resources.Load<AudioClip>("Sounds/" + file);
            var audioSource = gameObject.AddComponent<AudioSource>();
            audioSource.clip = soundClip;
        }
    }
}
```

Notas finales

1. Unity es inteligente cuando se trata de incluir activos en su construcción. Cualquier activo que no esté serializado (es decir, utilizado en una escena que se incluye en una compilación) se excluye de una compilación. SIN EMBARGO, esto NO se aplica a ningún activo dentro de la carpeta de Recursos. Por lo tanto, no exagere al agregar recursos a esta carpeta

2. Los activos que se cargan con Resources.Load o Resources.LoadAll se pueden descargar en el futuro utilizando Resources.UnloadUnusedAssets o Resources.UnloadAsset

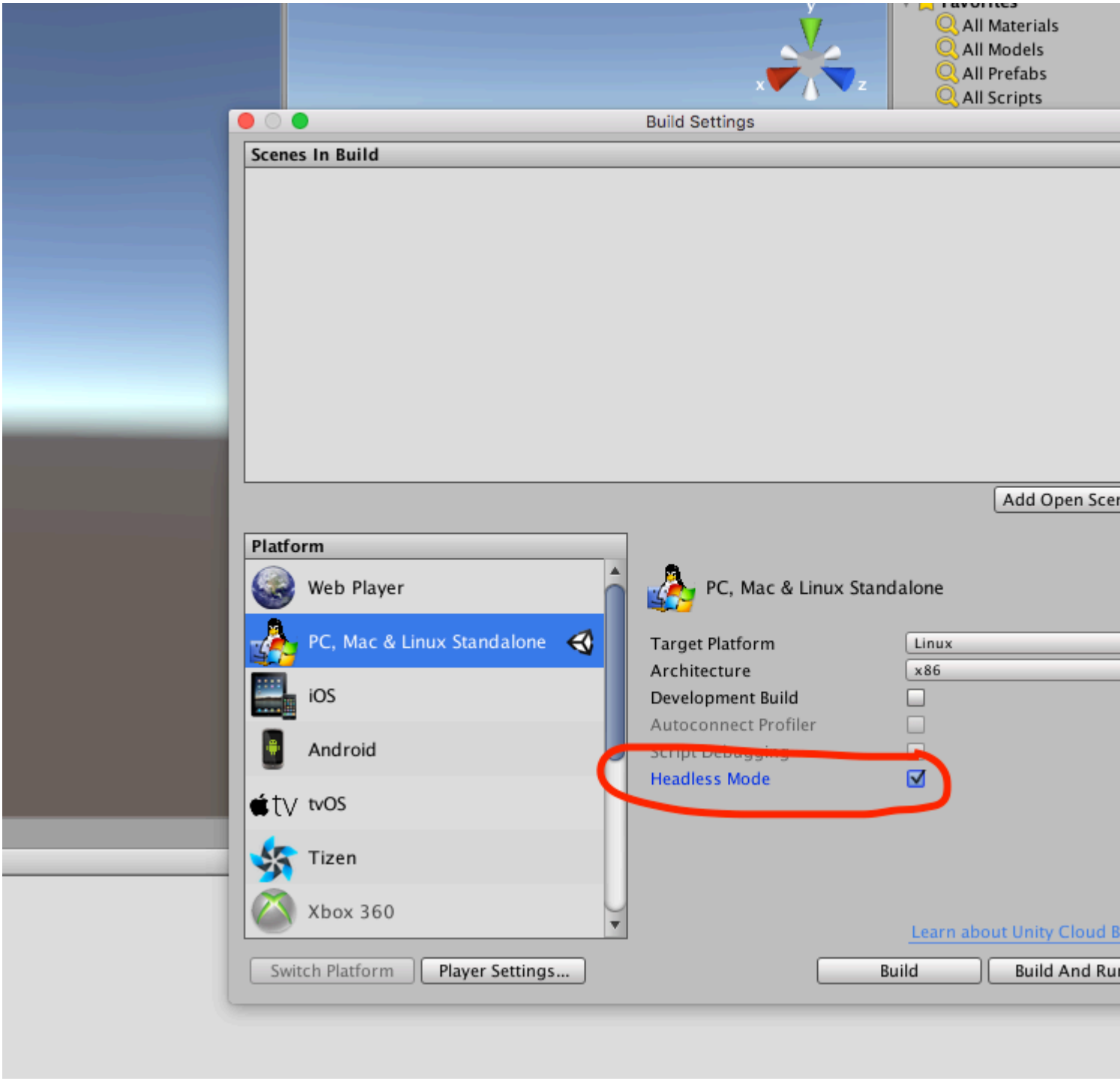
Lea Recursos en línea: https://riptutorial.com/es/unity3d/topic/4070/recursos

Capítulo 29: Redes

Observaciones

Modo sin cabeza en la unidad

Si está construyendo un servidor para implementar en Linux, la configuración de compilación tiene una opción de "Modo sin cabeza". Una compilación de la aplicación con esta opción no muestra nada y no lee la entrada del usuario, que generalmente es lo que queremos para un servidor.



Examples

Creando un servidor, un cliente, y enviando un mensaje.

Unity networking proporciona la API de alto nivel (HLA) para manejar las comunicaciones de red que se abstraen de las implementaciones de bajo nivel.

En este ejemplo, veremos cómo crear un servidor que pueda comunicarse con uno o varios clientes.

El HLA nos permite serializar fácilmente una clase y enviar objetos de esta clase a través de la red.

La Clase que estamos utilizando para serializar.

Esta clase tiene que heredarse de MessageBase, en este ejemplo solo enviaremos una cadena dentro de esta clase.

```
using System;
using UnityEngine.Networking;

public class MyNetworkMessage : MessageBase
{
    public string message;
}
```

Creando un servidor

Creamos un servidor que escucha el puerto 9999, permite un máximo de 10 conexiones y lee objetos de la red de nuestra clase personalizada.

El HLA asocia diferentes tipos de mensajes a una identificación. Hay tipos de mensajes predeterminados definidos en la clase MsgType de Unity Networking. Por ejemplo, el tipo de conexión tiene id 32 y se llama en el servidor cuando un cliente se conecta a él, o en el cliente cuando se conecta a un servidor. Puede registrar manejadores para gestionar los diferentes tipos de mensajes.

Cuando está enviando una clase personalizada, como nuestro caso, definimos un controlador con una nueva identificación asociada a la clase que estamos enviando a través de la red.

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;

public class Server : MonoBehaviour {

    int port = 9999;
```

```
int maxConnections = 10;

// The id we use to identify our messages and register the handler
short messageID = 1000;

// Use this for initialization
void Start () {
    // Usually the server doesn't need to draw anything on the screen
    Application.runInBackground = true;
    CreateServer();
}

void CreateServer() {
    // Register handlers for the types of messages we can receive
    RegisterHandlers ();

    var config = new ConnectionConfig ();
    // There are different types of channels you can use, check the official documentation
    config.AddChannel (QosType.ReliableFragmented);
    config.AddChannel (QosType.UnreliableFragmented);

    var ht = new HostTopology (config, maxConnections);

    if (!NetworkServer.Configure (ht)) {
        Debug.Log ("No server created, error on the configuration definition");
        return;
    } else {
        // Start listening on the defined port
        if(NetworkServer.Listen (port))
            Debug.Log ("Server created, listening on port: " + port);
        else
            Debug.Log ("No server created, could not listen to the port: " + port);
    }
}

void OnApplicationQuit() {
    NetworkServer.Shutdown ();
}

private void RegisterHandlers () {
    // Unity have different Messages types defined in MsgType
    NetworkServer.RegisterHandler (MsgType.Connect, OnClientConnected);
    NetworkServer.RegisterHandler (MsgType.Disconnect, OnClientDisconnected);

    // Our message use his own message type.
    NetworkServer.RegisterHandler (messageID, OnMessageReceived);
}

private void RegisterHandler(short t, NetworkMessageDelegate handler) {
    NetworkServer.RegisterHandler (t, handler);
}

void OnClientConnected(NetworkMessage netMessage)
{
    // Do stuff when a client connects to this server

    // Send a thank you message to the client that just connected
    MyNetworkMessage messageContainer = new MyNetworkMessage();
    messageContainer.message = "Thanks for joining!";

    // This sends a message to a specific client, using the connectionId
```

```
        NetworkServer.SendToClient(netMessage.conn.connectionId,messageID,messageContainer);

        // Send a message to all the clients connected
        messageContainer = new MyNetworkMessage();
        messageContainer.message = "A new player has conencted to the server";

        // Broadcast a message a to everyone connected
        NetworkServer.SendToAll(messageID,messageContainer);
    }

    void OnClientDisconnected(NetworkMessage netMessage)
    {
        // Do stuff when a client dissconnects
    }

    void OnMessageReceived(NetworkMessage netMessage)
    {
        // You can send any object that inherence from MessageBase
        // The client and server can be on different projects, as long as the MyNetworkMessage
or the class you are using have the same implementation on both projects
        // The first thing we do is deserialize the message to our custom type
        var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();
        Debug.Log("Message received: " + objectMessage.message);

    }
}
```

El cliente

Ahora creamos un cliente

```
using System;
using UnityEngine;
using UnityEngine.Networking;


public class Client : MonoBehaviour
{
    int port = 9999;
    string ip = "localhost";

    // The id we use to identify our messages and register the handler
    short messageID = 1000;

    // The network client
    NetworkClient client;

    public Client ()
    {
        CreateClient();
    }

    void CreateClient()
    {
        var config = new ConnectionConfig ();

        // Config the Channels we will use
        config.AddChannel (QosType.ReliableFragmented);
```

```
        config.AddChannel (QosType.UnreliableFragmented);

        // Create the client ant attach the configuration
        client = new NetworkClient ();
        client.Configure (config,1);

        // Register the handlers for the different network messages
        RegisterHandlers();

        // Connect to the server
        client.Connect (ip, port);
    }

    // Register the handlers for the different message types
    void RegisterHandlers () {

        // Unity have different Messages types defined in MsgType
        client.RegisterHandler (messageID, OnMessageReceived);
        client.RegisterHandler(MsgType.Connect, OnConnected);
        client.RegisterHandler(MsgType.Disconnect, OnDisconnected);
    }

    void OnConnected(NetworkMessage message) {
        // Do stuff when connected to the server

        MyNetworkMessage messageContainer = new MyNetworkMessage();
        messageContainer.message = "Hello server!";

        // Say hi to the server when connected
        client.Send(messageID,messageContainer);
    }

    void OnDisconnected(NetworkMessage message) {
        // Do stuff when disconnected to the server
    }

    // Message received from the server
    void OnMessageReceived(NetworkMessage netMessage)
    {
        // You can send any object that inherence from MessageBase
        // The client and server can be on different projects, as long as the MyNetworkMessage
or the class you are using have the same implementation on both projects
        // The first thing we do is deserialize the message to our custom type
        var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();

        Debug.Log("Message received: " + objectMessage.message);
    }
}
```

Lea Redes en línea: https://riptutorial.com/es/unity3d/topic/5671/redes

Capítulo 30: ScriptableObject

Observaciones

ScriptableObjects con AssetBundles

Preste atención cuando agregue prefabs a AssetBundles si contienen referencias a ScriptableObjects. Dado que los ScriptableObjects son esencialmente activos, Unity crea duplicados de ellos antes de agregarlos a AssetBundles, lo que puede resultar en un comportamiento no deseado durante el tiempo de ejecución.

Cuando carga un GameObject de este tipo desde un AssetBundle, puede ser necesario reinyectar los activos de ScriptableObject en los scripts cargados, reemplazando los empaquetados. Ver Inyección de dependencia.

Examples

Introducción

Los ScriptableObjects son objetos serializados que no están vinculados a escenas o juegos como MonoBehaviours. Para decirlo de una manera, son datos y métodos vinculados a archivos de activos dentro de su proyecto. Estos recursos de ScriptableObject se pueden pasar a MonoBehaviours u otros ScriptableObjects, donde se puede acceder a sus métodos públicos.

Debido a su naturaleza como activos serializados, constituyen excelentes clases de administrador y fuentes de datos.

Creación de activos ScriptableObject

A continuación se muestra una implementación simple de ScriptableObject.

```
using UnityEngine;

[CreateAssetMenu(menuName = "StackOverflow/Examples/MyScriptableObject")]
public class MyScriptableObject : ScriptableObject
{
    [SerializeField]
    int mySerializedNumber;

    int helloWorldCount = 0;

    public void HelloWorld()
    {
        helloWorldCount++;
        Debug.LogFormat("Hello! My number is {0}.", mySerializedNumber);
        Debug.LogFormat("I have been called {0} times.", helloWorldCount);
```

```
    }
}
```

Al agregar el atributo `CreateAssetMenu` a la clase, Unity lo incluirá en el submenú **Activos / Crear** . En este caso, se encuentra en **Activos / Crear / StackOverflow / Ejemplos** .

Una vez creadas, las instancias de ScriptableObject se pueden pasar a otros scripts y ScriptableObjects a través del Inspector.

```
using UnityEngine;

public class SampleScript : MonoBehaviour {

    [SerializeField]
    MyScriptableObject myScriptableObject;

    void OnEnable()
    {
        myScriptableObject.HelloWorld();
    }
}
```

Crear instancias de ScriptableObject a través de código

Crea nuevas instancias de ScriptableObject a través de `ScriptableObject.CreateInstance<T>()`

```
T obj = ScriptableObject.CreateInstance<T>();
```

Donde `T` extiende `ScriptableObject` .

> No cree ScriptableObjects llamando a sus constructores, es decir. `new ScriptableObject()` .

La creación de ScriptableObjects por código durante el tiempo de ejecución rara vez se requiere porque su uso principal es la serialización de datos. También podrías usar clases estándar en este punto. Es más común cuando se trata de extensiones de editor de secuencias de comandos.

ScriptableObjects se serializan en el editor incluso en PlayMode

Se debe tener especial cuidado al acceder a campos serializados en una instancia de ScriptableObject.

Si un campo se marca como `public` o se serializa a través de `SerializeField` , el cambio de su valor es permanente. No se reinician al salir de playmode como lo hacen MonoBehaviours. Esto puede ser útil a veces, pero también puede hacer un desastre.

Debido a esto, es mejor hacer que los campos serializados sean de solo lectura y evitar los campos públicos por completo.

```
public class MyScriptableObject : ScriptableObject
{
```

```
    [SerializeField]
    int mySerializedValue;

    public int MySerializedValue
    {
        get { return mySerializedValue; }
    }
}
```

Si desea almacenar valores públicos en un objeto Scriptable que se restablecen entre sesiones de juego, considere usar el siguiente patrón.

```
public class MyScriptableObject : ScriptableObject
{
    // Private fields are not serialized and will reset to default on reset
    private int mySerializedValue;

    public int MySerializedValue
    {
        get { return mySerializedValue; }
        set { mySerializedValue = value; }
    }
}
```

Encuentra ScriptableObjects existentes durante el tiempo de ejecución

Para encontrar ScriptableObjects *activos* durante el tiempo de ejecución, puede usar `Resources.FindObjectsOfTypeAll()` .

```
T[] instances = Resources.FindObjectsOfTypeAll<T>();
```

Donde `T` es el tipo de instancia de ScriptableObject que está buscando. *Activo* significa que se ha cargado en la memoria de alguna forma antes.

Este método es muy lento, así que recuerde almacenar en caché el valor de retorno y evite llamarlo con frecuencia. Hacer referencia a ScriptableObjects directamente en sus scripts debe ser su opción preferida.

> *Sugerencia:* puede mantener sus propias colecciones de instancias para búsquedas más rápidas. Haga que sus ScriptableObjects se registren en una colección compartida durante `OnEnable()` .

Lea ScriptableObject en línea: https://riptutorial.com/es/unity3d/topic/3434/scriptableobject

Capítulo 31: Singletons en la unidad

Observaciones

Si bien hay escuelas de pensamiento que presentan argumentos convincentes sobre por qué el uso sin restricciones de Singletons es una mala idea, por ejemplo, Singleton en gameprogrammingpatterns.com , hay ocasiones en las que es posible que desee conservar un GameObject en Unity en varias escenas (por ejemplo, para una música de fondo perfecta) mientras se asegura que no puede existir más de una instancia; un caso de uso perfecto para un Singleton.

Al agregar este script a un GameObject, una vez que se haya creado una instancia (por ejemplo, al incluirlo en cualquier lugar de una escena) permanecerá activo en todas las escenas, y solo existirá una instancia.

Las instancias de ScriptableObject (UnityDoc) proporcionan una alternativa válida a Singletons para algunos casos de uso. Si bien no aplican de manera implícita la regla de instancia única, conservan su estado entre escenas y juegan bien con el proceso de serialización de Unity. También promueven la inversión de control a medida que las dependencias se inyectan a través del editor .

```
// MyAudioManager.cs
using UnityEngine;

[CreateAssetMenu] // Remember to create the instance in editor
public class MyAudioManager : ScriptableObject {
    public void PlaySound() {}
}
```

```
// MyGameObject.cs
using UnityEngine;

public class MyGameObject : MonoBehaviour
{
    [SerializeField]
    MyAudioManager audioManager; //Insert through Inspector

    void OnEnable()
    {
        audioManager.PlaySound();
    }
}
```

Otras lecturas

- Implementación Singleton en C #

Examples

Implementación utilizando RuntimeInitializeOnLoadMethodAttribute

Desde **Unity 5.2.5** es posible usar RuntimeInitializeOnLoadMethodAttribute para ejecutar la lógica de inicialización sin pasar por el orden de ejecución de MonoBehaviour . Proporciona una manera de crear una implementación más limpia y robusta:

```
using UnityEngine;

sealed class GameDirector : MonoBehaviour
{
    // Because of using RuntimeInitializeOnLoadMethod attribute to find/create and
    // initialize the instance, this property is accessible and
    // usable even in Awake() methods.
    public static GameDirector Instance
    {
        get; private set;
    }

    // Thanks to the attribute, this method is executed before any other MonoBehaviour
    // logic in the game.
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    static void OnRuntimeMethodLoad()
    {
        var instance = FindObjectOfType<GameDirector>();

        if (instance == null)
            instance = new GameObject("Game Director").AddComponent<GameDirector>();

        DontDestroyOnLoad(instance);

        Instance = instance;
    }

    // This Awake() will be called immediately after AddComponent() execution
    // in the OnRuntimeMethodLoad(). In other words, before any other MonoBehaviour's
    // in the scene will begin to initialize.
    private void Awake()
    {
        // Initialize non-MonoBehaviour logic, etc.
        Debug.Log("GameDirector.Awake()", this);
    }
}
```

El orden de ejecución resultante:

1. `GameDirector.OnRuntimeMethodLoad()` comenzó ...
2. `GameDirector.Awake()`
3. `GameDirector.OnRuntimeMethodLoad()` completado.
4. `OtherMonoBehaviour1.Awake()`
5. `OtherMonoBehaviour2.Awake()` , etc.

Un sencillo Singleton MonoBehaviour en Unity C

En este ejemplo, una instancia estática privada de la clase se declara al principio.

El valor de un campo estático se comparte entre las instancias, por lo que si se crea una nueva instancia de esta clase, `if` encontrará una referencia al primer objeto Singleton, destruyendo la nueva instancia (o su objeto del juego).

```
using UnityEngine;

public class SingletonExample : MonoBehaviour {

    private static SingletonExample _instance;

    void Awake(){

        if (_instance == null){

            _instance = this;
            DontDestroyOnLoad(this.gameObject);

            //Rest of your Awake code

        } else {
            Destroy(this);
        }
    }

    //Rest of your class code

}
```

Unidad avanzada de Singleton

Este ejemplo combina múltiples variantes de singletons MonoBehaviour que se encuentran en Internet en uno solo y le permite cambiar su comportamiento según los campos estáticos globales.

Este ejemplo se probó con Unity 5. Para usar este singleton, todo lo que necesita hacer es extenderlo de la siguiente manera: `public class MySingleton : Singleton<MySingleton> {}`. También es posible que deba anular `AwakeSingleton` para usarlo en lugar de `Awake` habitual. Para realizar más ajustes, cambie los valores predeterminados de los campos estáticos como se describe a continuación.

---

1. Esta aplicación hace uso de DisallowMultipleComponent atributo para mantener una instancia por GameObject.
2. Esta clase es abstracta y solo se puede ampliar. También contiene un método virtual, `AwakeSingleton` que debe ser anulado en lugar de implementar `Awake` normal.
3. Esta implementación es segura para subprocesos.
4. Este singleton está optimizado. Al utilizar la `instantiated` lugar de la verificación nula de la instancia, evitamos la sobrecarga que conlleva la implementación de Unity del operador `==` . (Leer más)
5. Esta implementación no permite ninguna llamada a la instancia de singleton cuando está a

punto de ser destruida por Unity.

6. Este singleton viene con las siguientes opciones:

- `FindInactive` : si hay que buscar otras instancias de componentes del mismo tipo adjuntos a GameObject inactivo.
- `Persist` : mantener el componente vivo entre escenas.
- `DestroyOthers` : si destruir cualquier otro componente del mismo tipo y conservar solo uno.
- `Lazy` : si se debe establecer la instancia de singleton "sobre la marcha" (en `Awake`) o solo "a pedido" (cuando se llama getter).

```
using UnityEngine;

[DisallowMultipleComponent]
public abstract class Singleton<T> : MonoBehaviour where T : Singleton<T>
{
    private static volatile T instance;
    // thread safety
    private static object _lock = new object();
    public static bool FindInactive = true;
    // Whether or not this object should persist when loading new scenes. Should be set in
Init().
    public static bool Persist;
    // Whether or not destory other singleton instances if any. Should be set in Init().
    public static bool DestroyOthers = true;
    // instead of heavy comparision (instance != null)
    // http://blogs.unity3d.com/2014/05/16/custom-operator-should-we-keep-it/
    private static bool instantiated;

    private static bool applicationIsQuitting;

    public static bool Lazy;

    public static T Instance
    {
        get
        {
            if (applicationIsQuitting)
            {
                Debug.LogWarningFormat("[Singleton] Instance '{0}' already destroyed on
application quit. Won't create again - returning null.", typeof(T));
                return null;
            }
            lock (_lock)
            {
                if (!instantiated)
                {
                    Object[] objects;
                    if (FindInactive) { objects = Resources.FindObjectsOfTypeAll(typeof(T)); }
                    else { objects = FindObjectsOfType(typeof(T)); }
                    if (objects == null || objects.Length < 1)
                    {
                        GameObject singleton = new GameObject();
                        singleton.name = string.Format("{0} [Singleton]", typeof(T));
                        Instance = singleton.AddComponent<T>();
                        Debug.LogWarningFormat("[Singleton] An Instance of '{0}' is needed in
the scene, so '{1}' was created{2}", typeof(T), singleton.name, Persist ? " with
DontDestoryOnLoad." : ".");
                    }
```

```
                    else if (objects.Length >= 1)
                    {
                        Instance = objects[0] as T;
                        if (objects.Length > 1)
                        {
                            Debug.LogWarningFormat("[Singleton] {0} instances of '{1}'!",
objects.Length, typeof(T));
                            if (DestroyOthers)
                            {
                                for (int i = 1; i < objects.Length; i++)
                                {
                                    Debug.LogWarningFormat("[Singleton] Deleting extra '{0}'
instance attached to '{1}'", typeof(T), objects[i].name);
                                    Destroy(objects[i]);
                                }
                            }
                        }
                        return instance;
                    }
                }
                return instance;
            }
        }
        protected set
        {
            instance = value;
            instantiated = true;
            instance.AwakeSingleton();
            if (Persist) { DontDestroyOnLoad(instance.gameObject); }
        }
    }

    // if Lazy = false and gameObject is active this will set instance
    // unless instance was called by another Awake method
    private void Awake()
    {
        if (Lazy) { return; }
        lock (_lock)
        {
            if (!instantiated)
            {
                Instance = this as T;
            }
            else if (DestroyOthers && Instance.GetInstanceID() != GetInstanceID())
            {
                Debug.LogWarningFormat("[Singleton] Deleting extra '{0}' instance attached to
'{1}'", typeof(T), name);
                Destroy(this);
            }
        }
    }

    // this might be called for inactive singletons before Awake if FindInactive = true
    protected virtual void AwakeSingleton() {}

    protected virtual void OnDestroy()
    {
        applicationIsQuitting = true;
        instantiated = false;
    }
}
```

Implementación Singleton a través de clase base

En proyectos que cuentan con varias clases singleton (como suele ser el caso), puede ser limpio y conveniente abstraer el comportamiento singleton a una clase base:

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

public abstract class MonoBehaviourSingleton<T> : MonoBehaviour {

    private static Dictionary<Type, object> _singletons
        = new Dictionary<Type, object>();

    public static T Instance {
        get {
            return (T)_singletons[typeof(T)];
        }
    }

    void OnEnable() {
        if (_singletons.ContainsKey(GetType())) {
            Destroy(this);
        } else {
            _singletons.Add(GetType(), this);
            DontDestroyOnLoad(this);
        }
    }
}
```

Un MonoBehaviour puede entonces implementar el patrón de singleton al extender MonoBehaviourSingleton. Este enfoque permite utilizar el patrón con una huella mínima en el propio Singleton:

```
using UnityEngine;
using System.Collections;

public class SingletonImplementation : MonoBehaviourSingleton<SingletonImplementation> {

    public string Text= "String Instance";

    // Use this for initialisation
    IEnumerator Start () {
        var demonstration = "SingletonImplementation.Start()\n" +
                            "Note that the this text logs only once and\n"
                            "only one class instance is allowed to exist.";
        Debug.Log(demonstration);
        yield return new WaitForSeconds(2f);
        var secondInstance = new GameObject();
        secondInstance.AddComponent<SingletonImplementation>();
    }

}
```

Tenga en cuenta que uno de los beneficios del patrón singleton es que se puede acceder a una referencia a la instancia de forma estática:

```
// Logs: String Instance
Debug.Log(SingletonImplementation.Instance.Text);
```

Sin embargo, tenga en cuenta que esta práctica debe minimizarse para reducir el acoplamiento. Este enfoque también tiene un ligero costo de rendimiento debido al uso del Diccionario, pero como esta colección puede contener solo una instancia de cada clase de singleton, la compensación en términos del principio DRY (No se repita), legibilidad y la conveniencia es pequeña

Patrón Singleton utilizando el sistema Entity-Component de Unitys

La idea central es usar GameObjects para representar singletons, que tiene múltiples ventajas:

- Mantiene la complejidad al mínimo, pero admite conceptos como la inyección de dependencia.
- Los Singletons tienen un ciclo de vida normal de Unity como parte del sistema Entity-Component
- Los Singletons pueden cargarse y almacenarse de manera local en lugares donde se necesitan regularmente (por ejemplo, en ciclos de actualización)
- No se necesitan campos estáticos
- No es necesario modificar los MonoBehaviours / Componentes existentes para usarlos como Singletons
- Fácil de restablecer (solo destruye el Singletons GameObject), se cargará de nuevo en el próximo uso
- Fácil de inyectar simulacros (solo inicialízalo con el simulacro antes de usarlo)
- Inspección y configuración usando el editor normal de Unity y puede ocurrir ya en el tiempo del editor (Captura de pantalla de un Singleton accesible en el editor de Unity)

Test.cs (que usa el ejemplo singleton):

```
using UnityEngine;
using UnityEngine.Assertions;

public class Test : MonoBehaviour {
    void Start() {
        ExampleSingleton singleton = ExampleSingleton.instance;
        Assert.IsNotNull(singleton); // automatic initialization on first usage
        Assert.AreEqual("abc", singleton.myVar1);
        singleton.myVar1 = "123";
        // multiple calls to instance() return the same object:
        Assert.AreEqual(singleton, ExampleSingleton.instance);
        Assert.AreEqual("123", ExampleSingleton.instance.myVar1);
    }
}
```

ExampleSingleton.cs (que contiene un ejemplo y la clase Singleton real):

```
using UnityEngine;
using UnityEngine.Assertions;

public class ExampleSingleton : MonoBehaviour {
```

```
    public static ExampleSingleton instance { get { return Singleton.get<ExampleSingleton>();
} }
    public string myVar1 = "abc";
    public void Start() { Assert.AreEqual(this, instance, "Singleton more than once in
scene"); }
}

/// <summary> Helper that turns any MonBehaviour or other Component into a Singleton
</summary>
public static class Singleton {
    public static T get<T>() where T : Component {
        return GetOrAddGo("Singletons").GetOrAddChild("" + typeof(T)).GetOrAddComponent<T>();
    }
    private static GameObject GetOrAddGo(string goName) {
        var go = GameObject.Find(goName);
        if (go == null) { return new GameObject(goName); }
        return go;
    }
}

public static class GameObjectExtensionMethods {
    public static GameObject GetOrAddChild(this GameObject parentGo, string childName) {
        var childGo = parentGo.transform.FindChild(childName);
        if (childGo != null) { return childGo.gameObject; } // child found, return it
        var newChild = new GameObject(childName);        // no child found, create it
        newChild.transform.SetParent(parentGo.transform, false); // add it to parent
        return newChild;
    }

    public static T GetOrAddComponent<T>(this GameObject parentGo) where T : Component {
        var comp = parentGo.GetComponent<T>();
        if (comp == null) { return parentGo.AddComponent<T>(); }
        return comp;
    }
}
```

Los dos métodos de extensión para GameObject también son útiles en otras situaciones, si no los necesitas, muévelos dentro de la clase Singleton y hazlos privados.

Clase de Singleton basada en MonoBehaviour y ScriptableObject

La mayoría de los ejemplos de Singleton usan MonoBehaviour como la clase base. La principal desventaja es que esta clase Singleton solo vive durante el tiempo de ejecución. Esto tiene algunos inconvenientes:

- No hay forma de editar directamente los campos singleton que no sea cambiar el código.
- No hay forma de almacenar una referencia a otros activos en el Singleton.
- No hay forma de configurar el singleton como el destino de un evento de UI de Unity. Termino usando lo que llamo "Componentes Proxy", su única propuesta es tener métodos de 1 línea que llamen "GameManager.Instance.SomeGlobalMethod ()".

Como se señaló en los comentarios, hay implementaciones que intentan resolver esto utilizando ScriptableObjects como clase base pero pierden los beneficios de tiempo de ejecución de MonoBehaviour. Esta implementación resuelve estos problemas utilizando un objeto ScriptableObject como clase base y un MonoBehavior asociado durante el tiempo de ejecución:

- Es un activo por lo que sus propiedades pueden actualizarse en el editor como cualquier otro activo de Unity.
- Juega muy bien con el proceso de serialización de Unity.
- Es posible asignar referencias en el singleton a otros activos desde el editor (las dependencias se inyectan a través del editor).
- Los eventos de Unity pueden llamar directamente a los métodos en Singleton.
- Puede llamarlo desde cualquier lugar en la base de código usando "SingletonClassName.Instance"
- Tiene acceso para ejecutar eventos MonoBehaviour de tiempo y métodos como: Actualizar, Despertar, Iniciar, FixedUpdate, StartCoroutine, etc.

```
/*************************************************************
 * Better Singleton by David Darias
 * Use as you like - credit where due would be appreciated :D
 * Licence: WTFPL V2, Dec 2014
 * Tested on Unity v5.6.0 (should work on earlier versions)
 * 03/02/2017 - v1.1
 * ************************************************************/

using System;
using UnityEngine;
using SingletonScriptableObjectNamespace;

public class SingletonScriptableObject<T> :
SingletonScriptableObjectNamespace.BehaviourScriptableObject where T :
SingletonScriptableObjectNamespace.BehaviourScriptableObject
{
    //Private reference to the scriptable object
    private static T _instance;
    private static bool _instantiated;
    public static T Instance
    {
        get
        {
            if (_instantiated) return _instance;
            var singletonName = typeof(T).Name;
            //Look for the singleton on the resources folder
            var assets = Resources.LoadAll<T>("");
            if (assets.Length > 1) Debug.LogError("Found multiple " + singletonName + "s on
the resources folder. It is a Singleton ScriptableObject, there should only be one.");
            if (assets.Length == 0)
            {
                _instance = CreateInstance<T>();
                Debug.LogError("Could not find a " + singletonName + " on the resources
folder. It was created at runtime, therefore it will not be visible on the assets folder and
it will not persist.");
            }
            else _instance = assets[0];
            _instantiated = true;
            //Create a new game object to use as proxy for all the MonoBehaviour methods
            var baseObject = new GameObject(singletonName);
            //Deactivate it before adding the proxy component. This avoids the execution of
the Awake method when the the proxy component is added.
            baseObject.SetActive(false);
            //Add the proxy, set the instance as the parent and move to DontDestroyOnLoad
scene
            SingletonScriptableObjectNamespace.BehaviourProxy proxy =
baseObject.AddComponent<SingletonScriptableObjectNamespace.BehaviourProxy>();
```

```
            proxy.Parent = _instance;
            Behaviour = proxy;
            DontDestroyOnLoad(Behaviour.gameObject);
            //Activate the proxy. This will trigger the MonoBehaviourAwake.
            proxy.gameObject.SetActive(true);
            return _instance;
        }
    }
    //Use this reference to call MonoBehaviour specific methods (for example StartCoroutine)
    protected static MonoBehaviour Behaviour;
    public static void BuildSingletonInstance() {
SingletonScriptableObjectNamespace.BehaviourScriptableObject i = Instance; }
    private void OnDestroy(){ _instantiated = false; }
}

// Helper classes for the SingletonScriptableObject
namespace SingletonScriptableObjectNamespace
{
    #if UNITY_EDITOR
    //Empty custom editor to have cleaner UI on the editor.
    using UnityEditor;
    [CustomEditor(typeof(BehaviourProxy))]
    public class BehaviourProxyEditor : Editor
    {
        public override void OnInspectorGUI(){}
    }

    #endif

    public class BehaviourProxy : MonoBehaviour
    {
        public IBehaviour Parent;

        public void Awake() { if (Parent != null) Parent.MonoBehaviourAwake(); }
        public void Start() { if (Parent != null) Parent.Start(); }
        public void Update() { if (Parent != null) Parent.Update(); }
        public void FixedUpdate() { if (Parent != null) Parent.FixedUpdate(); }
    }

    public interface IBehaviour
    {
        void MonoBehaviourAwake();
        void Start();
        void Update();
        void FixedUpdate();
    }

    public class BehaviourScriptableObject : ScriptableObject, IBehaviour
    {
        public void Awake() { ScriptableObjectAwake(); }
        public virtual void ScriptableObjectAwake() { }
        public virtual void MonoBehaviourAwake() { }
        public virtual void Start() { }
        public virtual void Update() { }
        public virtual void FixedUpdate() { }
    }
}
```

Aquí hay un ejemplo de clase Singleton de GameManager usando SingletonScriptableObject (con muchos comentarios):

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

//this attribute is optional but recommended. It will allow the creation of the singleton via
the asset menu.
//the singleton asset should be on the Resources folder.
[CreateAssetMenu(fileName = "GameManager", menuName = "Game Manager", order = 0)]
public class GameManager : SingletonScriptableObject<GameManager> {

    //any properties as usual
    public int Lives;
    public int Points;

    //optional (but recommended)
    //this method will run before the first scene is loaded. Initializing the singleton here
    //will allow it to be ready before any other GameObjects on every scene and will
    //will prevent the "initialization on first usage".
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    public static void BeforeSceneLoad() { BuildSingletonInstance(); }

    //optional,
    //will run when the Singleton Scriptable Object is first created on the assets.
    //Usually this happens on edit mode, not runtime. (the override keyword is mandatory for
this to work)
    public override void ScriptableObjectAwake(){
        Debug.Log(GetType().Name + " created." );
    }

    //optional,
    //will run when the associated MonoBehavioir awakes. (the override keyword is mandatory
for this to work)
    public override void MonoBehaviourAwake(){
        Debug.Log(GetType().Name + " behaviour awake." );

        //A coroutine example:
        //Singleton Objects do not have coroutines.
        //if you need to use coroutines use the atached MonoBehaviour
        Behaviour.StartCoroutine(SimpleCoroutine());
    }

    //any methods as usual
    private IEnumerator SimpleCoroutine(){
        while(true){
            Debug.Log(GetType().Name + " coroutine step." );
            yield return new WaitForSeconds(3);
        }
    }

    //optional,
    //Classic runtime Update method (the override keyword is mandatory for this to work).
    public override void Update(){

    }

    //optional,
    //Classic runtime FixedUpdate method (the override keyword is mandatory for this to work).
    public override void FixedUpdate(){

    }
```

```
}

/*
*  Notes:
*  - Remember that you have to create the singleton asset on edit mode before using it. You
have to put it on the Resources folder and of course it should be only one.
*  - Like other Unity Singleton this one is accessible anywhere in your code using the
"Instance" property i.e: GameManager.Instance
*/
```

Lea Singletons en la unidad en línea: https://riptutorial.com/es/unity3d/topic/2137/singletons-en-la-unidad

Capítulo 32: Sistema de audio

Introducción

Esta es una documentación sobre la reproducción de audio en Unity3D.

Examples

Clase de audio - Reproducir audio

```
using UnityEngine;

public class Audio : MonoBehaviour {
    AudioSource audioSource;
    AudioClip audioClip;

    void Start() {
        audioClip = (AudioClip)Resources.Load("Audio/Soundtrack");
        audioSource.clip = audioClip;
        if (!audioSource.isPlaying) audioSource.Play();
    }
```

Lea Sistema de audio en línea: https://riptutorial.com/es/unity3d/topic/8064/sistema-de-audio

Capítulo 33: Sistema de entrada

Examples

Leyendo la lectura de teclas y la diferencia entre GetKey, GetKeyDown y GetKeyUp

La entrada debe leer desde la función Actualizar.

Referencia para todas las enumeración de Keycode disponible.

1. Leyendo la tecla presionando con `Input.GetKey` :

`Input.GetKey` devolverá **repetidamente** `true` mientras el usuario mantiene presionada la tecla especificada. Se puede usar para disparar **repetidamente** un arma mientras se mantiene presionada la tecla especificada. A continuación se muestra un ejemplo de disparo automático de bala cuando se mantiene presionada la tecla de espacio. El jugador no tiene que presionar y soltar la tecla una y otra vez.

```
public GameObject bulletPrefab;
public float shootForce = 50f;

void Update()
{
    if (Input.GetKey(KeyCode.Space))
    {
        Debug.Log("Shooting a bullet while SpaceBar is held down");

        //Instantiate bullet
        GameObject bullet = Instantiate(bulletPrefab, transform.position, transform.rotation)
as GameObject;

        //Get the Rigidbody from the bullet then add a force to the bullet
        bullet.GetComponent<Rigidbody>().AddForce(bullet.transform.forward * shootForce);
    }
}
```

2. Tecla de lectura presionando con `Input.GetKeyDown` :

`Input.GetKeyDown` solo se cumplirá una **vez** cuando se presione la tecla especificada. Esta es la diferencia clave entre `Input.GetKey` y `Input.GetKeyDown` . Un ejemplo de uso de su uso es para activar / desactivar una IU o una linterna o un elemento.

```
public Light flashLight;
bool enableFlashLight = false;

void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        //Toggle Light
```

```
        enableFlashLight = !enableFlashLight;
        if (enableFlashLight)
        {
            flashLight.enabled = true;
            Debug.Log("Light Enabled!");
        }
        else
        {
            flashLight.enabled = false;
            Debug.Log("Light Disabled!");
        }
    }
}
```

3. Pulsando la tecla de `Input.GetKeyUp` con `Input.GetKeyUp` :

Esto es exactamente lo contrario de `Input.GetKeyDown` . Se utiliza para detectar cuándo se presiona / levanta la pulsación de tecla. Al igual que `Input.GetKeyDown` , devuelve `true` solo **una vez** . Por ejemplo, puede `enable` luz cuando la tecla se mantiene presionada con `Input.GetKeyDown` luego deshabilitar la luz cuando se suelta la tecla con `Input.GetKeyUp` .

```
public Light flashLight;
void Update()
{
    //Disable Light when Space Key is pressed
    if (Input.GetKeyDown(KeyCode.Space))
    {
        flashLight.enabled = true;
        Debug.Log("Light Enabled!");
    }

    //Disable Light when Space Key is released
    if (Input.GetKeyUp(KeyCode.Space))
    {
        flashLight.enabled = false;
        Debug.Log("Light Disabled!");
    }
}
```

Sensor de acelerómetro de lectura (básico)

`Input.acceleration` se utiliza para leer el sensor del acelerómetro. Devuelve `Vector3` como un resultado que contiene los valores de los ejes `x` , `y` y `z` en el espacio 3D.

```
void Update()
{
    Vector3 acclerometerValue = rawAccelValue();
    Debug.Log("X: " + acclerometerValue.x + "  Y: " + acclerometerValue.y + "  Z: " +
acclerometerValue.z);
}

Vector3 rawAccelValue()
{
    return Input.acceleration;
}
```

Lea el sensor del acelerómetro (avance)

El uso de valores brutos directamente desde el sensor del acelerómetro para mover o rotar un GameObject puede causar problemas como movimientos bruscos o vibraciones. Se recomienda suavizar los valores antes de usarlos. De hecho, los valores del sensor del acelerómetro siempre se deben suavizar antes de usarlos. Esto se puede lograr con un filtro de paso bajo y aquí es donde `Vector3.Lerp` entra en su lugar.

```
//The lower this value, the less smooth the value is and faster Accel is updated. 30 seems
fine for this
const float updateSpeed = 30.0f;

float AccelerometerUpdateInterval = 1.0f / updateSpeed;
float LowPassKernelWidthInSeconds = 1.0f;
float LowPassFilterFactor = 0;
Vector3 lowPassValue = Vector3.zero;

void Start()
{
    //Filter Accelerometer
    LowPassFilterFactor = AccelerometerUpdateInterval / LowPassKernelWidthInSeconds;
    lowPassValue = Input.acceleration;
}

void Update()
{

    //Get Raw Accelerometer values (pass in false to get raw Accelerometer values)
    Vector3 rawAccelValue = filterAccelValue(false);
    Debug.Log("RAW X: " + rawAccelValue.x + "  Y: " + rawAccelValue.y + "  Z: " +
rawAccelValue.z);

    //Get smoothed Accelerometer values (pass in true to get Filtered Accelerometer values)
    Vector3 filteredAccelValue = filterAccelValue(true);
    Debug.Log("FILTERED X: " + filteredAccelValue.x + "  Y: " + filteredAccelValue.y + "  Z: "
+ filteredAccelValue.z);
}

//Filter Accelerometer
Vector3 filterAccelValue(bool smooth)
{
    if (smooth)
        lowPassValue = Vector3.Lerp(lowPassValue, Input.acceleration, LowPassFilterFactor);
    else
        lowPassValue = Input.acceleration;

    return lowPassValue;
}
```

Sensor de acelerómetro de lectura (precisión)

Lea el sensor del acelerómetro con precisión.

Este ejemplo asigna memoria:

```
void Update()
```

```
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
    Debug.Log("PRECISE X: " + accelValue.x + "  Y: " + accelValue.y + "  Z: " + accelValue.z);

}

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    foreach (AccelerationEvent tempAccelEvent in Input.accelerationEvents)
    {
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}
```

Este ejemplo **no** asigna memoria:

```
void Update()
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
    Debug.Log("PRECISE X: " + accelValue.x + "  Y: " + accelValue.y + "  Z: " + accelValue.z);

}

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    for (int i = 0; i < Input.accelerationEventCount; ++i)
    {
        AccelerationEvent tempAccelEvent = Input.GetAccelerationEvent(i);
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}
```

Tenga en cuenta que esto no se filtra. Consulte aquí cómo suavizar los valores del acelerómetro para eliminar el ruido.

Haga clic en el botón del mouse (izquierda, central, derecha) Clics

Estas funciones se utilizan para comprobar los clics del botón del ratón.

- `Input.GetMouseButton(int button);`
- `Input.GetMouseButtonDown(int button);`
- `Input.GetMouseButtonUp(int button);`

Todos toman el mismo parámetro.

- 0 = Clic izquierdo del ratón.
- 1 = clic derecho del ratón.
- 2 = clic medio del ratón.

`GetMouseButton` se usa para detectar cuándo se *mantiene* presionado el botón del mouse.

Devuelve `true` mientras se mantiene presionado el botón del mouse especificado.

```
void Update()
{
    if (Input.GetMouseButton(0))
    {
        Debug.Log("Left Mouse Button Down");
    }

    if (Input.GetMouseButton(1))
    {
        Debug.Log("Right Mouse Button Down");
    }

    if (Input.GetMouseButton(2))
    {
        Debug.Log("Middle Mouse Button Down");
    }
}
```

`GetMouseButtonDown` se usa para detectar cuándo hay un clic del mouse. Devuelve `true` si se presiona **una vez** . No volverá a ser `true` hasta que se suelte el botón del mouse y se presione nuevamente.

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Debug.Log("Left Mouse Button Clicked");
    }

    if (Input.GetMouseButtonDown(1))
    {
        Debug.Log("Right Mouse Button Clicked");
    }

    if (Input.GetMouseButtonDown(2))
    {
        Debug.Log("Middle Mouse Button Clicked");
    }
}
```

`GetMouseButtonUp` se utiliza para detectar cuándo se suelta el botón de mouse específico. Esto solo se devolverá `true` una vez que se suelte el botón del mouse especificado. Para devolver el valor verdadero nuevamente, se debe presionar y soltar nuevamente.

```
void Update()
{
    if (Input.GetMouseButtonUp(0))
    {
        Debug.Log("Left Mouse Button Released");
    }

    if (Input.GetMouseButtonUp(1))
    {
        Debug.Log("Right Mouse Button Released");
    }
```

```
    if (Input.GetMouseButtonUp(2))
    {
        Debug.Log("Middle Mouse Button Released");
    }
}
```

Lea Sistema de entrada en línea: https://riptutorial.com/es/unity3d/topic/3413/sistema-de-entrada

Capítulo 34: Sistema de interfaz de usuario (UI)

Examples

Suscripción al evento en código

De forma predeterminada, uno debe suscribirse al evento utilizando inspector, pero a veces es mejor hacerlo en código. En este ejemplo nos suscribimos para hacer clic en el evento de un botón para manejarlo.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Button))]
public class AutomaticClickHandler : MonoBehaviour
{
    private void Awake()
    {
        var button = this.GetComponent<Button>();
        button.onClick.AddListener(HandleClick);
    }

    private void HandleClick()
    {
        Debug.Log("AutomaticClickHandler.HandleClick()", this);
    }
}
```

Los componentes de la interfaz de usuario suelen proporcionar su escucha principal fácilmente

- Botón: onClick
- Desplegable : onValueChanged
- InputField: onEndEdit , onValidateInput , onValueChanged
- Barra de desplazamiento: onValueChanged
- ScrollRect: onValueChanged
- Control deslizante: onValueChanged
- Alternar: onValueChanged

Añadiendo oyentes del mouse

A veces, desea agregar escuchas en eventos particulares no proporcionados de forma nativa por los componentes, en particular eventos de mouse. Para hacerlo, tendrá que agregarlos usted mismo utilizando un componente `EventTrigger` :

```
using UnityEngine;
using UnityEngine.EventSystems;
```

```
[RequireComponent(typeof( EventTrigger ))]
public class CustomListenersExample : MonoBehaviour
{
    void Start( )
    {
        EventTrigger eventTrigger = GetComponent<EventTrigger>( );
        EventTrigger.Entry entry = new EventTrigger.Entry( );
        entry.eventID = EventTriggerType.PointerDown;
        entry.callback.AddListener( ( data ) => { OnPointerDownDelegate(
(PointerEventData)data ); } );
        eventTrigger.triggers.Add( entry );
    }

    public void OnPointerDownDelegate( PointerEventData data )
    {
        Debug.Log( "OnPointerDownDelegate called." );
    }
}
```

Varios eventID son posibles:

- PunteroEnter
- PunteroExit
- Puntero abajo
- Apuntador
- Puntero hacer clic
- Arrastrar
- soltar
- Voluta
- ActualizarSeleccionado
- Seleccionar
- Deseleccionar
- Movimiento
- InitializePotentialDrag
- BeginDrag
- EndDrag
- Enviar
- Cancelar

Lea Sistema de interfaz de usuario (UI) en línea:
https://riptutorial.com/es/unity3d/topic/2296/sistema-de-interfaz-de-usuario--ui-

Capítulo 35: Sistema de interfaz de usuario gráfico de modo inmediato (IMGUI)

Sintaxis

- Public static void GUILayout.Label (texto de cadena, params GUILayoutOption [] opciones)
- público bool GUILayout.Button estático (texto de cadena, params GUILayoutOption [] opciones)
- cadena estática pública GUILayout.TextArea (texto de cadena, params GUILayoutOption [] opciones)

Examples

GUILayout

Antigua herramienta de sistema de interfaz de usuario, ahora utilizada para crear prototipos o depurar de forma rápida y simple en el juego.

```
void OnGUI ()
{
    GUILayout.Label ("I'm a simple label text displayed in game.");

    if ( GUILayout.Button("CLICK ME") )
    {
        GUILayout.TextArea ("This is a \n
                             multiline comment.")
    }
}
```

La función **GUILayout** funciona dentro de la función **OnGUI** .

Lea Sistema de interfaz de usuario gráfico de modo inmediato (IMGUI) en línea: https://riptutorial.com/es/unity3d/topic/6947/sistema-de-interfaz-de-usuario-grafico-de-modo-inmediato--imgui-

Capítulo 36: Tienda de activos

Examples

Accediendo a la Tienda de Activos

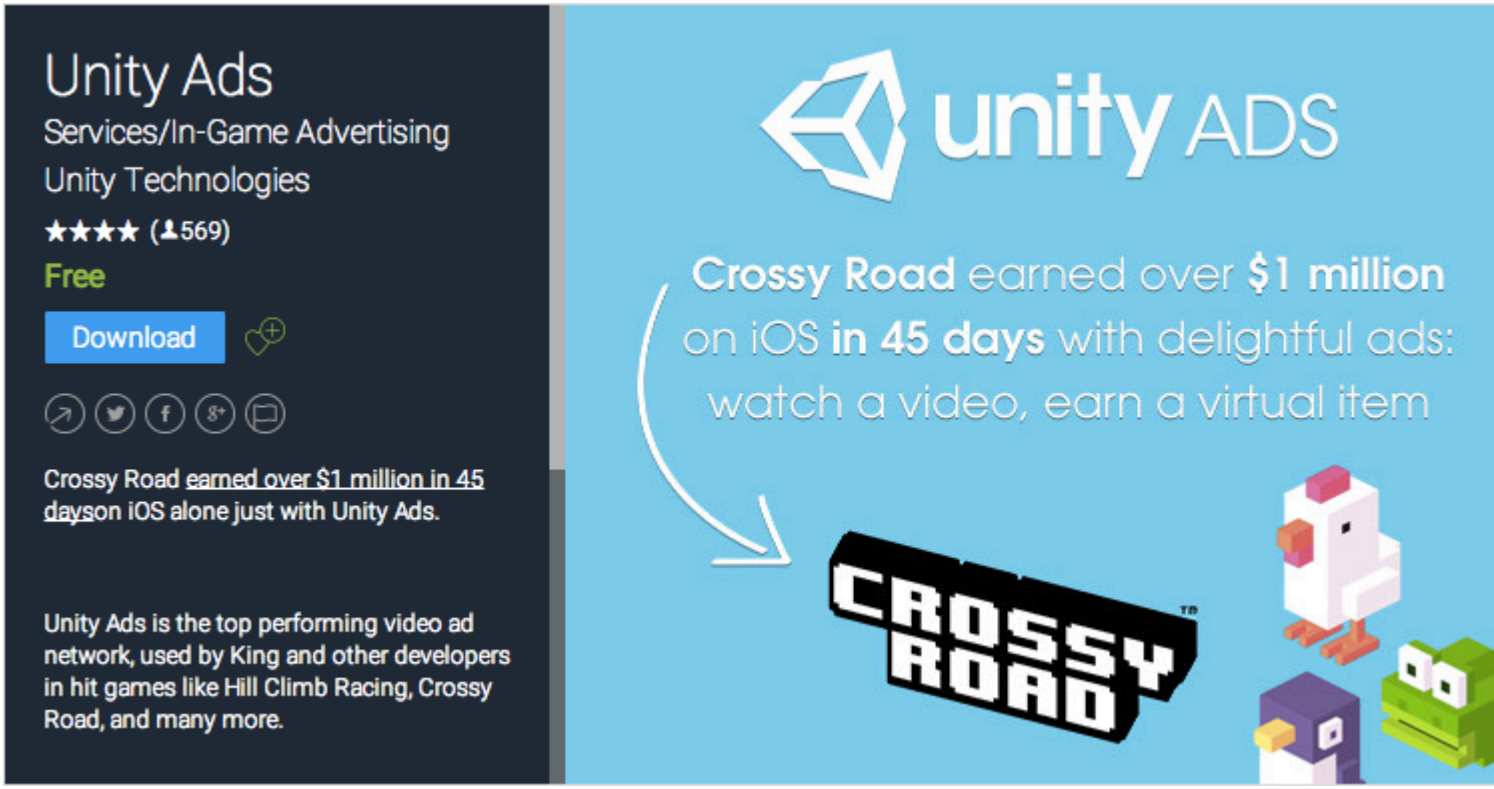
Hay tres formas de acceder a la Tienda de Activos de Unity:

- Abra la ventana de Asset Store seleccionando Ventana → Asset Store en el menú principal dentro de Unity.
- Utilice la tecla de acceso directo (Ctrl + 9 en Windows / 9 en Mac OS)
- Navegue por la interfaz web: https://www.assetstore.unity3d.com/

Es posible que se le solicite que cree una cuenta de usuario gratuita o que inicie sesión si es la primera vez que accede a Unity Asset Store.

Compra de activos

Después de acceder a la Tienda de recursos y ver el contenido que desea descargar, simplemente haga clic en el botón **Descargar** . El texto del botón también puede ser **Comprar ahora** si el activo tiene un costo asociado.



Si está viendo Unity Asset Store a través de la interfaz web, el texto del botón **Descargar** puede mostrarse como **Abrir en la unidad** . Al seleccionar este botón, se iniciará una instancia de Unity y se mostrará el activo dentro de la *ventana de la Tienda de Activos* .

Se le puede solicitar que cree una cuenta de usuario gratuita o que inicie sesión si es la primera

vez que compra en Unity Asset Store.

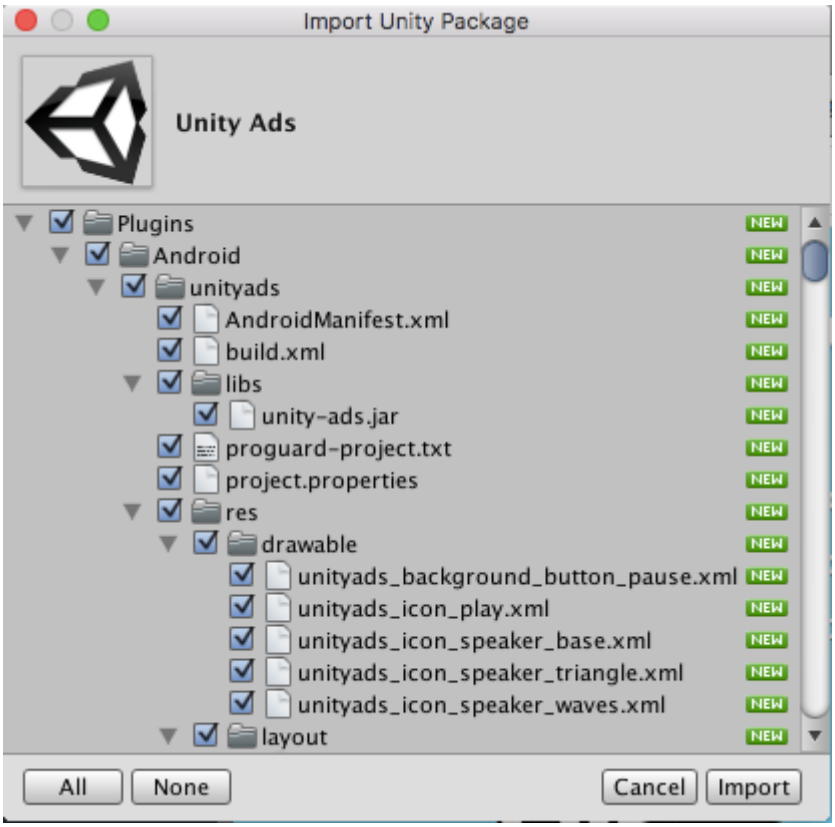
Luego, Unity procederá a aceptar su pago, si corresponde.

Importando activos

Una vez que el activo se haya descargado en Unity, el botón **Descargar** o **Comprar ahora** cambiará a **Importar** .

La selección de esta opción le indicará al usuario una ventana de *Importarity Package* , donde el usuario puede seleccionar los archivos de activos que desea importar dentro de su proyecto.

Seleccione **Importar** para confirmar el proceso, colocando los archivos de activos seleccionados dentro de la carpeta Activos que se muestra en la *ventana Vista* del *proyecto* .



Publicación de Activos

1. hacer una cuenta de editor
2. agregar un activo en la cuenta del editor
3. descargar las herramientas de la tienda de activos (desde la tienda de activos)
4. vaya a "Herramientas de Asset Store"> "Carga de paquetes"
5. seleccione el paquete correcto y la carpeta del proyecto en la ventana de herramientas de la tienda de activos
6. haga clic en subir
7. envíe su activo en línea

TODO - agregar imágenes, más detalles

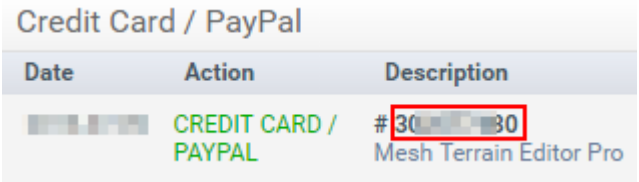
Confirmar el número de factura de una compra.

El número de factura se utiliza para verificar la venta para los editores. Muchos editores de activos pagados o complementos solicitan el número de factura a petición de soporte. El número de factura también se usa como una clave de licencia para activar algún activo o complemento.

El número de factura se puede encontrar en dos lugares:

1. Después de comprar el activo, se le enviará un correo electrónico cuyo asunto es "Confirmación de compra de Unity Asset Store ...". El número de factura se encuentra en el archivo PDF adjunto de este correo electrónico.

2. Abra https://www.assetstore.unity3d.com/#!/account/transactions , luego puede encontrar el número de factura en la columna *Descripción* .



Lea Tienda de activos en línea: https://riptutorial.com/es/unity3d/topic/5705/tienda-de-activos

Capítulo 37: Transforma

Sintaxis

- void Transform.Translate (vector3 traducción, espacio relativeTo = Space.Self)
- void Transform.Translate (float x, float y, float z, Space relativeTo = Space.Self)
- void Transform.Rotate (Vector3 eulerAngles, Space relativeTo = Space.Self)
- void Transform.Rotate (float xAngle, float yAngle, float zAngle, Space relativeTo = Space.Self)
- void Transform.Rotate (Vector3 axis, float angle, Space relativeTo = Space.Self)
- void Transform.RotateAround (punto Vector3, eje Vector3, ángulo de flotación)
- void Transform.LookAt (Transformar objetivo, Vector3 worldUp = Vector3.up)
- void Transform.LookAt (Vector3 worldPosition, Vector3 worldUp = Vector3.up)

Examples

Visión general

Las transformaciones contienen la mayoría de los datos sobre un objeto en unidad, incluidos los padres, los hijos, la posición, la rotación y la escala. También tiene funciones para modificar cada una de estas propiedades. Cada GameObject tiene una transformación.

Traducir (mover) un objeto

```
// Move an object 10 units in the positive x direction
transform.Translate(10, 0, 0);

// translating with a vector3
vector3 distanceToMove = new Vector3(5, 2, 0);
transform.Translate(distanceToMove);
```

Rotando un objeto

```
// Rotate an object 45 degrees about the Y axis
transform.Rotate(0, 45, 0);

// Rotates an object about the axis passing through point (in world coordinates) by angle in
degrees
transform.RotateAround(point, axis, angle);
// Rotates on it's place, on the Y axis, with 90 degrees per second
transform.RotateAround(Vector3.zero, Vector3.up, 90 * Time.deltaTime);

// Rotates an object to make it's forward vector point towards the other object
transform.LookAt(otherTransform);
// Rotates an object to make it's forward vector point towards the given position (in world
coordinates)
transform.LookAt(new Vector3(10, 5, 0));
```

Más información y ejemplos se pueden ver en la documentación de Unity .

También tenga en cuenta que si el juego utiliza cuerpos rígidos, entonces la transformación no se debe interactuar directamente (a menos que el cuerpo rígido sea `isKinematic == true`). En ese caso, use AddForce u otros métodos similares para actuar directamente sobre el cuerpo rígido.

Padres e hijos

Unity trabaja con jerarquías para mantener su proyecto organizado. Puede asignar objetos a un lugar en la jerarquía utilizando el editor, pero también puede hacerlo a través del código.

Crianza de los hijos

Puedes establecer el padre de un objeto con los siguientes métodos

```
var other = GetOtherGameObject();
other.transform.SetParent( transform );
other.transform.SetParent( transform, worldPositionStays );
```

Cada vez que establezca un elemento primario de transformación, mantendrá la posición de los objetos como una posición mundial. Puede elegir que esta posición sea relativa al pasar *falso* para el parámetro *worldPositionStays* .

También puede verificar si el objeto es un hijo de otra transformación con el siguiente método

```
other.transform.IsChildOf( transform );
```

Conseguir un niño

Como los objetos se pueden parear entre sí, también puede encontrar hijos en la jerarquía. La forma más sencilla de hacerlo es mediante el siguiente método

```
transform.Find( "other" );
transform.FindChild( "other" );
```

Nota: FindChild llama a Find under the hood

También puede buscar niños más abajo en la jerarquía. Para ello, agregue una "/" para especificar un nivel más profundo.

```
transform.Find( "other/another" );
transform.FindChild( "other/another" );
```

Otra forma de obtener un hijo es usando GetChild

```
transform.GetChild( index );
```

> GetChild requiere un número entero como índice que debe ser menor que el recuento total de hijos

```
int count = transform.childCount;
```

Cambio de índice de hermanos

Puedes cambiar el orden de los hijos de un GameObject. Puede hacer esto para definir el orden de sorteo de los hijos (suponiendo que estén en el mismo nivel Z y el mismo orden de clasificación).

```
other.transform.SetSiblingIndex( index );
```

También puede establecer rápidamente el índice de hermanos primero o último usando los siguientes métodos

```
other.transform.SetAsFirstSibling();
other.transform.SetAsLastSibling();
```

Separando a todos los niños

Si desea liberar a todos los hijos de una transformación, puede hacer esto:

```
foreach(Transform child in transform)
{
    child.parent = null;
}
```

Además, Unity proporciona un método para este propósito:

```
transform.DetachChildren();
```

Básicamente, tanto looping como `DetachChildren()` configuran a los padres de los niños de primer nivel en nulo, lo que significa que no tendrán padres.

(Hijos de primera profundidad: las transformaciones que son directamente hijos de la transformación)

Lea Transforma en línea: https://riptutorial.com/es/unity3d/topic/2190/transforma

Capítulo 38: Unity Profiler

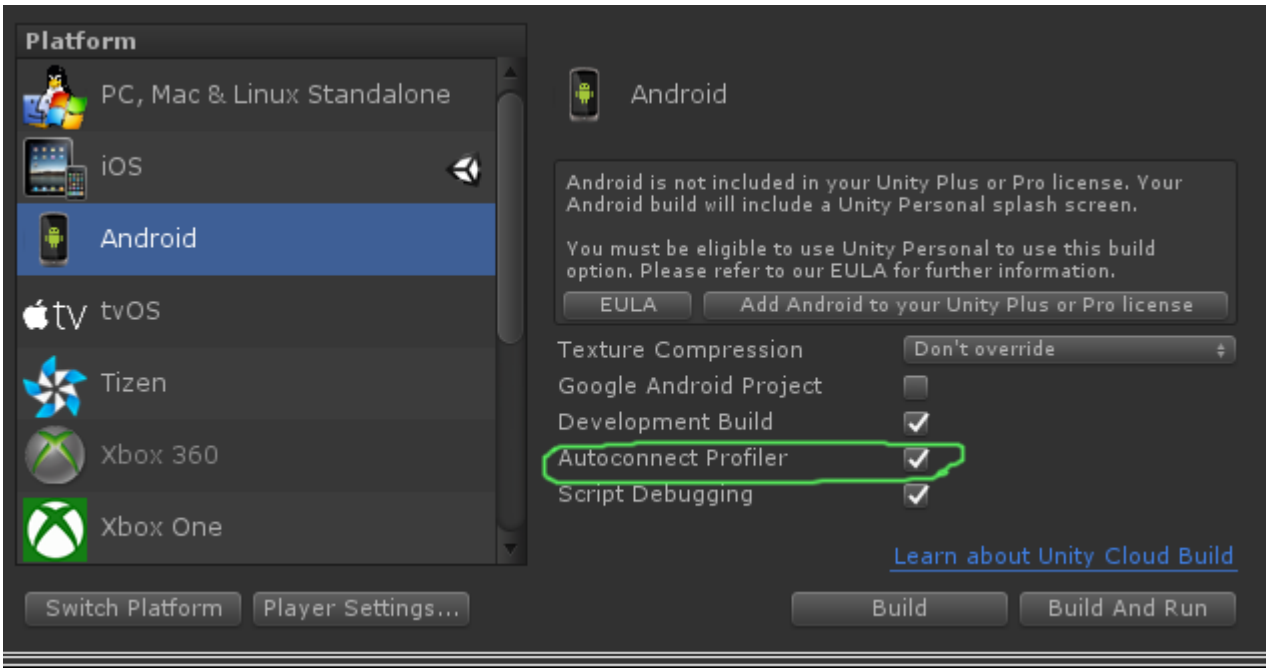
Observaciones

Usando Profiler en diferentes dispositivos

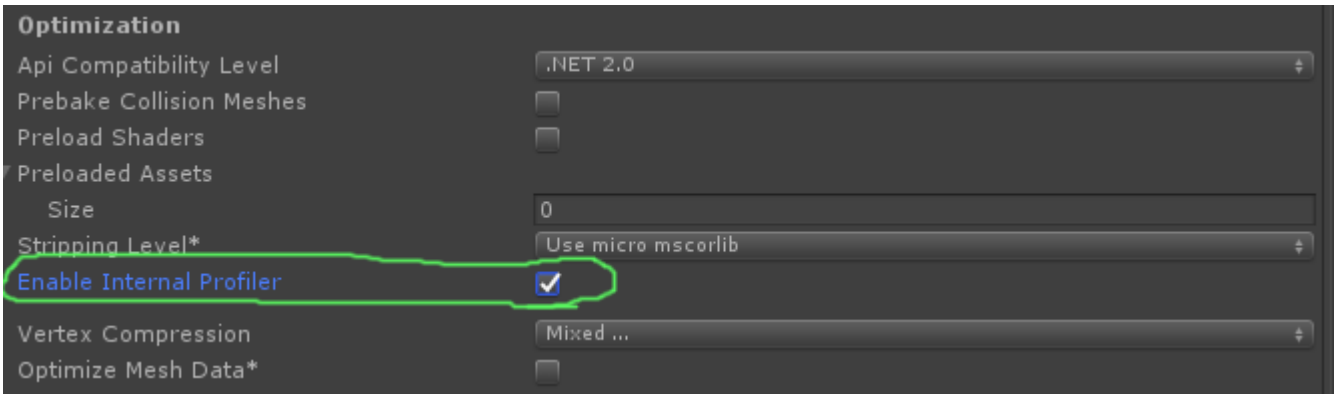
Hay algunas cosas importantes que se deben saber para conectar correctamente el Perfilador en diferentes plataformas.

Androide

Para adjuntar correctamente el perfil, se debe usar el botón "Crear y ejecutar" de la ventana Crear configuración con la opción **Autoconectar** perfil.



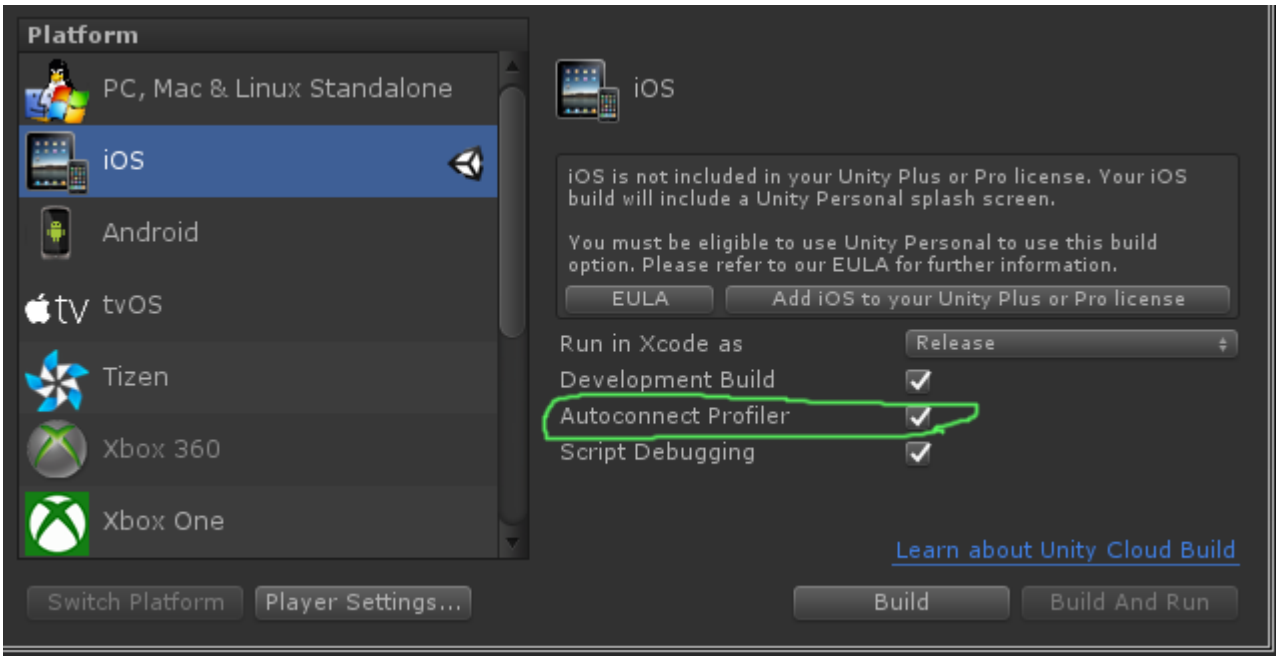
Otra opción obligatoria, en el inspector de configuración del reproductor de Android en la pestaña Otras configuraciones, hay una casilla de verificación **Habilitar el generador de perfiles interno** que debe verificarse para que LogCat genere la información del generador de perfiles.



Usar solo "Generar" no permitirá que el generador de perfiles se conecte a un dispositivo Android porque "Construir y Ejecutar" usa argumentos de línea de comando específicos para iniciarlo con LogCat.

iOS

Para adjuntar correctamente el perfil, el botón "Crear y ejecutar" de la ventana Crear configuración con la opción **Autoconectar** perfil seleccionado debe usarse en la primera ejecución.



En iOS, no hay ninguna opción en la configuración del reproductor que deba configurarse para que el Perfilador se habilite. Debería funcionar fuera de la caja.

Examples

Marcador de perfiles

Usando la Clase Profiler

Una muy buena práctica es usar Profiler.BeginSample y Profiler.EndSample porque tendrá su propia entrada en la ventana de Profiler.

Además, esas etiquetas se eliminarán en una compilación que no sea de desarrollo utilizando ConditionalAttribute, por lo que no es necesario que las elimines de tu código.

```
public class SomeClass : MonoBehaviour
{
    void SomeFunction()
    {
        Profiler.BeginSample("SomeClass.SomeFunction");
        // Various call made here
```

```
        Profiler.EndSample();
    }
}
```

Esto creará una entrada "SomeClass.SomeFunction" en la ventana del generador de perfiles que permitirá la depuración y la identificación más fáciles del cuello de la botella.

Lea Unity Profiler en línea: https://riptutorial.com/es/unity3d/topic/6974/unity-profiler

Capítulo 39: Usando el control de fuente Git con Unity

Examples

Usando Git Large File Storage (LFS) con Unity

Prefacio

Git puede trabajar con el desarrollo de videojuegos fuera de la caja. Sin embargo, la advertencia principal es que las versiones de archivos de medios grandes (> 5 MB) pueden ser un problema a largo plazo, ya que su historial de confirmaciones aumenta - Git simplemente no se creó originalmente para la versión de archivos binarios.

La gran noticia es que desde mediados de 2015, GitHub ha lanzado un complemento para Git llamado Git LFS que trata este problema directamente. ¡Ahora puedes fácilmente y eficientemente versionar archivos binarios grandes!

Finalmente, esta documentación se centra en los requisitos específicos y la información necesaria para garantizar que su vida Git funcione bien con el desarrollo de videojuegos. Esta guía no cubrirá cómo usar Git en sí.

Instalación de Git y Git-LFS

Como desarrollador, tienes una serie de opciones disponibles y la primera opción es si instalar la línea de comandos central de Git o dejar que una de las populares aplicaciones de la GUI de Git se encargue de eso por ti.

Opción 1: usar una aplicación Git GUI

Esta es realmente una preferencia personal aquí ya que hay bastantes opciones en términos de la GUI de Git o si usar una GUI en absoluto. Tiene una serie de aplicaciones para elegir, aquí hay 3 de las más populares:

- Sourcetree (Gratis)
- Escritorio Github (Gratis)
- SmartGit (Commerical)

Una vez que haya instalado la aplicación de su elección, busque en Google y siga las instrucciones para asegurarse de que esté configurada para Git-LFS. Vamos a omitir este paso en esta guía, ya que es específico de la aplicación.

Opción 2: instalar Git y Git-LFS

Esto es bastante simple: instala Git . Entonces. Instalar Git LFS .

Configurando Git Large File Storage en tu proyecto

Si está utilizando el complemento Git LFS para brindar un mejor soporte para archivos binarios, entonces necesitará configurar algunos tipos de archivos para que sean administrados por Git LFS. Agregue lo siguiente a su archivo `.gitattributes` en la raíz de su repositorio para admitir archivos binarios comunes utilizados en proyectos de Unity:

```
# Image formats:
*.tga filter=lfs diff=lfs merge=lfs -text
*.png filter=lfs diff=lfs merge=lfs -text
*.tif filter=lfs diff=lfs merge=lfs -text
*.jpg filter=lfs diff=lfs merge=lfs -text
*.gif filter=lfs diff=lfs merge=lfs -text
*.psd filter=lfs diff=lfs merge=lfs -text

# Audio formats:
*.mp3 filter=lfs diff=lfs merge=lfs -text
*.wav filter=lfs diff=lfs merge=lfs -text
*.aiff filter=lfs diff=lfs merge=lfs -text

# 3D model formats:
*.fbx filter=lfs diff=lfs merge=lfs -text
*.obj filter=lfs diff=lfs merge=lfs -text

# Unity formats:
*.sbsar filter=lfs diff=lfs merge=lfs -text
*.unity filter=lfs diff=lfs merge=lfs -text

# Other binary formats
*.dll filter=lfs diff=lfs merge=lfs -text
```

Configuración de un repositorio Git para Unity

Al inicializar un repositorio Git para el desarrollo de Unity, hay un par de cosas que deben hacerse.

Unidad ignorar carpetas

No todo debería estar versionado en el repositorio. Puede agregar la plantilla a continuación a su archivo `.gitignore` en la raíz de su repositorio. O alternativamente, puede verificar el código abierto Unity .gitignore en GitHub y alternativamente generar uno usando gitignore.io para la unidad.

```
# Unity Generated
[Tt]emp/
[Ll]ibrary/
[Oo]bj/

# Unity3D Generated File On Crash Reports
sysinfo.txt

# Visual Studio / MonoDevelop Generated
ExportedObj/
obj/
*.csproj
*.unityproj
*.sln
*.suo
*.tmp
*.user
*.userprefs
*.pidb
*.booproj
*.svd

# OS Generated
desktop.ini
.DS_Store
.DS_Store?
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db
```

Para obtener más información sobre cómo configurar un archivo .gitignore, consulte aquí .

Configuraciones del Proyecto Unity

Por defecto, los proyectos de Unity no están configurados para soportar las versiones correctamente.

1. (Omita este paso en v4.5 y versiones posteriores) Habilite la opción `External` en `Unity → Preferences → Packages → Repository` .
2. Cambie a `Visible Meta Files` en `Edit → Project Settings → Editor → Version Control Mode` .
3. Cambie a `Force Text` en `Edit → Project Settings → Editor → Asset Serialization Mode` .
4. Guarda la escena y el proyecto desde el menú `File` .

Configuración adicional

Una de las pocas molestias importantes que uno tiene al usar Git con los proyectos de Unity es que a Git no le importan los directorios y con mucho gusto dejará los directorios vacíos alrededor después de eliminar los archivos de ellos. Unity `*.meta` archivos `*.meta` para estos directorios y puede provocar un poco de batalla entre los miembros del equipo cuando Git se comprometa a seguir agregando y eliminando estos meta archivos.

Agregue este `/.git/hooks/` Git post-merge a la carpeta `/.git/hooks/` para repositorios con proyectos Unity en ellos. Después de cualquier extracción / combinación de Git, verá qué archivos se han eliminado, verificará si el directorio en el que existía está vacío y, si es así, eliminarlo.

Combinación de escenas y prefabricados

Un problema común cuando se trabaja con Unity es cuando 2 o más desarrolladores están modificando una escena o prefab de Unity (archivos * .unity). Git no sabe cómo combinarlos correctamente fuera de la caja. Afortunadamente, el equipo de Unity implementó una herramienta llamada SmartMerge que hace que la fusión simple sea automática. Lo primero que debe hacer es agregar las siguientes líneas a su archivo `.git` o `.gitconfig` : (Windows: `%USERPROFILE%\.gitconfig` , Linux / Mac OS X: `~/.gitconfig`)

```
[merge]
tool = unityyamlmerge

[mergetool "unityyamlmerge"]
trustExitCode = false
cmd = '<path to UnityYAMLMerge>' merge -p "$BASE" "$REMOTE" "$LOCAL" "$MERGED"
```

En **Windows** la ruta a UnityYAMLMerge es:

```
C:\Program Files\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

o

```
C:\Program Files (x86)\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

y en **MacOSX** :

```
/Applications/Unity/Unity.app/Contents/Tools/UnityYAMLMerge
```

Una vez hecho esto, el mergetool estará disponible cuando surjan conflictos durante la fusión / rebase. No te olvides de ejecutar `git mergetool` manualmente para activar UnityYAMLMerge.

Lea Usando el control de fuente Git con Unity en línea:
https://riptutorial.com/es/unity3d/topic/2195/usando-el-control-de-fuente-git-con-unity

Capítulo 40: Vector3

Introducción

La estructura `Vector3` representa una coordenada 3D, y es una de las estructuras de la `UnityEngine` troncal de la biblioteca `UnityEngine` . La estructura `Vector3` se encuentra más comúnmente en el componente `Transform` de la mayoría de los objetos del juego, donde se usa para mantener la *posición* y la *escala* . `Vector3` proporciona una buena funcionalidad para realizar operaciones vectoriales comunes. Puedes leer más sobre la estructura `Vector3` en la API de Unity.

Sintaxis

- Vector3 público ();
- Vector3 público (float x, float y);
- Vector3 público (float x, float y, float z);
- Vector3.Lerp (Vector3 startPosition, Vector3 targetPosition, float movementFraction);
- Vector3.LerpUnclamped (Vector3 startPosition, Vector3 targetPosition, float movementFraction);
- Vector3.MoveTowards (Vector3 startPosition, Vector3 targetPosition, distancia flotante);

Examples

Valores estáticos

La estructura `Vector3` contiene algunas variables estáticas que proporcionan valores `Vector3` comúnmente utilizados. La mayoría representa una *dirección* , pero aún se pueden usar creativamente para proporcionar funcionalidad adicional.

`Vector3.zero` y `Vector3.one`

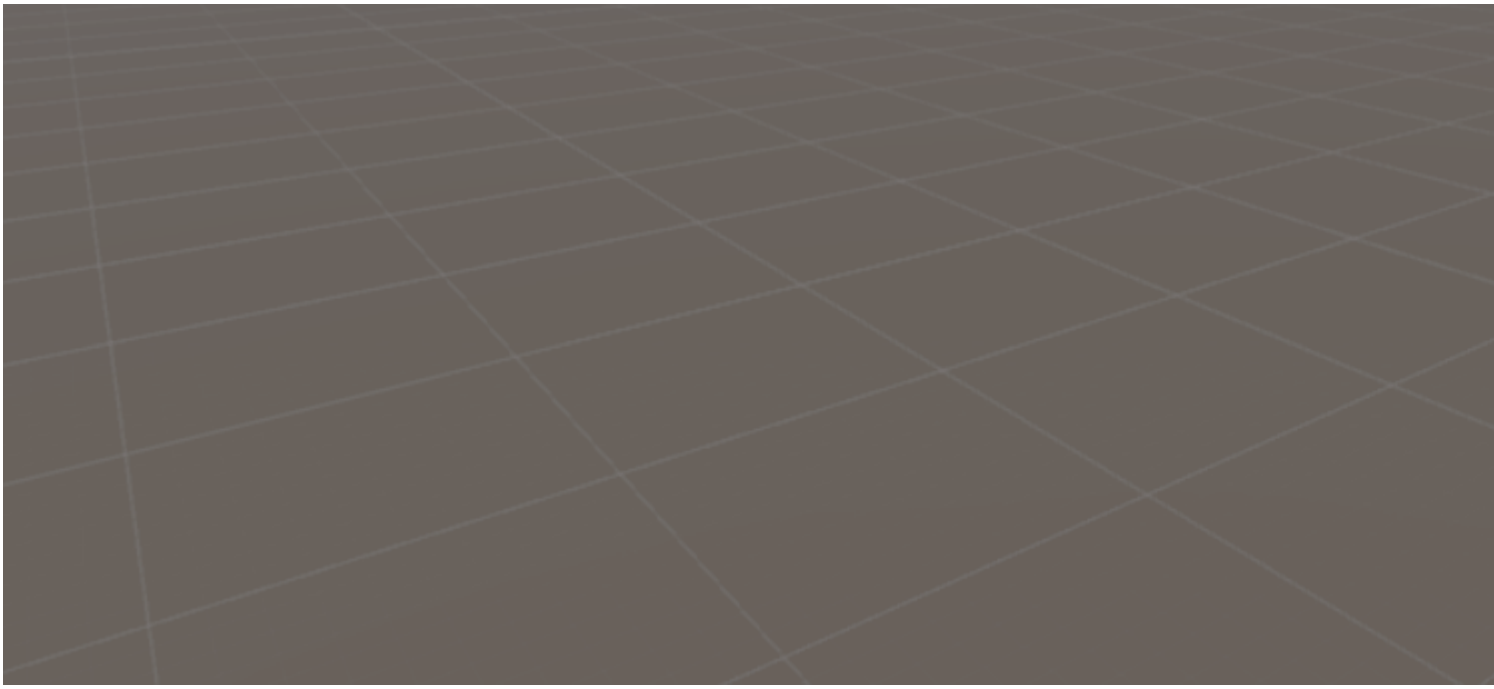
`Vector3.zero` y `Vector3.one` se usan típicamente en conexión con un `Vector3` *normalizado* ; es decir, un `Vector3` donde los valores `x` , `y` y `z` tienen una magnitud de 1. Como tal, `Vector3.zero` representa el valor más bajo, mientras que `Vector3.one` representa el valor más grande.

`Vector3.zero` también se usa comúnmente para establecer la posición predeterminada en las transformaciones de objetos.

La siguiente clase usa `Vector3.zero` y `Vector3.one` para inflar y desinflar una esfera.

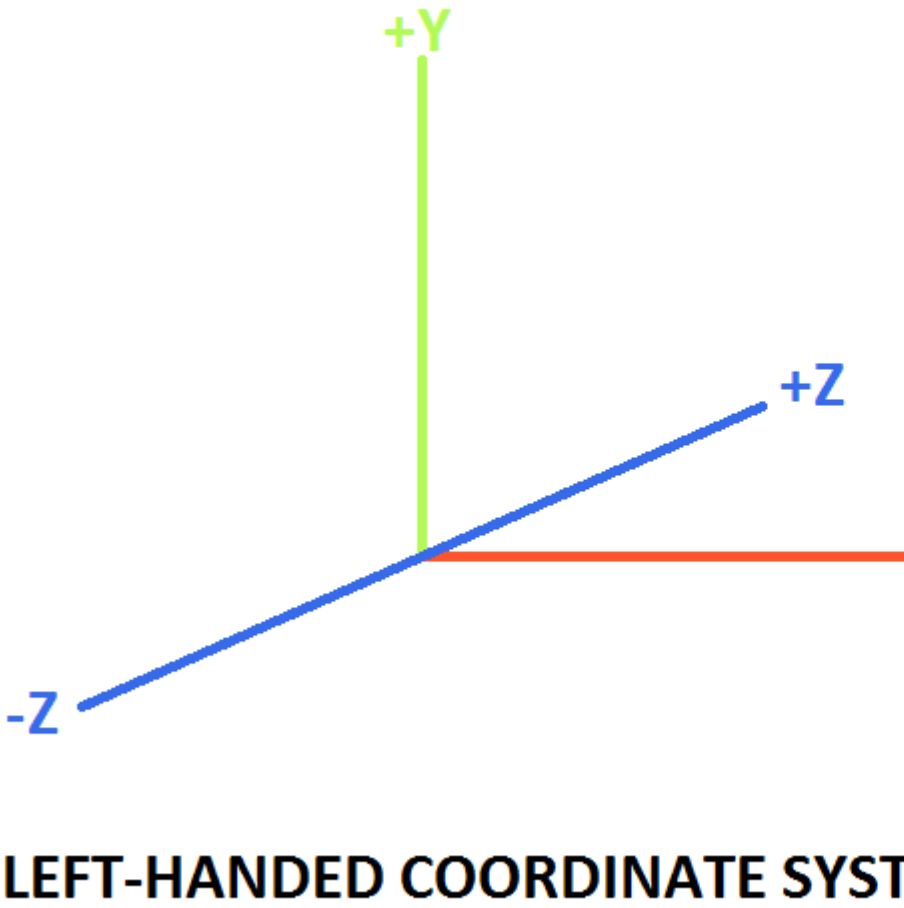
```
using UnityEngine;
```

```
public class Inflater : MonoBehaviour
{
    <summary>A sphere set up to inflate and deflate between two values.</summary>
    public ScaleBetween sphere;

    ///<summary>On start, set the sphere GameObject up to inflate
    /// and deflate to the corresponding values.</summary>
    void Start()
    {
        // Vector3.zero = Vector3(0, 0, 0); Vector3.one = Vector3(1, 1, 1);
        sphere.SetScale(Vector3.zero, Vector3.one);
    }
}
```



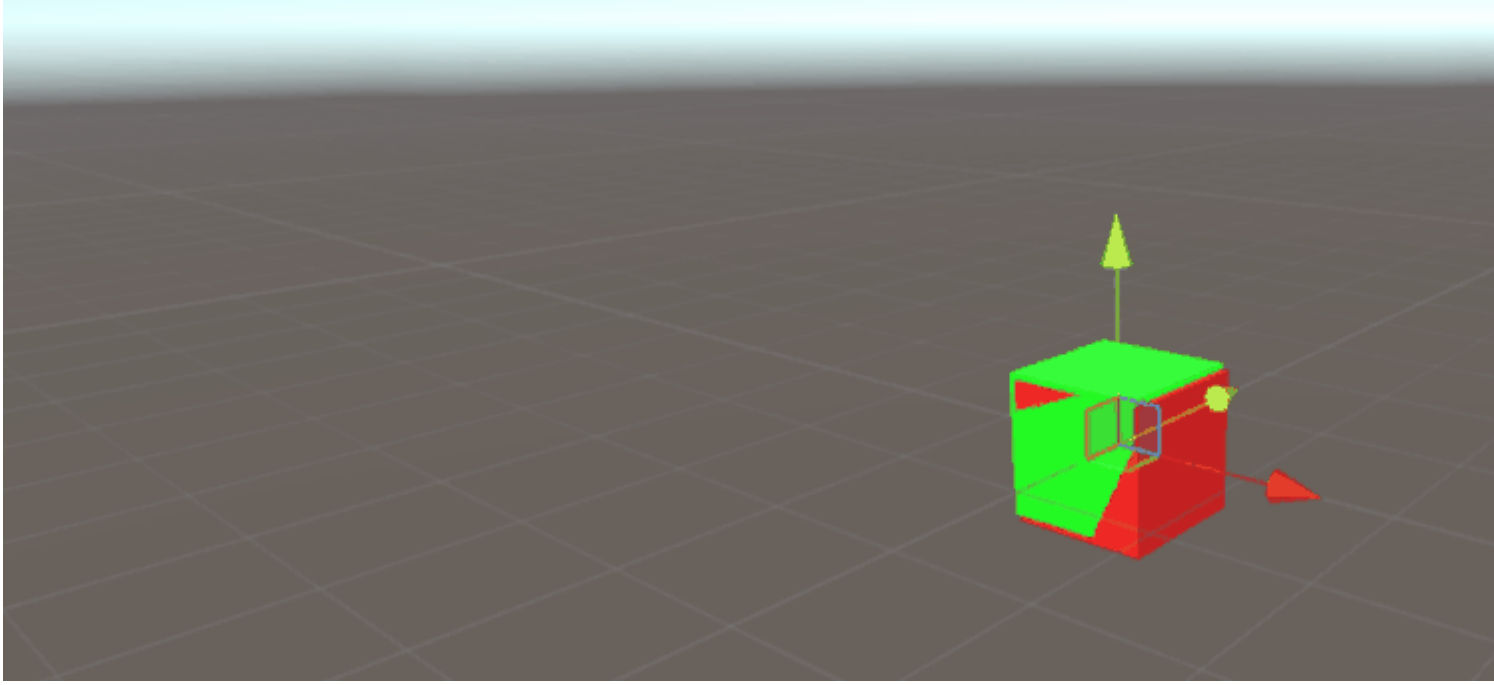
Direcciones estáticas

Las direcciones estáticas pueden ser útiles en varias aplicaciones, con dirección a lo largo de los tres ejes positivo y negativo. Es importante tener en cuenta que Unity emplea un sistema de coordenadas para zurdos, que tiene un efecto en la dirección.



La siguiente clase utiliza las direcciones estáticas `Vector3` para mover objetos a lo largo de los tres ejes.

```
using UnityEngine;

public class StaticMover : MonoBehaviour
{
    <summary>GameObjects set up to move back and forth between two directions.</summary>
    public MoveBetween xMovement, yMovement, zMovement;

    ///<summary>On start, set each MoveBetween GameObject up to move
    /// in the corresponding direction(s).</summary>
    void Start()
    {
        // Vector3.left = Vector3(-1, 0, 0); Vector3.right = Vector3(1, 0, 0);
        xMovement.SetDirections(Vector3.left, Vector3.right);

        // Vector3.down = Vector3(0, -1, 0); Vector3.up = Vector3(0, 0, 1);
        yMovement.SetDirections(Vector3.down, Vector3.up);

        // Vector3.back = Vector3(0, 0, -1); Vector3.forward = Vector3(0, 0, 1);
        zMovement.SetDirections(Vector3.back, Vector3.forward);
    }
}
```



Índice

Valor	X	y	z	`new Vector3()` método equivalente `new Vector3()`
`Vector3.zero`	0	0	0	`new Vector3(0, 0, 0)`
`Vector3.one`	1	1	1	`new Vector3(1, 1, 1)`
`Vector3.left`	-1	0	0	`new Vector3(-1, 0, 0)`
`Vector3.right`	1	0	0	`new Vector3(1, 0, 0)`
`Vector3.down`	0	-1	0	`new Vector3(0, -1, 0)`
`Vector3.up`	0	1	0	`new Vector3(0, 1, 0)`
`Vector3.back`	0	0	-1	`new Vector3(0, 0, -1)`
`Vector3.forward`	0	0	1	`new Vector3(0, 0, 1)`

Creando un Vector3

Una estructura `Vector3` se puede crear de varias maneras. `Vector3` es una estructura, y como tal, normalmente deberá ser instanciada antes de su uso.

Constructores

Hay tres constructores integrados para instanciar un `Vector3` .

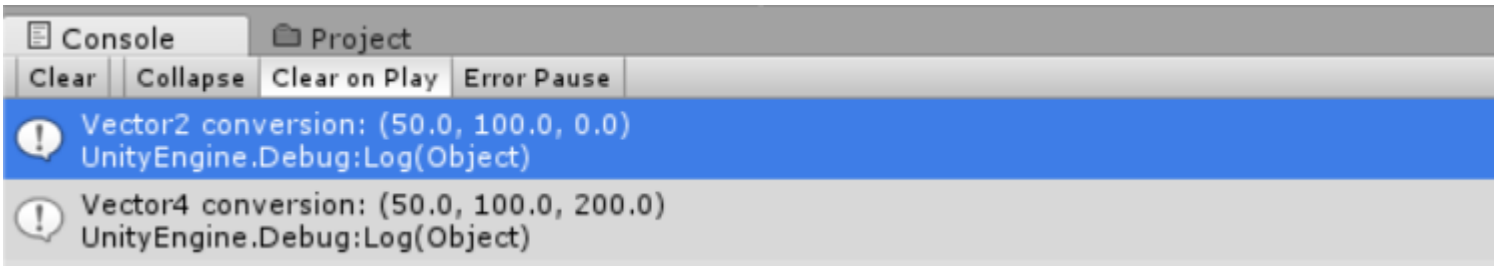
Constructor	Resultado
`new Vector3()`	Crea una estructura `Vector3` con coordenadas de (0, 0, 0).
`new Vector3(float x, float y)`	Crea una estructura `Vector3` con las coordenadas `x` e `y` dadas. `z` se establecerá en 0.
`new Vector3(float x, float y, float z)`	Crea una estructura `Vector3` con las coordenadas `x` , `y` y `z` dadas.

Convertir desde un `Vector2` o `Vector4`

Si bien es poco frecuente, puede encontrarse con situaciones en las que tendría que tratar las coordenadas de una estructura `Vector2` o `Vector4` como un `Vector3` . En tales casos, simplemente puede pasar el `Vector2` o `Vector4` directamente al `Vector3` , sin instanciarlo previamente. Como debe suponerse, una estructura `Vector2` solo pasará los valores `x` e `y` , mientras que una clase `Vector4` omitirá su `w` .

Podemos ver la conversión directa en el siguiente script.

```
void VectorConversionTest()
{
    Vector2 vector2 = new Vector2(50, 100);
    Vector4 vector4 = new Vector4(50, 100, 200, 400);

    Vector3 fromVector2 = vector2;
    Vector3 fromVector4 = vector4;

    Debug.Log("Vector2 conversion: " + fromVector2);
    Debug.Log("Vector4 conversion: " + fromVector4);
}
```



Aplicando movimiento

La estructura `Vector3` contiene algunas funciones estáticas que pueden proporcionar utilidad cuando deseamos aplicar movimiento al `Vector3` .

`Lerp` y `LerpUnclamped`

Las funciones lerp proporcionan movimiento entre dos coordenadas basadas en una fracción proporcionada. Donde `Lerp` solo permitirá el movimiento entre las dos coordenadas, `LerpUnclamped` permite las fracciones que se mueven fuera de los límites entre las dos coordenadas.

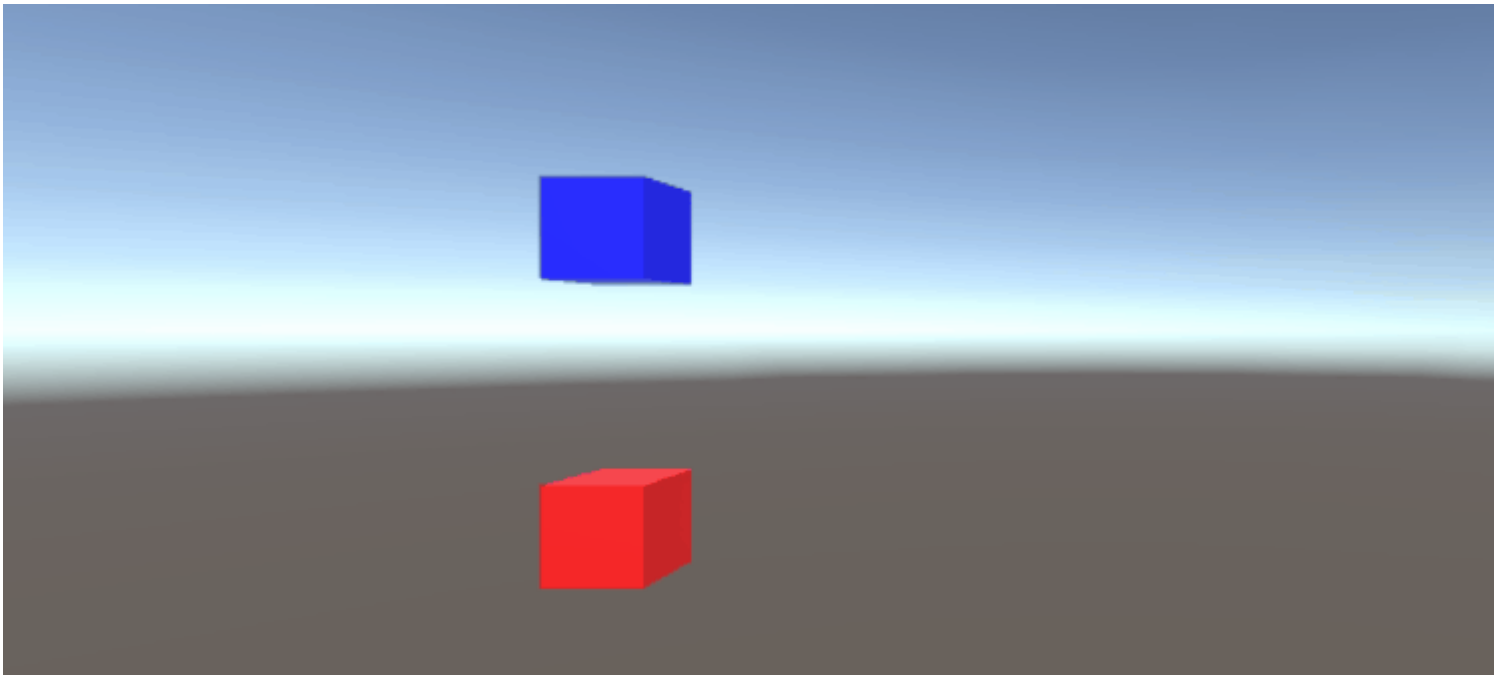
Proporcionamos la fracción de movimiento como un `float` . Con un valor de `0.5` , encontramos el punto medio entre las dos `Vector3` . Un valor de `0` o `1` devolverá el primer o segundo `Vector3` , respectivamente, ya que estos valores se correlacionan con ningún movimiento (devolviendo así el primer `Vector3`), o con un movimiento completado (devolviendo el segundo `Vector3`). Es importante tener en cuenta que ninguna de las funciones se adaptará al cambio en la fracción de movimiento. Esto es algo que debemos tener en cuenta manualmente.

Con `Lerp` , todos los valores se fijan entre `0` y `1` . Esto es útil cuando queremos proporcionar movimiento hacia una dirección y no queremos sobrepasar el destino. `LerpUnclamped` puede tomar cualquier valor y puede usarse para proporcionar movimiento *lejos* del destino o más *allá* del destino.

La siguiente secuencia de comandos usa `Lerp` y `LerpUnclamped` para mover un objeto a un ritmo constante.

```
using UnityEngine;

public class Lerping : MonoBehaviour
{
    /// <summary>The red box will use Lerp to move. We will link
    /// this object in via the inspector.</summary>
    public GameObject lerpObject;
    /// <summary>The starting position for our red box.</summary>
    public Vector3 lerpStart = new Vector3(0, 0, 0);
    /// <summary>The end position for our red box.</summary>
    public Vector3 lerpTarget = new Vector3(5, 0, 0);

    /// <summary>The blue box will use LerpUnclamped to move. We will
    /// link this object in via the inspector.</summary>
    public GameObject lerpUnclampedObject;
    /// <summary>The starting position for our blue box.</summary>
    public Vector3 lerpUnclampedStart = new Vector3(0, 3, 0);
    /// <summary>The end position for our blue box.</summary>
    public Vector3 lerpUnclampedTarget = new Vector3(5, 3, 0);

    /// <summary>The current fraction to increment our lerp functions by.</summary>
    public float lerpFraction = 0;

    private void Update()
    {
        // First, I increment the lerp fraction.
        // delaTime * 0.25 should give me a value of +1 every second.
        lerpFraction += (Time.deltaTime * 0.25f);

        // Next, we apply the new lerp values to the target transform position.
        lerpObject.transform.position
            = Vector3.Lerp(lerpStart, lerpTarget, lerpFraction);
```

```
        lerpUnclampedObject.transform.position
            = Vector3.LerpUnclamped(lerpUnclampedStart, lerpUnclampedTarget, lerpFraction);
    }
}
```



MoveTowards

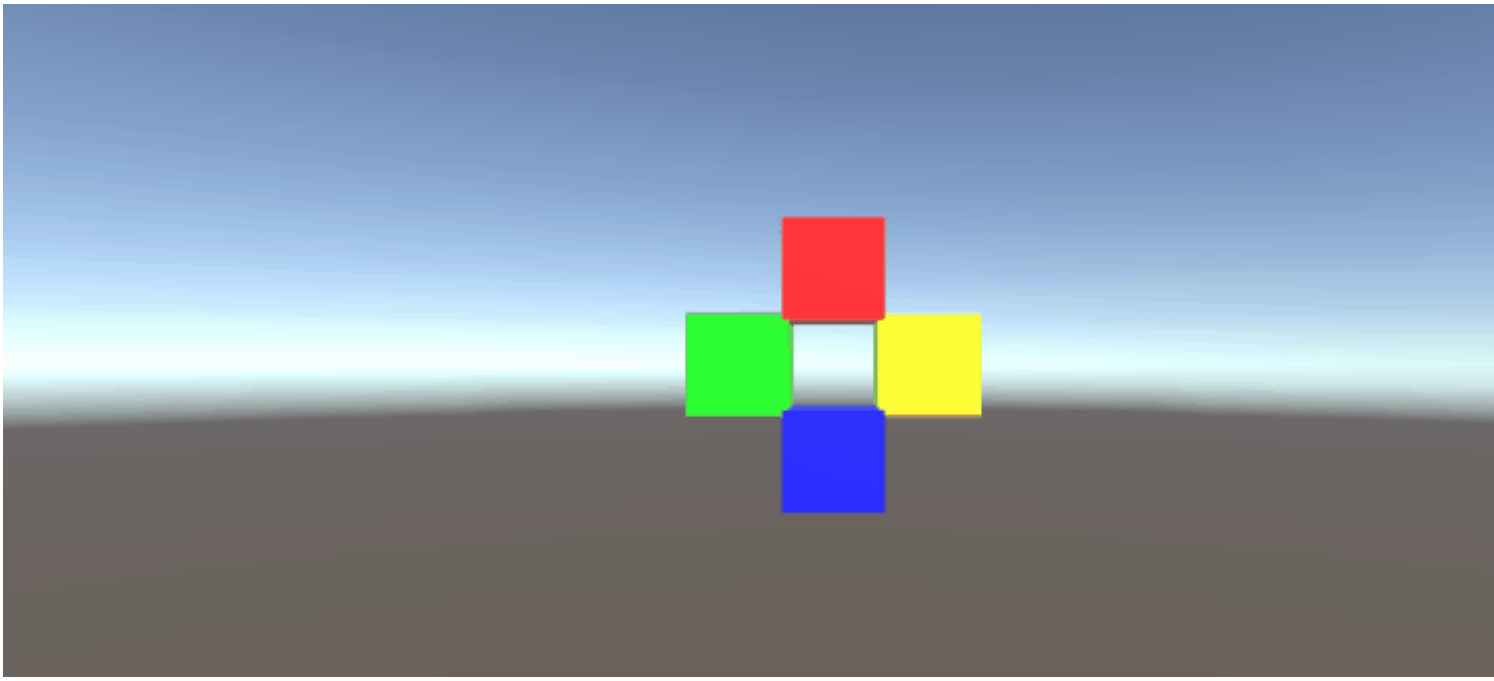
`MoveTowards` comporta *muy similar* a `Lerp` ; la diferencia principal es que proporcionamos una *distancia* real para moverse, en lugar de una *fracción* entre dos puntos. Es importante tener en cuenta que `MoveTowards` no se extenderá más allá del `Vector3` objetivo.

Al igual que con `LerpUnclamped` , podemos proporcionar un valor de distancia *negativo* para *alejarnos* del `Vector3` objetivo. En tales casos, nunca pasamos del `Vector3` objetivo y, por lo tanto, el movimiento es indefinido. En estos casos, podemos tratar el `Vector3` objetivo como una "dirección opuesta"; siempre que el `Vector3` apunte en la misma dirección, en relación con el `Vector3` inicio, el movimiento negativo debería comportarse como normal.

La siguiente secuencia de comandos utiliza `MoveTowards` para mover un grupo de objetos hacia un conjunto de posiciones utilizando una distancia suavizada.

```
using UnityEngine;

public class MoveTowardsExample : MonoBehaviour
{
    /// <summary>The red cube will move up, the blue cube will move down,
    /// the green cube will move left and the yellow cube will move right.
    /// These objects will be linked via the inspector.</summary>
    public GameObject upCube, downCube, leftCube, rightCube;
    /// <summary>The cubes should move at 1 unit per second.</summary>
    float speed = 1f;
```

```
    void Update()
    {
        // We determine our distance by applying a deltaTime scale to our speed.
        float distance = speed * Time.deltaTime;

        // The up cube will move upwards, until it reaches the
        //position of (Vector3.up * 2), or (0, 2, 0).
        upCube.transform.position
            = Vector3.MoveTowards(upCube.transform.position, (Vector3.up * 2f), distance);

        // The down cube will move downwards, as it enforces a negative distance..
        downCube.transform.position
            = Vector3.MoveTowards(downCube.transform.position, Vector3.up * 2f, -distance);

        // The right cube will move to the right, indefinetly, as it is constantly updating
        // its target position with a direction based off the current position.
        rightCube.transform.position = Vector3.MoveTowards(rightCube.transform.position,
            rightCube.transform.position + Vector3.right, distance);

        // The left cube does not need to account for updating its target position,
        // as it is moving away from the target position, and will never reach it.
        leftCube.transform.position
            = Vector3.MoveTowards(leftCube.transform.position, Vector3.right, -distance);
    }
}
```



SmoothDamp

Piense en `SmoothDamp` como una variante de `MoveTowards` con suavizado incorporado. Según la documentación oficial, esta función es la más utilizada para realizar un seguimiento suave de la cámara.

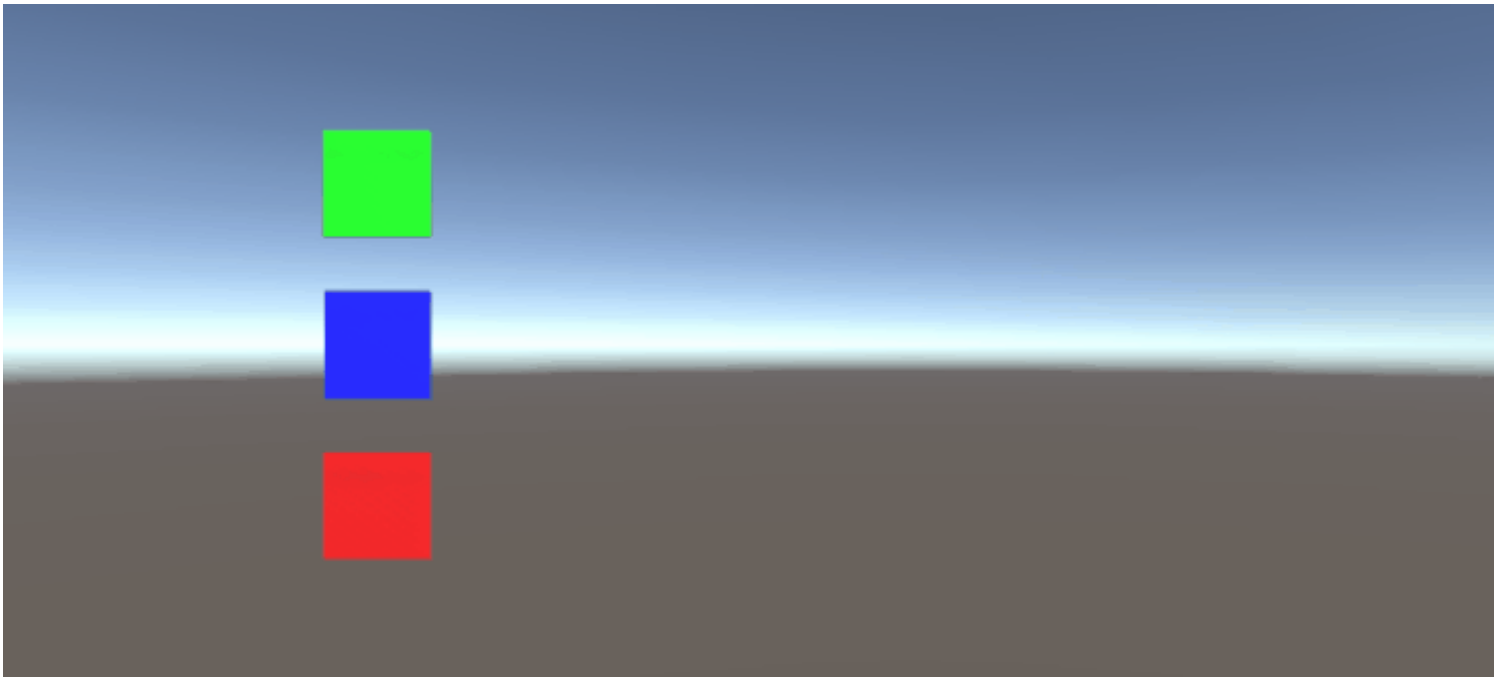
Junto con las coordenadas de inicio y objetivo de `Vector3` , también debemos proporcionar un `Vector3` para representar la velocidad y un `float` represente el tiempo *aproximado* que debe tomar

para completar el movimiento. A diferencia de los ejemplos anteriores, proporcionamos la velocidad como *referencia* , para ser incrementada, internamente. Es importante tomar nota de esto, ya que cambiar la velocidad fuera de la función mientras aún estamos realizando la función puede tener resultados no deseados.

Además de las variables *requeridas* , también podemos proporcionar un `float` para representar la velocidad máxima de nuestro objeto, y un `float` para representar el intervalo de tiempo desde la llamada de `SmoothDamp` anterior al objeto. No *necesitamos* proporcionar estos valores; de forma predeterminada, no habrá velocidad máxima, y el intervalo de tiempo se interpretará como `Time.deltaTime` . Más importante aún, si está llamando a la función de uno por objeto dentro de un `MonoBehaviour.Update()` la función, *no* debe necesitar para declarar un intervalo de tiempo.

```
using UnityEngine;

public class SmoothDampMovement : MonoBehaviour
{
    /// <summary>The red cube will imitate the default SmoothDamp function.
    /// The blue cube will move faster by manipulating the "time gap", while
    /// the green cube will have an enforced maximum speed. Note that these
    /// objects have been linked via the inspector.</summary>
    public GameObject smoothObject, fastSmoothObject, cappedSmoothObject;

    /// <summary>We must instantiate the velocities, externally, so they may
    /// be manipulated from within the function. Note that by making these
    /// vectors public, they will be automatically instantiated as Vector3.Zero
    /// through the inspector. This also allows us to view the velocities,
    /// from the inspector, to observe how they change.</summary>
    public Vector3 regularVelocity, fastVelocity, cappedVelocity;

    /// <summary>Each object should move 10 units along the X-axis.</summary>
    Vector3 regularTarget = new Vector3(10f, 0f);
    Vector3 fastTarget = new Vector3(10f, 1.5f);
    Vector3 cappedTarget = new Vector3(10f, 3f);

    /// <summary>We will give a target time of 5 seconds.</summary>
    float targetTime = 5f;

    void Update()
    {
        // The default SmoothDamp function will give us a general smooth movement.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime);

        // Note that a "maxSpeed" outside of reasonable limitations should not have any
        // effect, while providing a "deltaTime" of 0 tells the function that no time has
        // passed since the last SmoothDamp call, resulting in no movement, the second time.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime, 10f, 0f);

        // Note that "deltaTime" defaults to Time.deltaTime due to an assumption that this
        // function will be called once per update function. We can call the function
        // multiple times during an update function, but the function will assume that enough
        // time has passed to continue the same approximate movement. As a result,
        // this object should reach the target, quicker.
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
```

```
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);

        // Lastly, note that a "maxSpeed" becomes irrelevant, if the object does not
        // realistically reach such speeds. Linear speed can be determined as
        // (Distance / Time), but given the simple fact that we start and end slow, we can
        // infer that speed will actually be higher, during the middle. As such, we can
        // infer that a value of (Distance / Time) or (10/5) will affect the
        // function. We will half the "maxSpeed", again, to make it more noticeable.
        cappedSmoothObject.transform.position = Vector3.SmoothDamp(
            cappedSmoothObject.transform.position,
            cappedTarget, ref cappedVelocity, targetTime, 1f);
    }
}
```



Lea Vector3 en línea: https://riptutorial.com/es/unity3d/topic/7827/vector3

Creditos

S. No	Capítulos	Contributors
1	Empezando con unity3d	Alexey Shimansky, Chris McFarland, Community, Desutoroiya, driconmax, Ḟ̣áḿíṅḡ ỏḿƀíé, James Radvan, josephsw, Linus Juhlin, Luís Fonseca, Maarten Bicknese, martinhodler, matiaslauriti, Mike B, Minzkraut, PlanetVaster, R.K123, S. Tarık Çetin, Skyblade, SourabhV, SP., tenpn, tim, user3071284
2	Agrupación de objetos	Chris McFarland, Ed Marty, lase, matiaslauriti, S. Tarık Çetin, Thulani Chivandikwa, Thundernerd, ɬolɐəz əɥɬ qoq, volvis
3	Animación de la unidad	4444, Fiery Raccoon, Guglie
4	API de CullingGroup	volvis
5	Atributos	4444, Thundernerd
6	Capas	Arijoon, dreadnought, Light Drake, RamenChef, Skyblade
7	Colisión	Ḟ̣áḿíṅḡ ỏḿƀíé, jjhavokk, Xander Luciano
8	Cómo utilizar paquetes de activos	Ḟ̣áḿíṅḡ ỏḿƀíé
9	Comunicación con el servidor	David Martinez, devon t, Ḟ̣áḿíṅḡ ỏḿƀíé, Maxim Kamalov, tim
10	Coroutines	agiro, Fattie, Fehr, Giuseppe De Francesco, Problematic, Skyblade, Thulani Chivandikwa, Thundernerd, ɬolɐəz əɥɬ qoq, volvis
11	Cuaterniones	matiaslauriti, Tiziano Coroneo, Xander Luciano, yummypasta
12	Desarrollo multiplataforma	user3797758, volvis
13	Encontrar y coleccionar GameObjects	Pierrick Bignet, S. Tarık Çetin, volvis
14	Etiquetas	Arijoon, Augure, glaubergft, Gnemlock, MadJlzz, Skyblade, Trent

15	Extendiendo el Editor	Pierrick Bignet, Skyblade, Thundernerd, ɬolɐəz əɥɬ qoq, volvis
16	Física	eunoia, Ḟ̣ámíṅḡ ỏḿƀié, jack jay
17	Iluminación de la unidad	Ḟ̣ámíṅḡ ỏḿƀié
18	Implementación de la clase MonoBehaviour.	matiaslauriti, Skyblade, Thundernerd, user3797758
19	Importadores y (Post) Procesadores	gman, Skyblade, volvis
20	Integración de anuncios	ɬolɐəz əɥɬ qoq
21	Mejoramiento	Ed Marty, EvilTak, Ḟ̣ámíṅḡ ỏḿƀié, Grigory, JohnTube, Skyblade, Thulani Chivandikwa, volvis
22	Patrones de diseño	Ian Newland
23	Plataformas moviles	Airwarfare, Skyblade
24	Plugins de Android 101 - Una introducción	Venkat at Axiom Studios
25	Prefabricados	Brandon Mintern, Dávid Florek, Ḟ̣ámíṅḡ ỏḿƀié, gman, Gnemlock, Guglie, James Radvan, Jean Vitor, josephsw, Lich, matiaslauriti, Skyblade, Thulani Chivandikwa, ɬolɐəz əɥɬ qoq, Woltus, yummypasta
26	Raycast	driconmax, Meinkraft, Skyblade, user3570542, volvis, wouterrobot
27	Realidad Virtual (VR)	4444, Airwarfare, Guglie, pew., Pratham Sehgal, tim
28	Recursos	glaubergft, MadJlzz, Skyblade, Venkat at Axiom Studios
29	Redes	David Martinez, driconmax, Rafiwui, RamenChef
30	ScriptableObject	volvis
31	Singletons en la unidad	David Darias, Fehr, James Radvan, JohnTube, matiaslauriti, Maxim Kamalov, Simon Heinen, SP., Tiziano Coroneo, Umair M , volvis, Zze,
32	Sistema de audio	R4mbi, ɬolɐəz əɥɬ qoq

33	Sistema de entrada	Programmer, Skyblade, ɬolɐəz əuɬ qoq
34	Sistema de interfaz de usuario (UI)	Hellium, matiaslauriti, Maxim Kamalov, Programmer, RamenChef, Skyblade, Umair M
35	Sistema de interfaz de usuario gráfico de modo inmediato (IMGUI)	Skyblade, Soaring Code
36	Tienda de activos	JakeD, Trent, zwcloud
37	Transforma	ADB, Jean Vitor, matiaslauriti, S. Tarık Çetin, Skyblade, Thundernerd, Xander Luciano
38	Unity Profiler	Amitayu Chakraborty, ForceMagic, RamenChef, Skyblade
39	Usando el control de fuente Git con Unity	Commodore Yournero, Hacky, James Radvan, matiaslauriti, Max Yankov, Maxim Kamalov, Pierrick Bignet, Ricardo Amores, S. Tarık Çetin, S.Richmond, Skyblade, Thulani Chivandikwa, YsenGrimm, yummypasta
40	Vector3	driconmax, Ḟ̄ḁḿíṅḡ ỏḿƀíé, Gnemlock