



EBook Gratuito

APPENDIMENTO

unity3d

Free unaffiliated eBook created from
Stack Overflow contributors.

#unity3d

Sommario

Di.....	1
Capitolo 1: Iniziare con unity3d	2
Osservazioni.....	2
Versioni.....	2
Examples.....	5
Installazione o configurazione.....	5
Panoramica	5
Installazione	6
Installazione di versioni multiple di unità	6
Editor e codice di base.....	6
disposizione.....	6
Layout Linux.....	7
Uso di base.....	7
Scripting di base.....	8
Layout dell'editor.....	8
Personalizzazione del tuo spazio di lavoro.....	10
Capitolo 2: API CullingGroup	13
Osservazioni.....	13
Examples.....	13
Individuazione delle distanze dell'oggetto.....	13
Cogliendo la visibilità dell'oggetto.....	15
Distanze limite.....	16
Visualizzazione delle distanze limite	16
Capitolo 3: attributi	18
Sintassi.....	18
Osservazioni.....	18
SerializeField	18
Examples.....	19
Attributi comuni dell'ispettore.....	19
Attributi del componente.....	21

Attributi di runtime.....	22
Attributi del menu.....	23
Attributi dell'editor.....	25
Capitolo 4: Collisione.....	29
Examples.....	29
collider.....	29
Box Collider.....	29
Proprietà.....	29
Esempio.....	30
Sphere Collider.....	30
Proprietà.....	30
Esempio.....	31
Capsule Collider.....	31
Proprietà.....	32
Esempio.....	32
Wheel Collider.....	32
Proprietà.....	32
Sospensione Primavera.....	33
Esempio.....	33
Mesh Collider.....	33
Proprietà.....	34
Esempio.....	35
Wheel Collider.....	35
Trigger Collider.....	37
metodi.....	37
Trigger Collider Scripting.....	37
Esempio.....	38
Capitolo 5: Come utilizzare i pacchetti di asset.....	39
Examples.....	39
Pacchetti di attività.....	39
Importazione di un pacchetto .unity.....	39

Capitolo 6: Comunicazione con il server	41
Examples.....	41
Ottenere.....	41
Post semplice (campi postali).....	41
Posta (Carica un file).....	42
Carica un file zip sul server.....	42
Invio di una richiesta al server.....	42
Capitolo 7: coroutine	45
Sintassi.....	45
Osservazioni.....	45
Considerazioni sulle prestazioni	45
Riduci la spazzatura memorizzando nella cache YieldInstructions.....	45
Examples.....	45
coroutine.....	45
Esempio	47
Termina una coroutine.....	47
Metodi MonoBehaviour che possono essere Coroutine.....	49
Coroutine a catena.....	49
Modi per cedere.....	51
Capitolo 8: Estendere l'editor	54
Sintassi.....	54
Parametri.....	54
Examples.....	54
Ispettore doganale.....	54
Cassetto delle proprietà personalizzate.....	56
Voci del menu.....	59
aggeggi.....	64
Esempio 1	64
Esempio due	66
Risultato	66
Non selezionato.....	67

Selezionato.....	67
Finestra dell'editor.....	68
Perché una finestra dell'editor?.....	68
Crea una finestra Editor di base.....	68
Semplice esempio.....	68
Andando più a fondo.....	69
Argomenti avanzati.....	72
Disegnare in SceneView.....	72
Capitolo 9: Fisica.....	76
Examples.....	76
Rigidbody.....	76
Panoramica.....	76
Aggiunta di un componente Rigidbody.....	76
Spostare un oggetto Rigidbody.....	76
Massa.....	76
Trascinare.....	76
isKinematic.....	77
vincoli.....	77
collisioni.....	77
Gravità nel corpo rigido.....	78
Capitolo 10: Implementazione della classe MonoBehaviour.....	80
Examples.....	80
Nessun metodo sottoposto a override.....	80
Capitolo 11: Importatori e (Post) Processori.....	81
Sintassi.....	81
Osservazioni.....	81
Examples.....	81
Postprocessor Texture.....	81
Un importatore di base.....	82
Capitolo 12: Integrazione annunci.....	86
introduzione.....	86

Osservazioni.....	86
Examples.....	86
Nozioni sugli annunci Unity in C #.....	86
Nozioni sugli annunci Unity in JavaScript.....	87
Capitolo 13: Livelli.....	88
Examples.....	88
Utilizzo dei livelli.....	88
Struttura LayerMask.....	88
Capitolo 14: Modelli di progettazione.....	90
Examples.....	90
Modello di progettazione Model View Controller (MVC).....	90
Capitolo 15: Negozio di beni.....	94
Examples.....	94
Accedere al negozio di beni.....	94
Acquisti di beni.....	94
Importazione di beni.....	95
Attività editoriali.....	95
Conferma il numero di fattura di un acquisto.....	96
Capitolo 16: Networking.....	97
Osservazioni.....	97
Modalità senza testa in Unity.....	97
Examples.....	98
Creazione di un server, un client e invio di un messaggio.....	98
La classe che stiamo usando per serializzare.....	98
Creazione di un server.....	98
Il cliente.....	100
Capitolo 17: Ottimizzazione.....	102
Osservazioni.....	102
Examples.....	102
Controlli veloci ed efficienti.....	102
Controllo distanza / intervallo.....	102

Controlli limitati	102
Avvertenze	102
Coroutine Power	102
uso	102
Divisione di routine a esecuzione prolungata su più frame	103
Esecuzione di azioni costose meno frequentemente	103
Insidie comuni	104
stringhe	104
Le operazioni con le stringhe costruiscono garbage	104
Memorizza le tue operazioni sulle stringhe	104
La maggior parte delle operazioni con le stringhe sono messaggi di debug	105
Confronto di stringhe	106
Riferimenti di cache	106
Evita di chiamare metodi usando stringhe	107
Evita i metodi di unità vuoti	108
Capitolo 18: Piattaforme mobili	109
Sintassi	109
Examples	109
Rilevazione del tocco	109
TouchPhase	109
Capitolo 19: Plugin Android 101 - Un'introduzione	111
introduzione	111
Osservazioni	111
A partire dai plugin Android	111
Cenni sulla creazione di un plug-in e una terminologia	111
Scegliere tra i metodi di creazione del plugin	112
Examples	112
UnityAndroidPlugin.cs	112
UnityAndroidNative.java	112
UnityAndroidPluginGUI.cs	113
Capitolo 20: Pooling di oggetti	114

Examples.....	114
Pool di oggetti.....	114
Pool di oggetti semplice.....	116
Un altro pool di oggetti semplice.....	118
Capitolo 21: prefabbricati.....	120
Sintassi.....	120
Examples.....	120
introduzione.....	120
Creazione di prefabbricati.....	120
Ispettore prefabbricato.....	121
Prefabbricati istanziati.....	122
Creazione di istanze temporali.....	122
Istanziamento runtime.....	123
Prefabbricati annidati.....	123
Capitolo 22: quaternions.....	128
Sintassi.....	128
Examples.....	128
Introduzione a Quaternion vs Eulero.....	128
Quaternion Look Rotation.....	128
Capitolo 23: raycast.....	130
Parametri.....	130
Examples.....	130
Raggio di fisica.....	130
Physics2D Raycast2D.....	130
Incapsulare chiamate Raycast.....	131
Ulteriori letture.....	132
Capitolo 24: Realtà virtuale (VR).....	133
Examples.....	133
Piattaforme VR.....	133
SDK:.....	133
Documentazione:.....	133

Abilitazione del supporto VR	133
Hardware	134
Capitolo 25: risorse	136
Examples	136
introduzione	136
Risorse 101	136
introduzione	136
Mettere tutto insieme	137
Note finali	137
Capitolo 26: ScriptableObject	139
Osservazioni	139
ScriptableObjects con AssetBundles	139
Examples	139
introduzione	139
Creazione di risorse ScriptableObject	139
Crea istanze ScriptableObject tramite codice	140
ScriptableObjects sono serializzati nell'editor anche in PlayMode	140
Trova ScriptableObjects esistente durante il runtime	141
Capitolo 27: Singletons in Unity	142
Osservazioni	142
Ulteriori letture	142
Examples	142
Implementazione utilizzando RuntimeInitializeOnLoadMethodAttribute	143
Un semplice Singleton MonoBehaviour in Unity C #	143
Advanced Unity Singleton	144
Implementazione di Singleton attraverso la classe base	146
Singleton Pattern che utilizza il sistema Entity-Component di Unitys	148
Classe Singleton basata su MonoBehaviour & ScriptableObject	149
Capitolo 28: Sistema audio	154
introduzione	154
Examples	154

Classe audio - Riproduci audio.....	154
Capitolo 29: Sistema di input.....	155
Examples.....	155
Lettura della chiave Stampa e differenza tra GetKey, GetKeyDown e GetKeyUp.....	155
Leggi sensore accelerometro (base).....	156
Leggi sensore accelerometro (Advance).....	156
Leggi sensore accelerometro (precisione).....	157
Leggi il pulsante del mouse (sinistra, metà, destra) Clic.....	158
Capitolo 30: Sistema di interfaccia utente (UI).....	161
Examples.....	161
Iscrivendosi all'evento nel codice.....	161
Aggiungere i listener del mouse.....	161
Capitolo 31: Sistema di interfaccia utente grafica in modalità immediata (IMGUI).....	163
Sintassi.....	163
Examples.....	163
GUILayout.....	163
Capitolo 32: Sviluppo multiplatforma.....	164
Examples.....	164
Definizioni del compilatore.....	164
Organizzazione di metodi specifici per piattaforma per classi parziali.....	164
Capitolo 33: tag.....	166
introduzione.....	166
Examples.....	166
Creazione e applicazione di tag.....	166
Impostazione dei tag nell'editor.....	166
Impostazione dei tag tramite Script.....	166
Creazione di tag personalizzati.....	167
Ricerca di oggetti di gioco per tag:.....	168
Trovare un singolo GameObject.....	168
Trovare una matrice di istanze GameObject.....	169
Confronto di tag.....	169
Capitolo 34: Trasformazioni.....	171

Sintassi.....	171
Examples.....	171
Panoramica.....	171
Parenting e bambini.....	172
Capitolo 35: Trovare e collezionare GameObjects.....	174
Sintassi.....	174
Osservazioni.....	174
Quale metodo usare.....	174
Andando più a fondo.....	174
Examples.....	175
Ricerca per nome di GameObject.....	175
Ricerca per tag di GameObject.....	175
Inserito negli script in modalità Modifica.....	175
Trovare gli oggetti GameObjects tramite MonoBehaviour.....	175
Trova GameObjects per nome dagli oggetti figlio.....	176
Capitolo 36: Unity Animation.....	177
Examples.....	177
Animazione di base per l'esecuzione.....	177
Creazione e utilizzo di clip di animazione.....	178
Animazione 2D Sprite.....	180
Curve di animazione.....	182
Capitolo 37: Unity Lighting.....	185
Examples.....	185
Tipi di luce.....	185
Area Light.....	185
Luce direzionale.....	185
Punto luce.....	186
Riflettore.....	187
Nota su Ombre.....	188
emissione.....	189
Capitolo 38: Unity Profiler.....	191

Osservazioni.....	191
Utilizzo di Profiler su un altro dispositivo.....	191
androide.....	191
iOS.....	192
Examples.....	192
Profiler Markup.....	192
Utilizzo della classe Profiler.....	192
Capitolo 39: Utilizzo del controllo del codice sorgente Git con Unity.....	194
Examples.....	194
Utilizzo di Git Large File Storage (LFS) con Unity.....	194
Prefazione.....	194
Installazione di Git & Git-LFS.....	194
Opzione 1: utilizzare un'applicazione Git GUI.....	194
Opzione 2: installa Git & Git-LFS.....	195
Configurazione di Git Large File Storage sul tuo progetto.....	195
Impostazione di un repository Git per Unity.....	195
Unity Ignora le cartelle.....	195
Impostazioni del progetto Unity.....	196
Configurazione aggiuntiva.....	196
Fusione di scene e prefabbricati.....	197
Capitolo 40: Vector3.....	198
introduzione.....	198
Sintassi.....	198
Examples.....	198
Valori statici.....	198
Vector3.zero e Vector3.one.....	198
Indicazioni statiche.....	199
Indice.....	201
Creare un Vector3.....	201
Costruttori.....	201

Conversione da un Vector2 o Vector4	202
Applicazione del movimento.....	202
Lerp e LerpUnclamped.....	203
MoveTowards.....	204
SmoothDamp.....	205
Titoli di coda	208

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Capitolo 1: Iniziare con unity3d

Osservazioni

Unity fornisce un ambiente di sviluppo di giochi multiplatforma per gli sviluppatori. Gli sviluppatori possono utilizzare il linguaggio C # e / o la sintassi basata su JavaScript UnityScript per programmare il gioco. Le piattaforme di distribuzione target possono essere cambiate facilmente nell'editor. Tutti i codici di gioco di base rimangono identici tranne alcune funzioni dipendenti dalla piattaforma. Un elenco di tutte le versioni e i download e le note di rilascio corrispondenti sono disponibili qui: <https://unity3d.com/get-unity/download/archive> .

Versioni

Versione	Data di rilascio
Unity 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

Versione	Data di rilascio
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

Versione	Data di rilascio
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

Versione	Data di rilascio
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

Installazione o configurazione

Panoramica

Unity funziona su Windows e Mac. È disponibile anche una [versione alfa di Linux](#) .

Esistono 4 diversi piani di pagamento per Unity:

1. **Personale** - Gratuito (*vedi sotto*)
2. **Plus** - \$ 35 USD al mese per posto (*vedi sotto*)
3. **Pro** - \$ 125 USD al mese per posto - Dopo aver sottoscritto il piano Pro per 24 mesi consecutivi, hai la possibilità di interrompere la sottoscrizione e mantenere la versione che hai.
4. **Enterprise** - [Contact Unity per ulteriori informazioni](#)

Secondo EULA: le società o le entità incorporate che hanno registrato un fatturato superiore a 100.000 USD nel loro ultimo anno fiscale devono utilizzare **Unity Plus** (o una licenza più elevata); superiori a US \$ 200.000 devono utilizzare **Unity Pro** (o Enterprise).

Installazione

1. Scarica l' [assistente per il download di Unity](#) .
2. Eseguire l'assistente e scegliere i moduli che si desidera scaricare e installare, come editor Unity, IDE di MonoDevelop, documentazione e moduli di build della piattaforma desiderati.

Se si dispone di una versione precedente, è possibile [eseguire l'aggiornamento all'ultima versione stabile](#) .

Se si desidera installare Unity download Unity assistant, è possibile ottenere i programmi di **installazione dei componenti** dalle [note di rilascio di Unity 5.5.1](#) .

Installazione di versioni multiple di unità

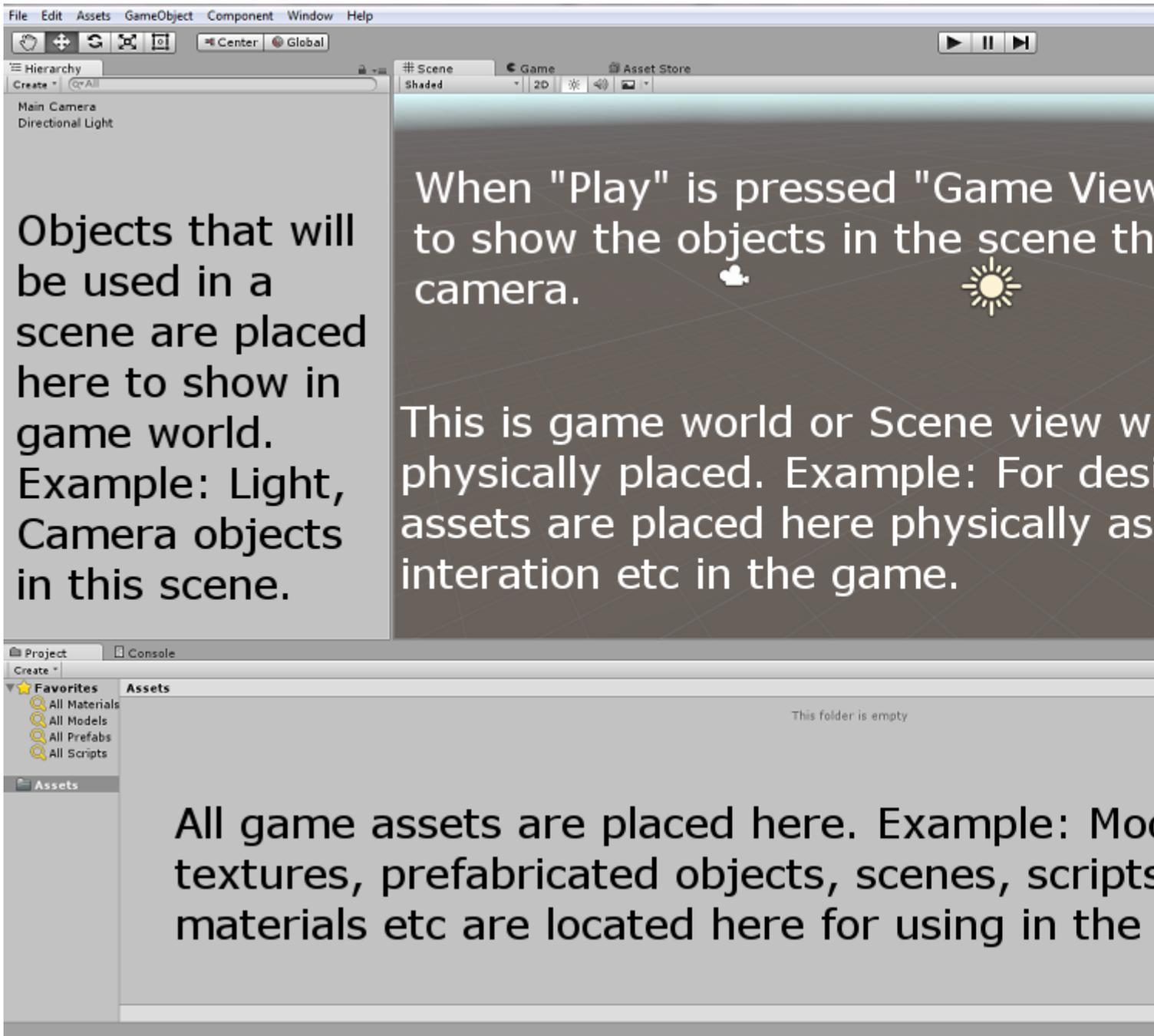
Spesso è necessario installare più versioni di Unity contemporaneamente. Fare così:

- Su Windows, cambia la directory di installazione predefinita in una cartella vuota precedentemente creata come `Unity 5.3.1f1` .
- Su Mac, il programma di installazione verrà sempre installato in `/Applications/Unity` . Rinominare questa cartella per l'installazione esistente (ad esempio in `/Applications/Unity5.3.1f1`) prima di eseguire il programma di installazione per la versione diversa.
- Puoi tenere premuto `Alt` all'avvio di Unity per forzarlo a consentire la scelta di un progetto da aprire. In caso contrario, l'ultimo progetto caricato tenderà di caricare (se disponibile) e potrebbe richiedere di aggiornare un progetto che non si desidera aggiornare.

Editor e codice di base

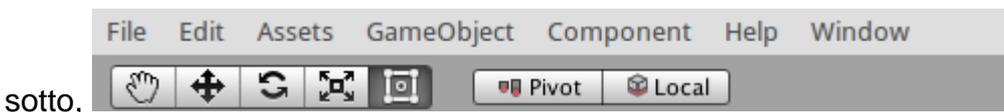
disposizione

L'editor di base di Unity sarà visualizzato di seguito. Le funzionalità di base di alcune finestre / tab predefinite sono descritte nell'immagine.



Layout Linux

C'è una piccola differenza nel layout del menu della versione di Linux, come lo screenshot qui



Uso di base

Creare un oggetto `GameObject` vuoto `GameObject` clic con il tasto destro nella finestra Gerarchia e selezionare `Create Empty`. Crea un nuovo script facendo clic con il pulsante destro del mouse nella finestra Progetto e seleziona `Create > C# Script`. Rinominalo secondo necessità.

Quando il `GameObject` vuoto è selezionato nella finestra Gerarchia, trascina e rilascia lo script

appena creato nella finestra Inspector. Ora lo script è collegato all'oggetto nella finestra Gerarchia. Apri lo script con l'IDE MonoDevelop predefinito o le tue preferenze.

Scripting di base

Il codice di base sarà simile sotto eccetto la riga `Debug.Log("hello world!!");` .

```
using UnityEngine;
using System.Collections;

public class BasicCode : MonoBehaviour {

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

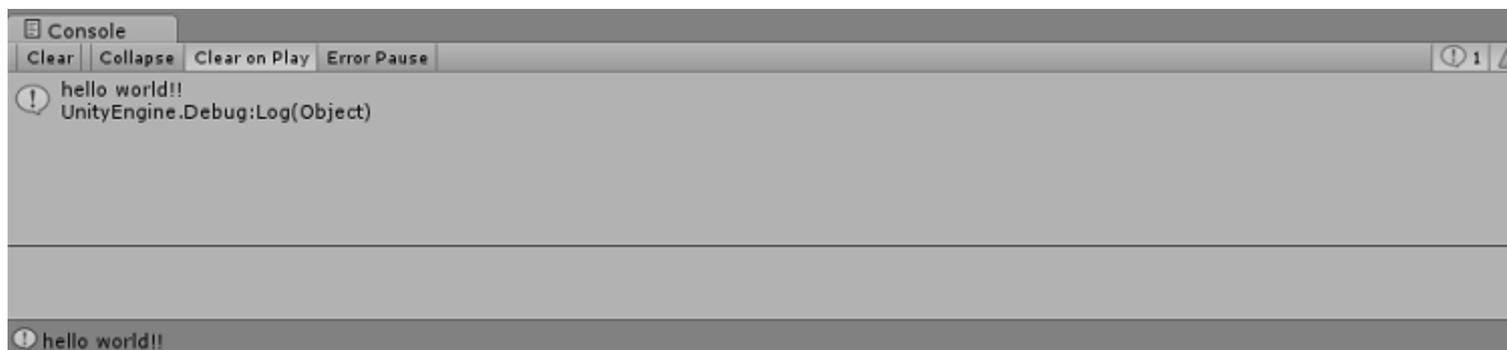
    // Update is called once per frame
    void Update () {

    }

}
```

Aggiungi la riga `Debug.Log("hello world!!");` nel metodo `void Start()` . Salva lo script e torna all'editor. Eseguilo premendo **Play** nella parte superiore dell'editor.

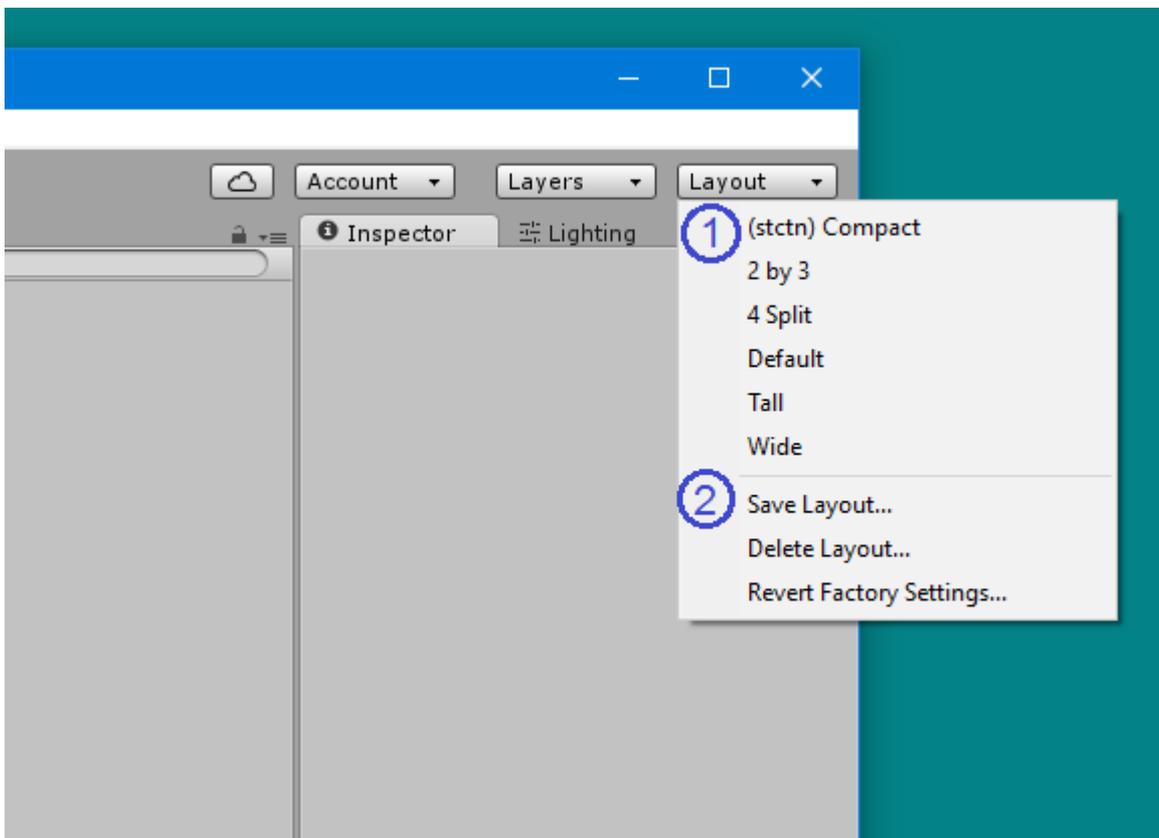
Il risultato dovrebbe essere come sotto nella finestra della console:



Layout dell'editor

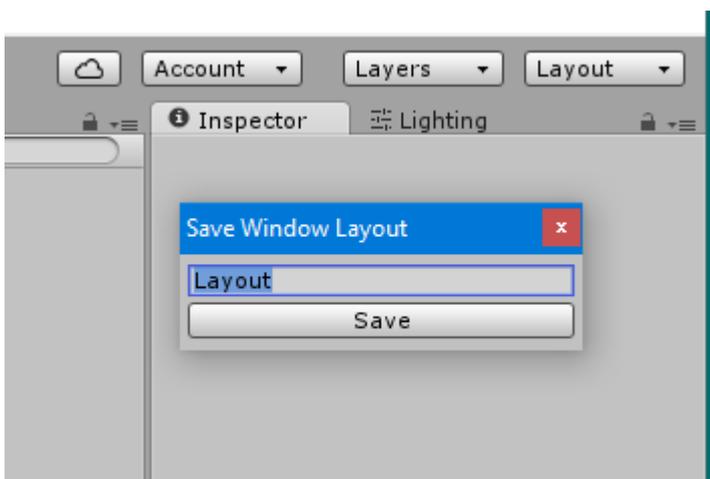
Puoi salvare il layout delle tue schede e finestre per standardizzare il tuo ambiente di lavoro.

Il menu dei layout è disponibile nell'angolo in alto a destra di Unity Editor:

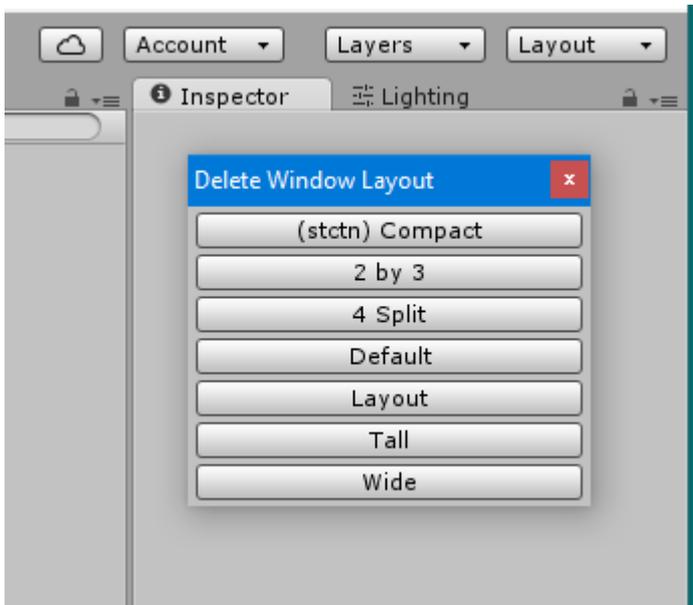


Unity viene fornito con 5 layout predefiniti (2 per 3, 4 suddivisi, predefinito, alto, largo) (contrassegnato con 1). Nella figura sopra, oltre ai layout di default, c'è anche un layout personalizzato nella parte superiore.

È possibile aggiungere i propri layout facendo clic sul pulsante "**Salva layout ...**" nel menu (contrassegnato con 2):



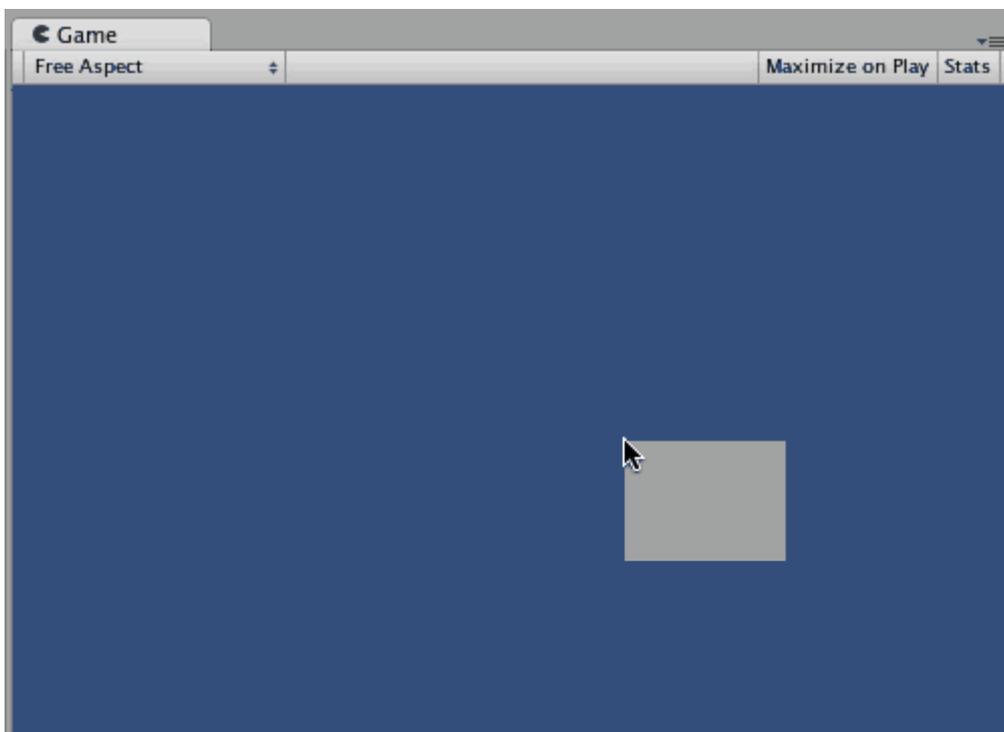
Puoi anche eliminare qualsiasi layout facendo clic sul pulsante "**Elimina layout ...**" nel menu (contrassegnato con 2):



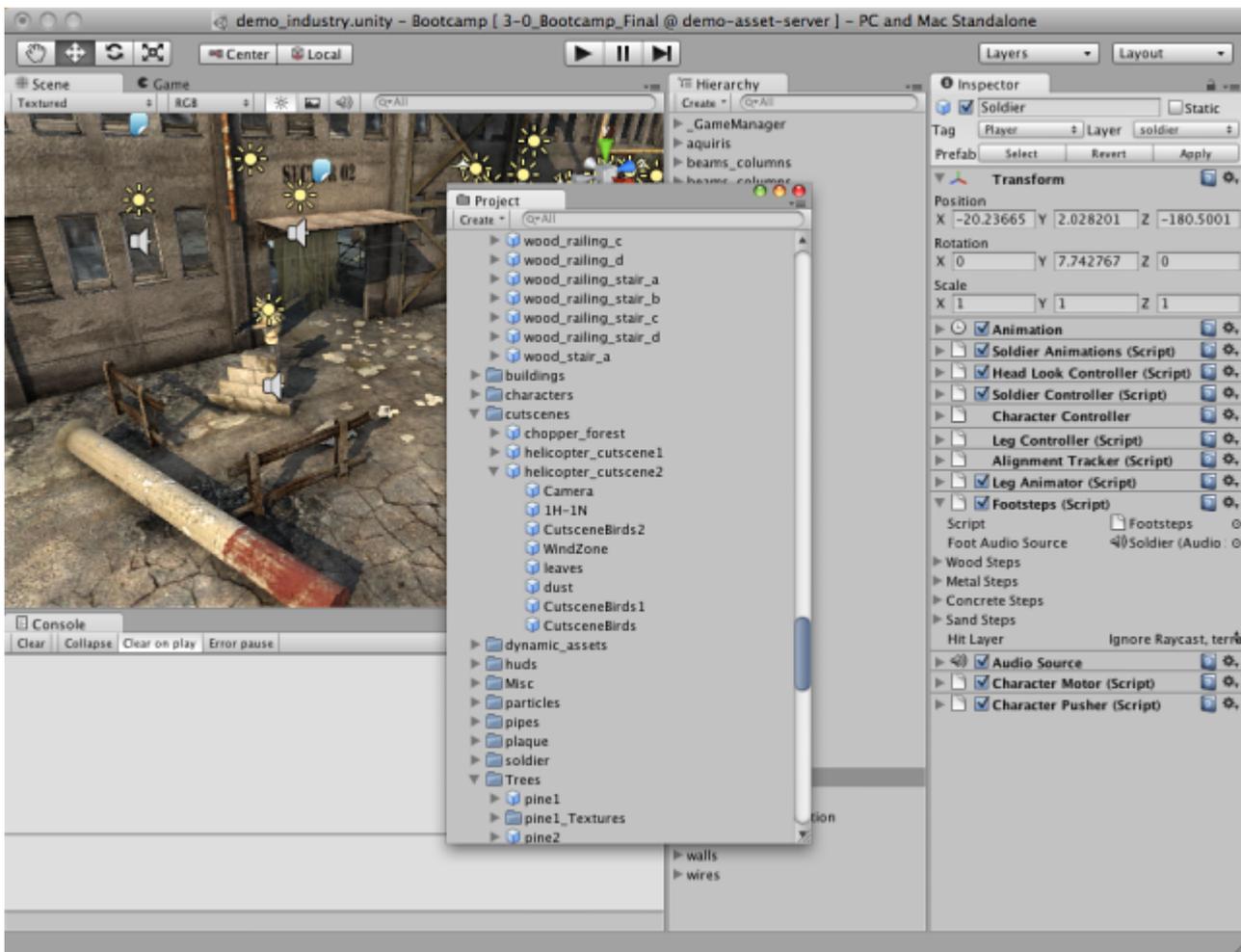
Il pulsante "**Ripristina impostazioni di fabbrica ...**" rimuove tutti i layout personalizzati e ripristina i layout predefiniti (*contrassegnati con 2*).

Personalizzazione del tuo spazio di lavoro

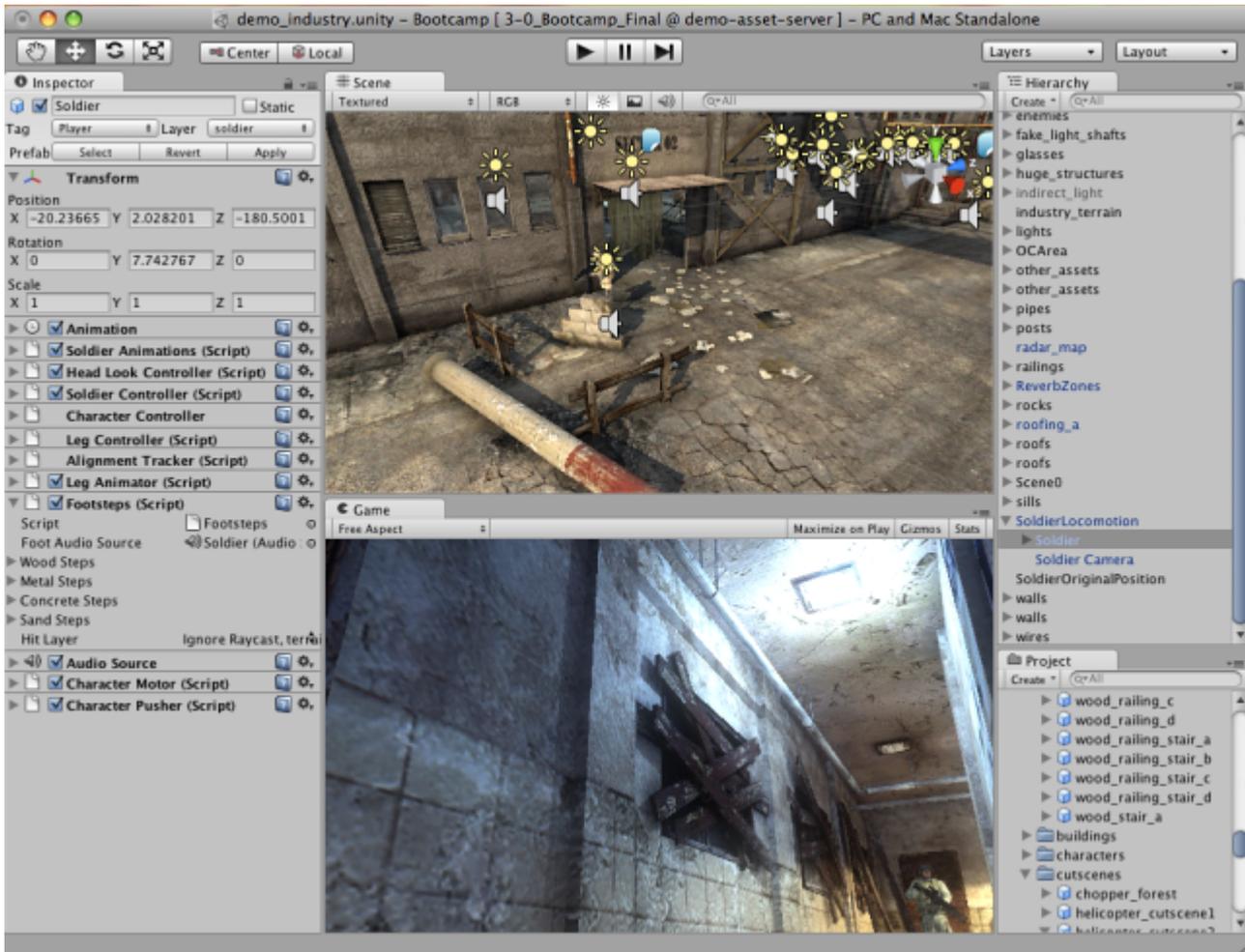
È possibile personalizzare il proprio layout delle viste facendo clic tenendo premuto il tasto Tab di qualsiasi vista in una delle diverse posizioni. L'eliminazione di una scheda nell'area di tabulazione di una finestra esistente aggiungerà la scheda accanto a qualsiasi scheda esistente. In alternativa, l'eliminazione di una scheda in qualsiasi Dock Zone aggiungerà la vista in una nuova finestra.



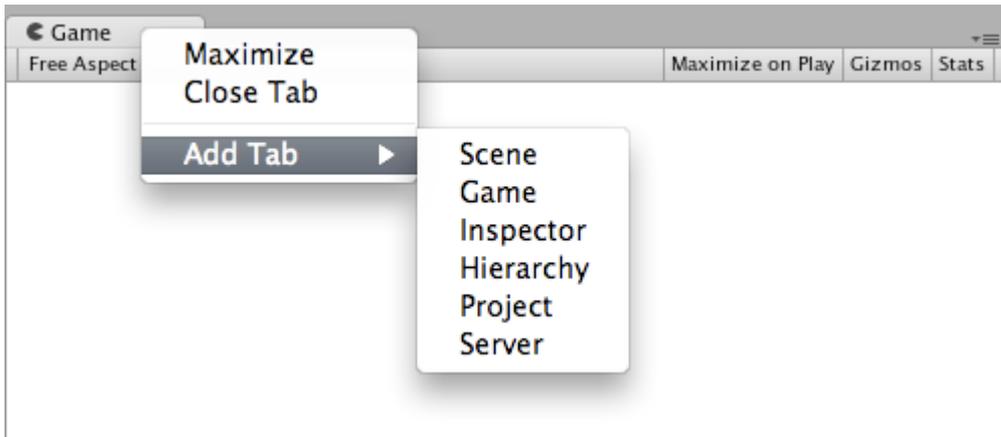
Le schede possono anche essere separate dalla finestra dell'editor principale e disposte nelle proprie finestre di Editor mobili. Le finestre mobili possono contenere arrangiamenti di viste e tabulazioni, proprio come la finestra dell'editor principale.



Quando hai creato un layout di editor, puoi salvare il layout e ripristinarlo in qualsiasi momento. [Fare riferimento a questo esempio per i layout dell'editor](#) .



In qualsiasi momento, puoi fare clic con il tasto destro del mouse sulla scheda di qualsiasi vista per visualizzare opzioni aggiuntive come Ingrandisci o aggiungere una nuova scheda alla stessa finestra.



Leggi Iniziare con unity3d online: <https://riptutorial.com/it/unity3d/topic/846/iniziare-con-unity3d>

Capitolo 2: API CullingGroup

Osservazioni

Poiché usare CullingGroups non è sempre molto semplice, può essere utile incapsulare la maggior parte della logica dietro una classe manager.

Di seguito è riportato un modello su come un simile gestore potrebbe operare.

```
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);
}
```

L'essenza di ciò è che si riserva una sfera di abbattimento dal manager che restituisce l'indice della sfera riservata. Quindi usi l'indice dato per manipolare la tua sfera riservata.

Examples

Individuazione delle distanze dell'oggetto

L'esempio seguente illustra come utilizzare CullingGroups per ottenere notifiche in base al punto di riferimento della distanza.

Questo script è stato semplificato per brevità e utilizza diversi metodi pesanti per le prestazioni.

```
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;
}

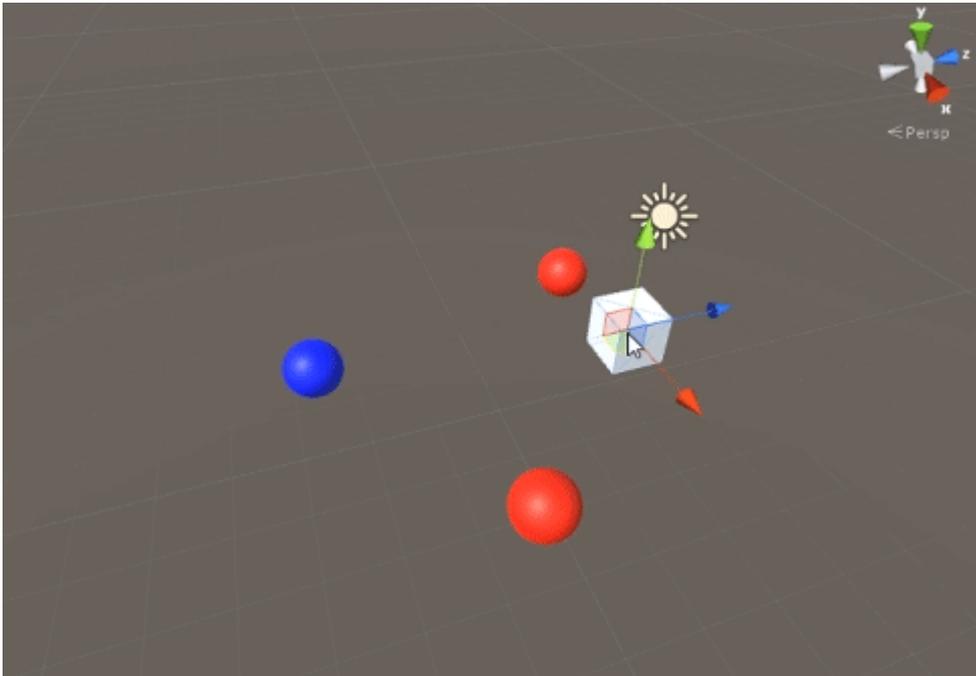
void FixedUpdate()
{
    localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
    for (var i = 0; i < meshTransforms.Length; i++)
    {
        cullingPoints[i].position = meshTransforms[i].position;
    }
}

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();
}
}

```

Aggiungi lo script a un GameObject (in questo caso un cubo) e premi Riproduci. Ogni altro oggetto GameObject nella scena cambia colore in base alla loro distanza dal punto di riferimento.



Cogliendo la visibilità dell'oggetto

Il seguente script illustra come ricevere eventi in base alla visibilità di una telecamera impostata.

Questo script utilizza diversi metodi di prestazioni pesanti per brevità.

```
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;
    }
}
```

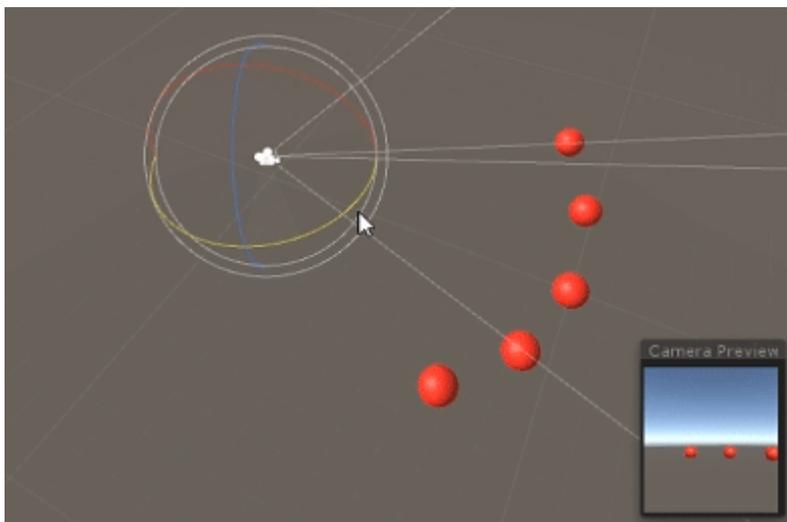
```

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

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

```

Aggiungi lo script alla scena e premi Riproduci. Tutte le geometrie nella scena cambieranno colore in base alla loro visibilità.



Un effetto simile può essere ottenuto usando il metodo `MonoBehaviour.OnBecameVisible()` se l'oggetto ha un componente `MeshRenderer`. Usa `CullingGroups` quando hai bisogno di selezionare `GameObjects` vuoti, coordinate `Vector3` o quando vuoi un metodo centralizzato per monitorare la visibilità degli oggetti.

Distanze limite

È possibile aggiungere distanze di limitazione in cima al raggio del punto di smistamento. Sono in un certo modo condizioni di innesco aggiuntive al di fuori del raggio principale dei punti di abbattimento, come "vicino", "lontano" o "molto lontano".

```

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

```

Le distanze limite influiscono solo quando usate con un punto di riferimento della distanza. Non hanno alcun effetto durante l'eliminazione della telecamera.

Visualizzazione delle distanze limite

Ciò che inizialmente potrebbe causare confusione è il modo in cui le distanze di limitazione

vengono aggiunte sopra i raggi della sfera.

Innanzitutto, il gruppo di selezione calcola l' *area* sia della sfera di delimitazione che della distanza di delimitazione. Le due aree vengono sommate e il risultato è l'area di innesco per la banda di distanza. Il raggio di quest'area può essere usato per visualizzare il campo d'effetto della distanza di delimitazione.

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

Leggi API CullingGroup online: <https://riptutorial.com/it/unity3d/topic/4574/api-cullinggroup>

Capitolo 3: attributi

Sintassi

- [AddComponentMenu (string menuName)]
- [AddComponentMenu (string menuName, int order)]
- [CanEditMultipleObjects]
- [ContextMenu (nome stringa, funzione stringa)]
- [ContextMenu (nome stringa)]
- [CustomEditor (Type inspectedType)]
- [CustomEditor (Type inspectedType, bool editorForChildClasses)]
- [CustomPropertyDrawer (Tipo tipo)]
- [CustomPropertyDrawer (Tipo tipo, bool useForChildren)]
- [DisallowMultipleComponent]
- [DrawGizmo (GizmoType gizmo)]
- [DrawGizmo (GizmoType gizmo, Type drawnGizmoType)]
- [ExecuteInEditMode]
- [Intestazione (intestazione stringa)]
- [HideInInspector]
- [InitializeOnLoad]
- [InitializeOnLoadMethod]
- [MenuItem (stringa itemName)]
- [MenuItem (stringa itemName, bool isValidFunction)]
- [MenuItem (stringa itemName, bool isValidFunction, int priority)]
- [Multiline (int lines)]
- [PreferenceItem (nome stringa)]
- [Range (float min, float max)]
- [RequireComponent (Type type)]
- [RuntimeInitializeOnLoadMethod]
- [RuntimeInitializeOnLoadMethod (RuntimeInitializeLoadType loadType)]
- [SerializeField]
- [Spazio (altezza del galleggiante)]
- [TextArea (int minLines, int maxLines)]
- [Descrizione comando (tooltip stringa)]

Osservazioni

SerializeField

Il sistema di serializzazione di Unity può essere usato per fare quanto segue:

- **Può** serializzare campi non statici pubblici (di tipi serializzabili)
- **Può** serializzare i campi non statici non pubblici contrassegnati con l'attributo [SerializeField]

- **Non è possibile** serializzare i campi statici
- **Impossibile** serializzare le proprietà statiche

Il tuo campo, anche se contrassegnato con l'attributo `SerializeField`, verrà attribuito solo se è di un tipo che Unity può serializzare, ovvero:

- Tutte le classi ereditate da `UnityEngine.Object` (es. `GameObject`, `Component`, `MonoBehaviour`, `Texture2D`)
- Tutti i tipi di dati di base come `int`, `string`, `float`, `bool`
- Alcuni tipi built-in come `Vector2 / 3/4`, `Quaternion`, `Matrix4x4`, `Color`, `Rect`, `LayerMask`
- Matrici di un tipo serializzabile
- Elenco di un tipo serializzabile
- Enums
- Structs

Examples

Attributi comuni dell'ispettore

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

[HideInInspector]
public string MyHiddenString;

[Multiline( 5 )]
public string MyMultilineString;

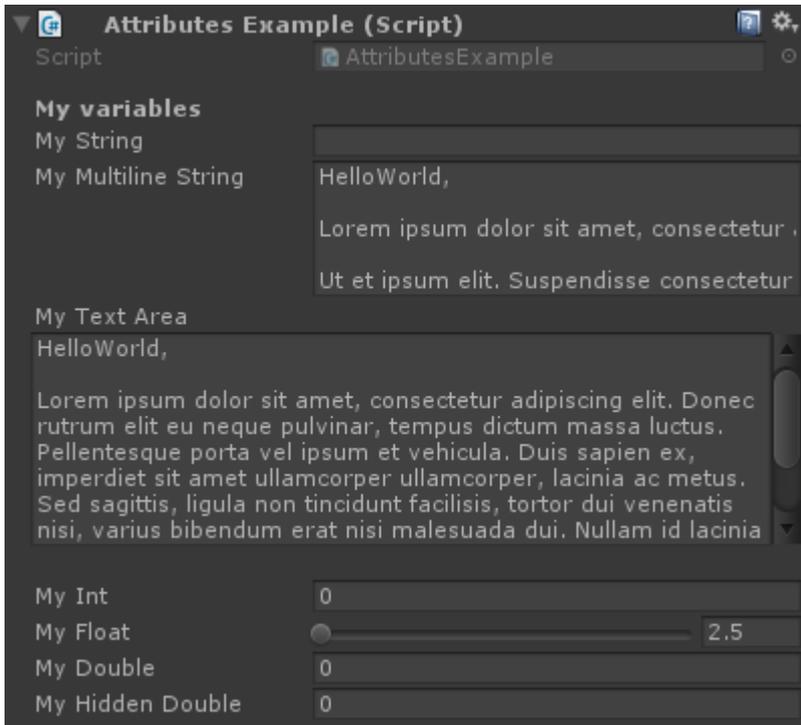
[TextArea( 2, 8 )]
public string MyTextArea;

[Space( 15 )]
public int MyInt;

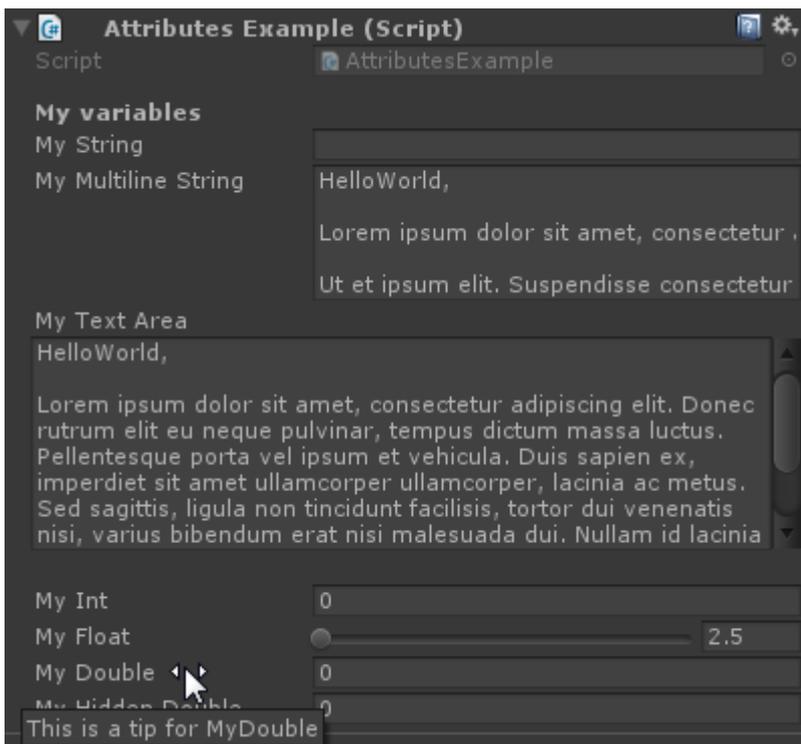
[Range( 2.5f, 12.5f )]
public float MyFloat;

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

[SerializeField]
private double myHiddenDouble;
```



Quando si passa sopra l'etichetta di un campo:



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

L'intestazione posiziona un'etichetta in grassetto contenente il testo sopra il campo attribuito. Questo è spesso usato per etichettare i gruppi per farli risaltare rispetto ad altre etichette.

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

HideInInspector impedisce che i campi pubblici vengano mostrati nell'ispettore. Questo è utile per accedere ai campi da altre parti del codice dove non sono altrimenti visibili o mutabili.

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

Multiline crea una casella di testo con un numero di righe specificato. Il superamento di questo importo non espanderà la scatola né avvolgerà il testo.

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

TextArea consente il testo in stile multilinea con il wordwrap automatico e le barre di scorrimento se il testo supera l'area assegnata.

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

Lo spazio costringe l'ispettore ad aggiungere ulteriore spazio tra gli elementi precedenti e attuali, utile per distinguere e separare i gruppi.

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

L'intervallo forza un valore numerico tra un minimo e un massimo. Questo attributo funziona anche su numeri interi e doppi, anche se min e max sono specificati come float.

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

Tooltip mostra una descrizione aggiuntiva ogni volta che l'etichetta del campo viene posizionata sopra il puntatore.

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

SerializeField forza Unity a serializzare il campo, utile per i campi privati.

Attributi del componente

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

```
[DisallowMultipleComponent]
```

L'attributo `DisallowMultipleComponent` impedisce agli utenti di aggiungere più istanze di questo componente a un `GameObject`.

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

L'attributo `RequireComponent` consente di specificare un altro componente (o più) come requisiti per quando questo componente viene aggiunto a un oggetto `GameObject`. Quando aggiungi questo componente a un `GameObject`, i componenti richiesti verranno aggiunti automaticamente (se non già presenti) e tali componenti non potranno essere rimossi finché non verrà rimosso quello che li richiede.

Attributi di runtime

```
[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
```

L'attributo `ExecuteInEditMode` forza Unity ad eseguire i metodi magici di questo script anche mentre il gioco non è in esecuzione.

Le funzioni non vengono costantemente chiamate come nella modalità di riproduzione

- L'aggiornamento viene chiamato solo quando qualcosa nella scena è cambiato.
- OnGUI viene chiamato quando la vista di gioco riceve un evento.
- OnRenderObject e le altre funzioni di callback di rendering sono richiamate su ogni ridisegno della vista scena o vista gioco.

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

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

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

L'attributo `RuntimeInitializeOnLoadMethod` consente di chiamare un metodo di classe runtime quando il gioco carica il runtime, senza alcuna interazione da parte dell'utente.

È possibile specificare se si desidera che il metodo venga richiamato prima o dopo il caricamento della scena (dopo l'impostazione predefinita). L'ordine di esecuzione non è garantito per i metodi che utilizzano questo attributo.

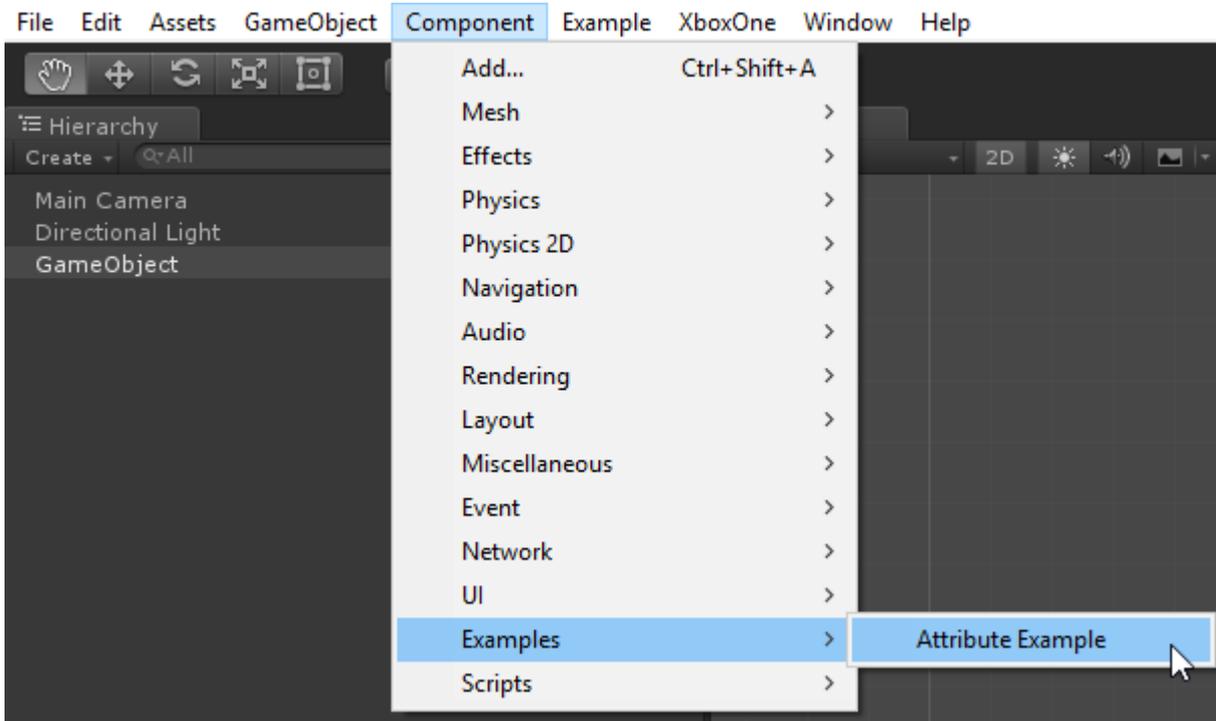
Attributi del menu

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

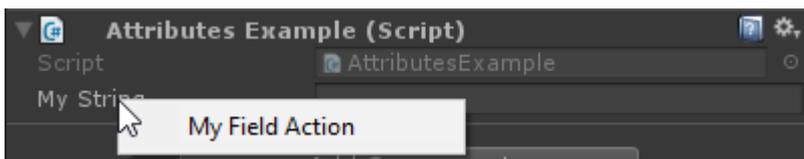
    private void MyFieldContextAction()
    {
        [...]
    }

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

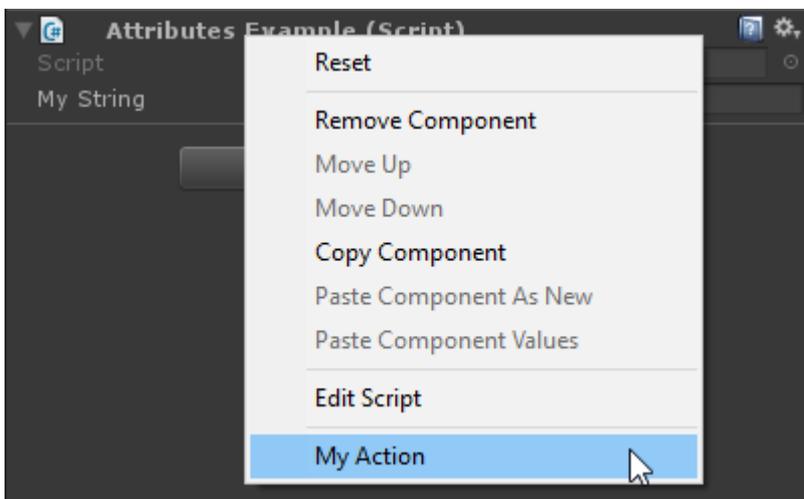
Il risultato dell'attributo `[AddComponentMenu]`



Il risultato dell'attributo [ContextMenuItem]



Il risultato dell'attributo [ContextMenu]



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

L'attributo AddComponentMenu consente di posizionare il componente in qualsiasi punto del menu Componente invece del menu Componente-> Script.

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

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

L'attributo `ContextMenuItem` consente di definire funzioni che possono essere aggiunte al menu di scelta rapida di un campo. Queste funzioni saranno eseguite alla selezione.

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

L'attributo `ContextMenu` consente di definire le funzioni che possono essere aggiunte al menu di scelta rapida del componente.

Attributi dell'editor

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{
    static AttributesExample()
    {
        [...]
    }

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

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{
    static AttributesExample()
    {
        [...]
    }
}
```

L'attributo `InitializeOnLoad` consente all'utente di inizializzare una classe senza alcuna interazione da parte dell'utente. Ciò accade ogni volta che l'editor si avvia o su una ricompilazione. Il costruttore statico garantisce che questo verrà chiamato prima di qualsiasi altra funzione statica.

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

```
}
```

L'attributo `InitializeOnLoad` consente all'utente di inizializzare una classe senza alcuna interazione da parte dell'utente. Ciò accade ogni volta che l'editor si avvia o su una ricompilazione. L'ordine di esecuzione non è garantito per i metodi che utilizzano questo attributo.

```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
{
    public int MyInt;

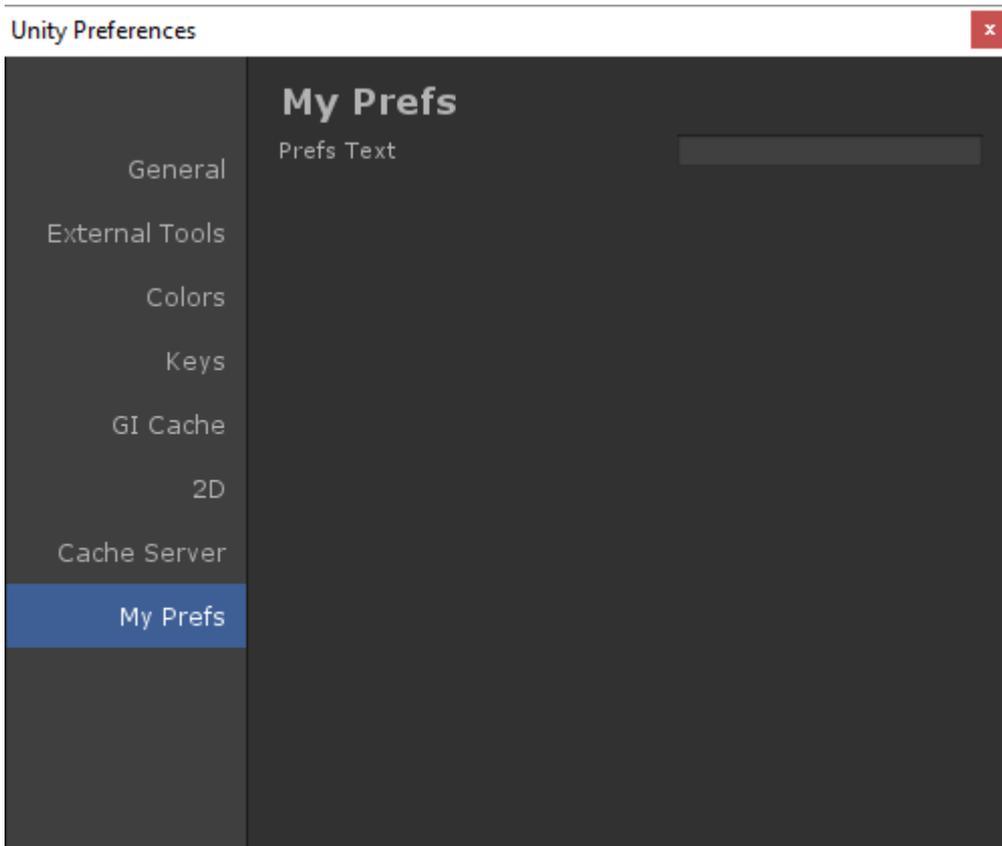
    private static string prefsText = "";

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

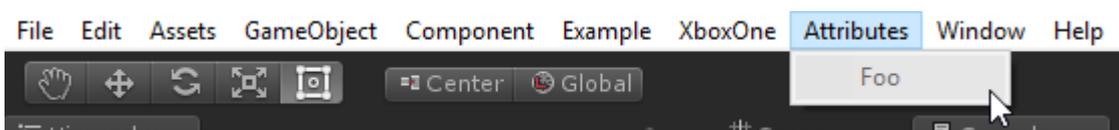
    [MenuItem( "Attributes/Foo" )]
    private static void Foo()
    {
        [...]
    }

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

Il risultato dell'attributo `[PreferenceItem]`



Il risultato dell'attributo [MenuItem]



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

L'attributo `CanEditMultipleObjects` consente di modificare i valori dal componente su più oggetti `GameObject`. Senza questo componente non vedrai il tuo componente apparire come normale quando selezioni più `GameObject` ma vedrai invece il messaggio "Modifica di più oggetti non supportata"

Questo attributo è per gli editor personalizzati che supportano il multi editing. Gli editor non personalizzati supportano automaticamente il multi editing.

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

L'attributo `PreferenceItem` ti consente di creare un oggetto extra nel menu delle preferenze di Unity. Il metodo di ricezione deve essere statico per poter essere utilizzato.

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

```
}

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

L'attributo MenuItem consente di creare voci di menu personalizzate per l'esecuzione delle funzioni. Questo esempio utilizza anche una funzione di convalida (che restituisce sempre false) per impedire l'esecuzione della funzione.

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

L'attributo CustomEditor ti consente di creare editor personalizzati per i tuoi componenti. Questi editor verranno utilizzati per disegnare il componente nell'ispettore e devono derivare dalla classe Editor.

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

L'attributo CustomPropertyDrawer consente di creare un cassetto delle proprietà personalizzate per nell'ispettore. Puoi usare questi cassette per i tuoi tipi di dati personalizzati in modo che possano essere visti usati nell'ispettore.

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

L'attributo DrawGizmo ti consente di disegnare oggetti personalizzati per i tuoi componenti. Questi oggetti saranno disegnati nella vista scena. Puoi decidere quando disegnare il gizmo usando il parametro GizmoType nell'attributo DrawGizmo.

Il metodo di ricezione richiede due parametri, il primo è il componente per disegnare il gizmo e il secondo è lo stato in cui si trova l'oggetto che ha bisogno del disegno del gizmo.

Leggi attributi online: <https://riptutorial.com/it/unity3d/topic/5535/attributi>

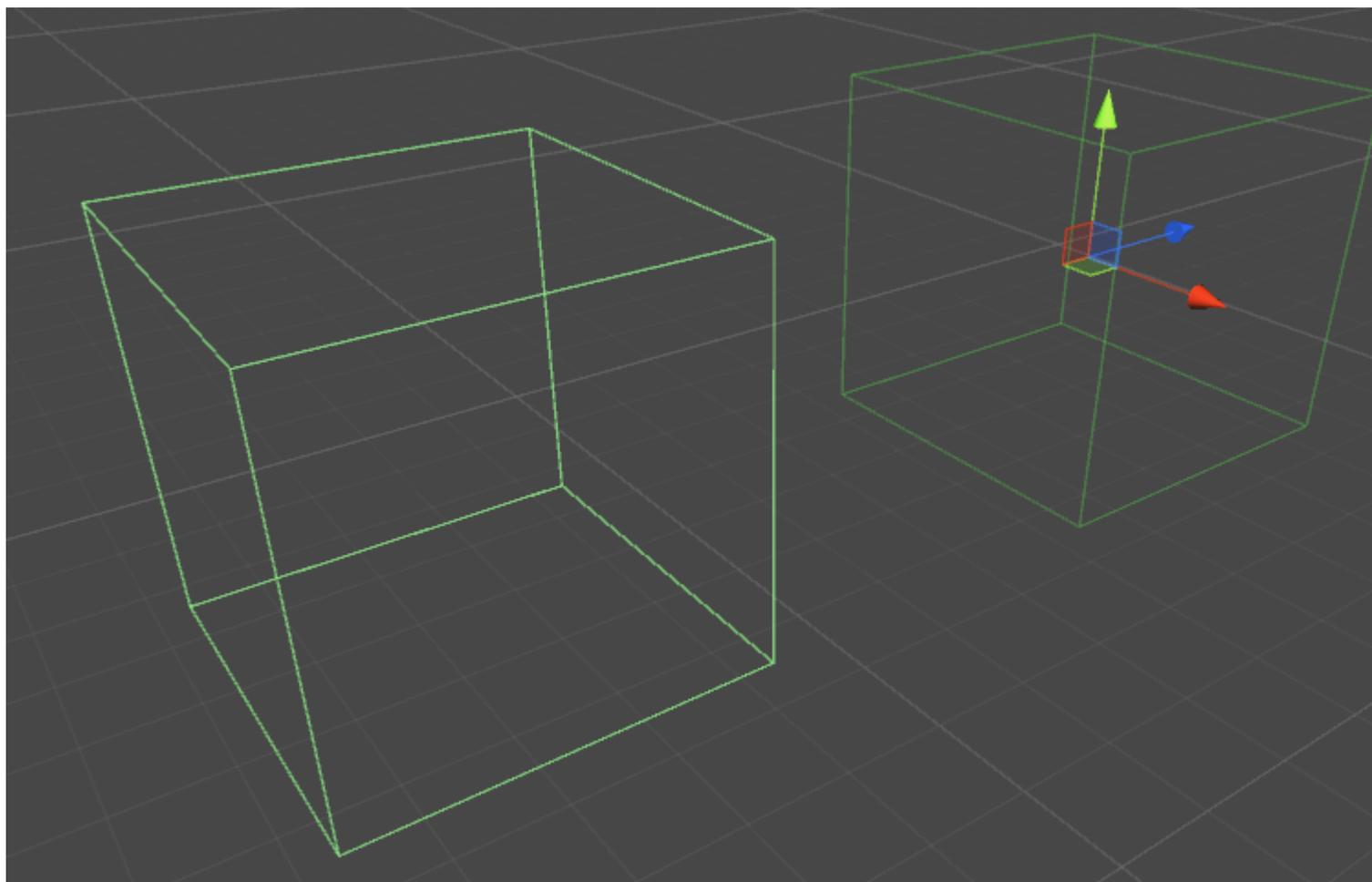
Capitolo 4: Collisione

Examples

collider

Box Collider

Un Collider primitivo a forma di cuboide.



Proprietà

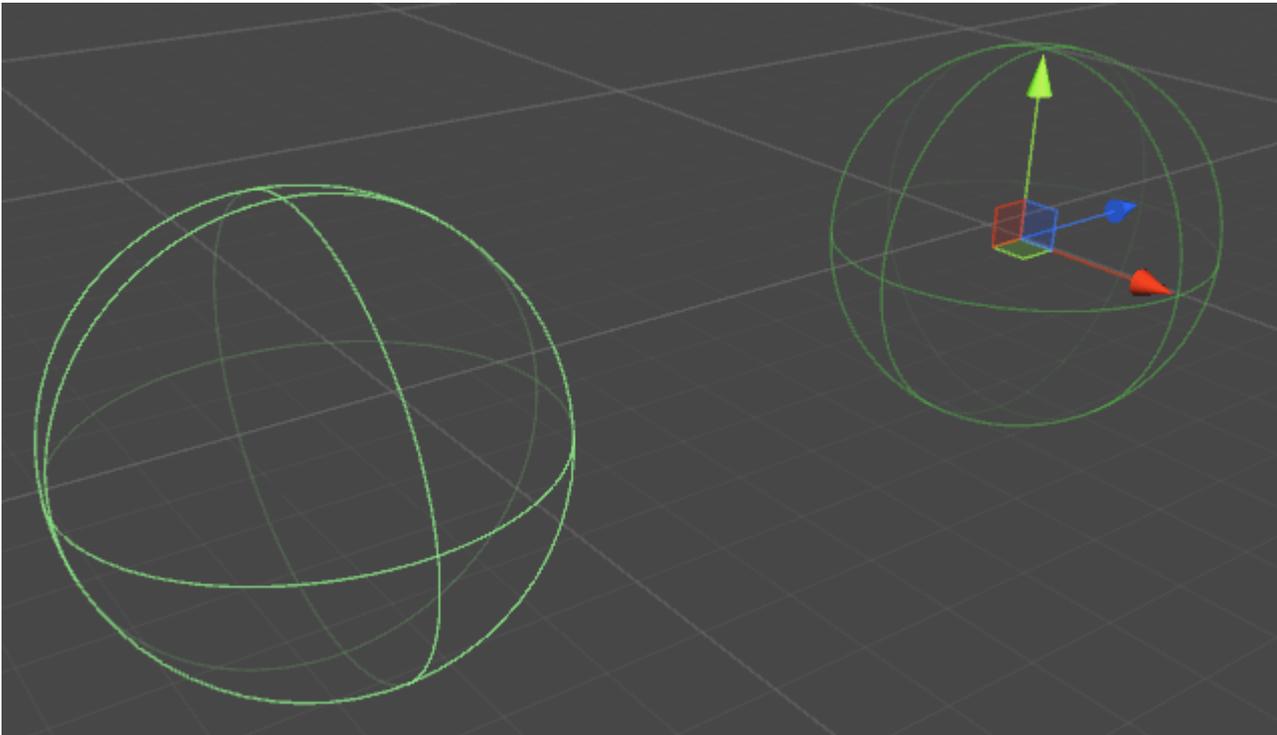
- **È Trigger** - Se selezionata, la Casella Collider ignorerà la fisica e diventare un trigger Collider
- **Materiale** : un riferimento, se specificato, al materiale fisico di Box Collider
- **Centro** : la posizione centrale di Box Collider nello spazio locale
- **Dimensione** : la dimensione del Box Collider misurata nello spazio locale

Esempio

```
// 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;
```

Sphere Collider

Un Collider primitivo a forma di sfera.



Proprietà

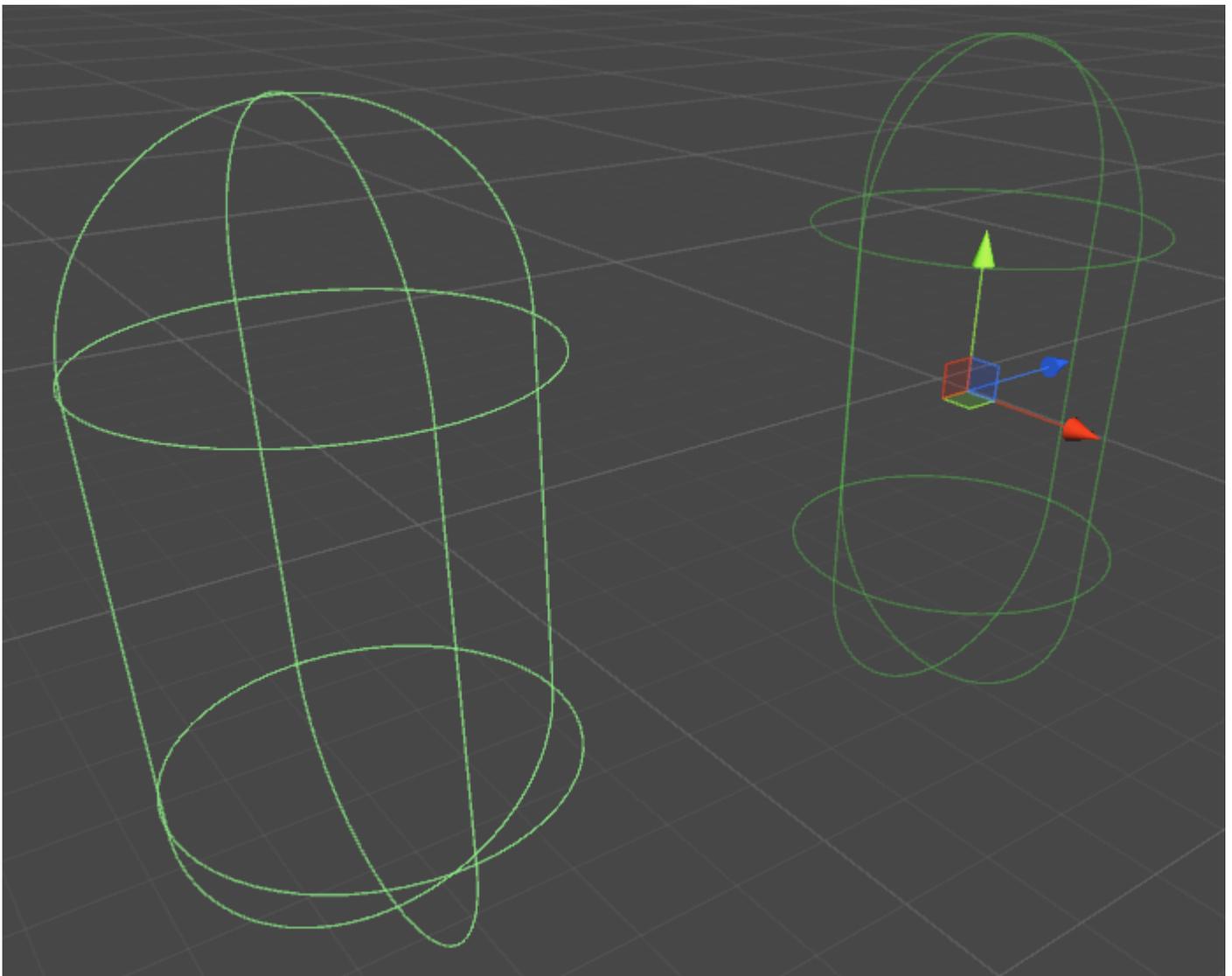
- **Trigger** : se spuntato, Sphere Collider ignorerà la fisica e diventerà un Trigger Collider
- **Materiale** : un riferimento, se specificato, al materiale fisico di Sphere Collider
- **Centro** : la posizione centrale di The Sphere Collider nello spazio locale
- **Raggio** : il raggio del Collider

Esempio

```
// 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;
```

Capsule Collider

Due mezze sfere unite da un cilindro.



Proprietà

- **Trigger** : se spuntato, Capsule Collider ignorerà la fisica e diventerà un Trigger Collider
- **Materiale** : un riferimento, se specificato, al materiale fisico del Capsule Collider
- **Centro** : la posizione centrale di Capsule Collider nello spazio locale
- **Raggio** - Il raggio nello spazio locale
- **Altezza** - **Altezza** totale del Collisore
- **Direzione** : l'asse di orientamento nello spazio locale

Esempio

```
// 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;
```

Wheel Collider

Proprietà

- **Massa** : la massa del Wheel Collider
- **Raggio** - Il raggio nello spazio locale
- **Velocità di smorzamento delle ruote** - Valore di smorzamento per il Wheel Collider

- **Distanza di sospensione** - Estensione massima lungo l'asse Y nello spazio locale
- **Forza la distanza del punto dell'app** - Il punto in cui verranno applicate le forze,
- **Centro** : centro del Wheel Collider nello spazio locale

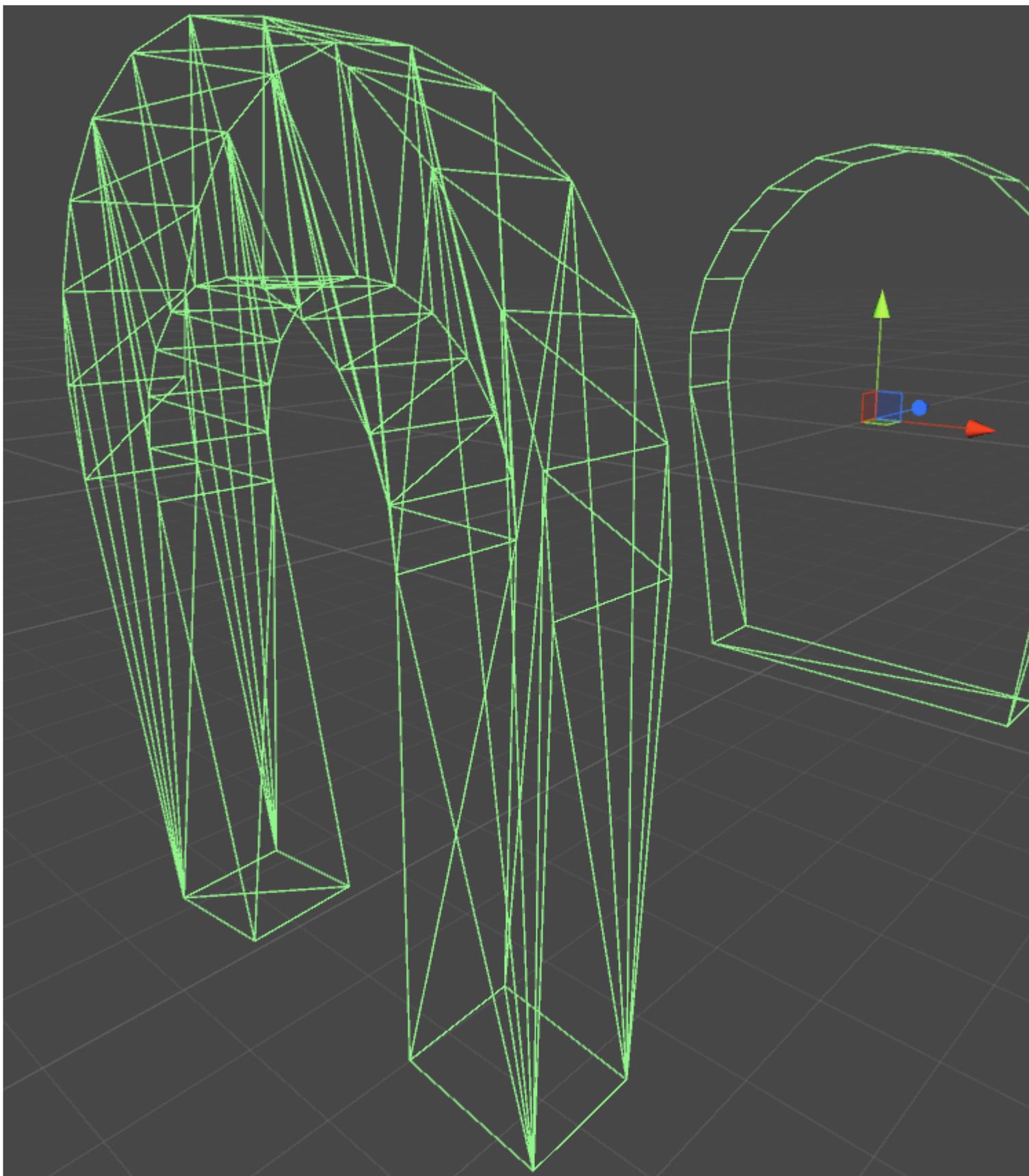
Sospensione Primavera

- **Spring** : la velocità con cui la ruota tenta di tornare alla posizione target
- **Damper** - Un valore maggiore smorza maggiormente la velocità e le sospensioni si muovono più lentamente
- **Posizione di destinazione** : il valore predefinito è 0,5, a 0 la sospensione viene portata a fondo, a 1 è all'estensione completa
- **Attrito avanti / lateralmente** - come si comporta il pneumatico quando si muove in avanti o lateralmente

Esempio

Mesh Collider

Un Collider basato su un Mesh Asset.



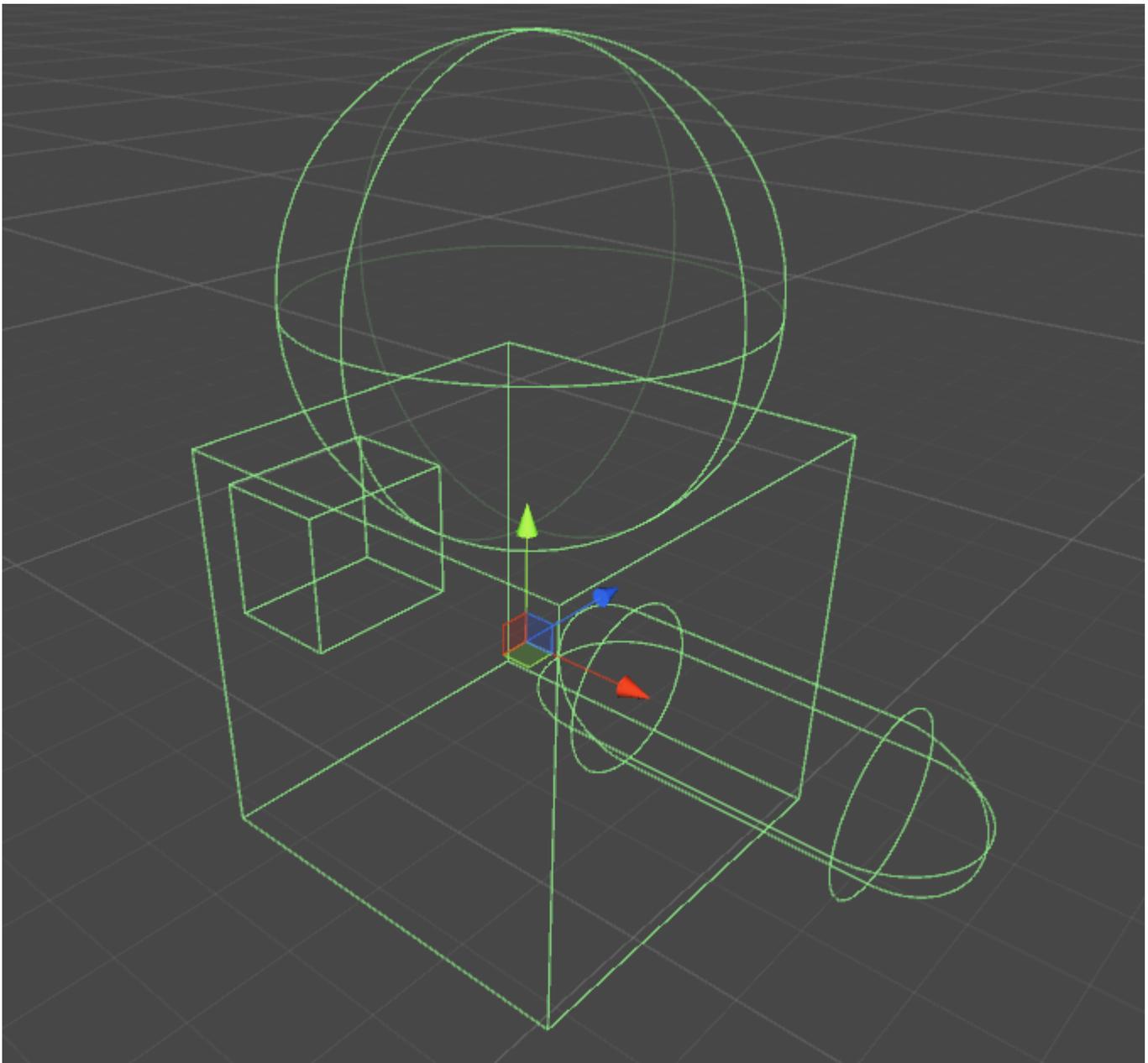
Proprietà

- **Trigger** : se spuntato, Box Collider ignorerà la fisica e diventerà un Trigger Collider
- **Materiale** : un riferimento, se specificato, al materiale fisico di Box Collider

- **Mesh** : un riferimento alla mesh su cui è basato il Collider
- **Convex** - I collettori di Convex Mesh sono limitati a 255 poligoni - se abilitato, questo Collider può scontrarsi con altri raccoglitori di mesh

Esempio

Se si applica più di un Collider a un GameObject, viene chiamato Compound Collider.



Wheel Collider

Il collisore all'interno dell'unità è basato sul collettore ruota PhysX di Nvidia e pertanto condivide molte proprietà simili. Tecnicamente l'unità è un programma "senza unità", ma per rendere tutto ha senso, sono necessarie alcune unità standard.

Proprietà di base

- Massa: il peso della ruota in chilogrammi, questo è usato per il momento della ruota e il momento dell'inerzia quando gira.
- Raggio: in metri, il raggio del collisore.
- Velocità di smorzamento delle ruote - Regola la modalità di risposta "reattiva" delle ruote alla coppia applicata.
- Distanza di sospensione - Distanza percorsa totale in metri che la ruota può percorrere
- Forza la distanza del punto dell'app - dove si trova la forza dalla sospensione applicata al corpo rigido genitore
- Centro: la posizione centrale della ruota

Impostazioni di sospensione

- Spring - Questa è la costante di primavera, K, in Newton / metro nell'equazione:

$$\text{Forza} = \text{costante} * \text{Distanza}$$

Un buon punto di partenza per questo valore dovrebbe essere la massa totale del tuo veicolo, diviso per il numero di ruote, moltiplicato per un numero compreso tra 50 e 100. Ad esempio, se hai una macchina da 2.000 kg con 4 ruote, allora ogni ruota dovrebbe supportare 500 kg. Moltiplicalo per 75, e la tua costante di primavera dovrebbe essere 37.500 Newton / metro.

- Ammortizzatore - l'equivalente di un ammortizzatore in una macchina. Tassi più alti rendono la sospensione "più rigida" e tassi più bassi lo rendono "più morbido" e più probabile che oscilli. Però non conosco le unità o l'equazione, penso che abbia a che fare con un'equazione di frequenza in fisica.

Impostazioni di attrito lateralmente

La curva di attrito in unità ha un valore di slittamento determinato da quanto la ruota sta scivolando (in m / s) dalla posizione desiderata rispetto alla posizione effettiva.

- Extremum Slip - Questa è la quantità massima (in m / s) di una ruota che può scivolare prima di perdere aderenza
- Valore estremo: è la quantità massima di attrito che deve essere applicata a una ruota.

I valori di Extremum Slip dovrebbero essere compresi tra 0,2 e 2 m / s per le auto più realistiche. 2 m / s è di circa 6 piedi al secondo o 5 miglia orarie, che è un sacco di scivolone. Se ritieni che il tuo veicolo debba avere un valore superiore a 2m / s per lo slittamento, dovresti considerare l'aumento dell'attrito massimo (valore Extremum).

Max Fraction (Extremum Value) è il coefficiente di attrito nell'equazione:

$$\text{Force of Friction (in newton)} = \text{Coefficient of Friction} * \text{Downward Force (in newton)}$$

Questo significa che con un coefficiente di 1, si applica l'intera forza dell'auto + sospensione opposta rispetto alla direzione di slittamento. Nelle applicazioni del mondo reale, valori superiori a 1 sono rari, ma non impossibili. Per un pneumatico su asfalto asciutto, i valori tra 0,7 e 1 sono realistici, quindi è preferibile il valore predefinito di 1,0.

Questo valore non dovrebbe realisticamente non superare il 2,5, poiché comincerà a verificarsi uno strano comportamento. Ad esempio, si inizia a girare a destra, ma poiché questo valore è così elevato, viene applicata una grande forza opposta alla propria direzione e si inizia a scivolare nel turno anziché lontano.

Se hai raggiunto il valore massimo di entrambi i valori, dovresti quindi iniziare ad aumentare lo slittamento e il valore dell'asymptote. Asymptote Slip dovrebbe essere compreso tra 0,5 e 2 m / se definisce il coefficiente di attrito per qualsiasi valore di slittamento oltre lo slittamento Asymptote. Se trovi che i tuoi veicoli si comportano bene fino a quando non rompono la trazione, a quel punto si comporta come se fosse su ghiaccio, dovresti aumentare il valore di Asymptote. Se trovi che il tuo veicolo non è in grado di andare alla deriva, dovresti abbassare il valore.

Attrito in avanti

L'attrito anteriore è identico all'attrito laterale, con l'eccezione che questo definisce quanta trazione la ruota ha nella direzione del movimento. Se i valori sono troppo bassi, i veicoli faranno burnouts e semplicemente ruotano le gomme prima di andare avanti, lentamente. Se è troppo alto, il tuo veicolo potrebbe avere la tendenza a provare a fare un wheely, o peggio, flip.

Note aggiuntive

Non aspettarti di essere in grado di creare un clone GTA o altri cloni da corsa semplicemente regolando questi valori. Nella maggior parte dei giochi di guida, questi valori vengono costantemente modificati nella sceneggiatura per diverse velocità, terreni e valori di virata. Inoltre, se si applica una coppia costante ai raccoglitori di ruote quando si preme un tasto, il gioco non si comporterà realisticamente. Nel mondo reale, le auto hanno curve di coppia e trasmissioni per cambiare la coppia applicata alle ruote.

Per ottenere i migliori risultati, è necessario sintonizzare questi valori fino a ottenere un'automobile che risponda abbastanza bene, quindi apportare modifiche alla coppia di ruote, all'angolo di sterzata massimo e ai valori di attrito nella sceneggiatura.

Maggiori informazioni sui carrelli a ruote possono essere trovate nella documentazione di Nvidia: <http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Vehicles.html>

Trigger Collider

metodi

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

È possibile creare un Collider in **Trigger** per utilizzare i `OnTriggerEnter()`, `OnTriggerStay()` e `OnTriggerExit()`. Un Trigger Collider non reagisce fisicamente alle collisioni, altri GameObject semplicemente lo attraversano. Sono utili per rilevare quando un altro GameObject si trova in una determinata area o no, ad esempio, quando si raccoglie un oggetto, potremmo essere in grado di scorrelo ma rilevare quando ciò accade.

Trigger Collider Scripting

Esempio

Il metodo seguente è un esempio di un listener di trigger che rileva quando un altro collisore entra nel collisore di un oggetto GameObject (come un giocatore). I metodi di trigger possono essere aggiunti a qualsiasi script assegnato a un GameObject.

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

Leggi Collisione online: <https://riptutorial.com/it/unity3d/topic/4405/collisione>

Capitolo 5: Come utilizzare i pacchetti di asset

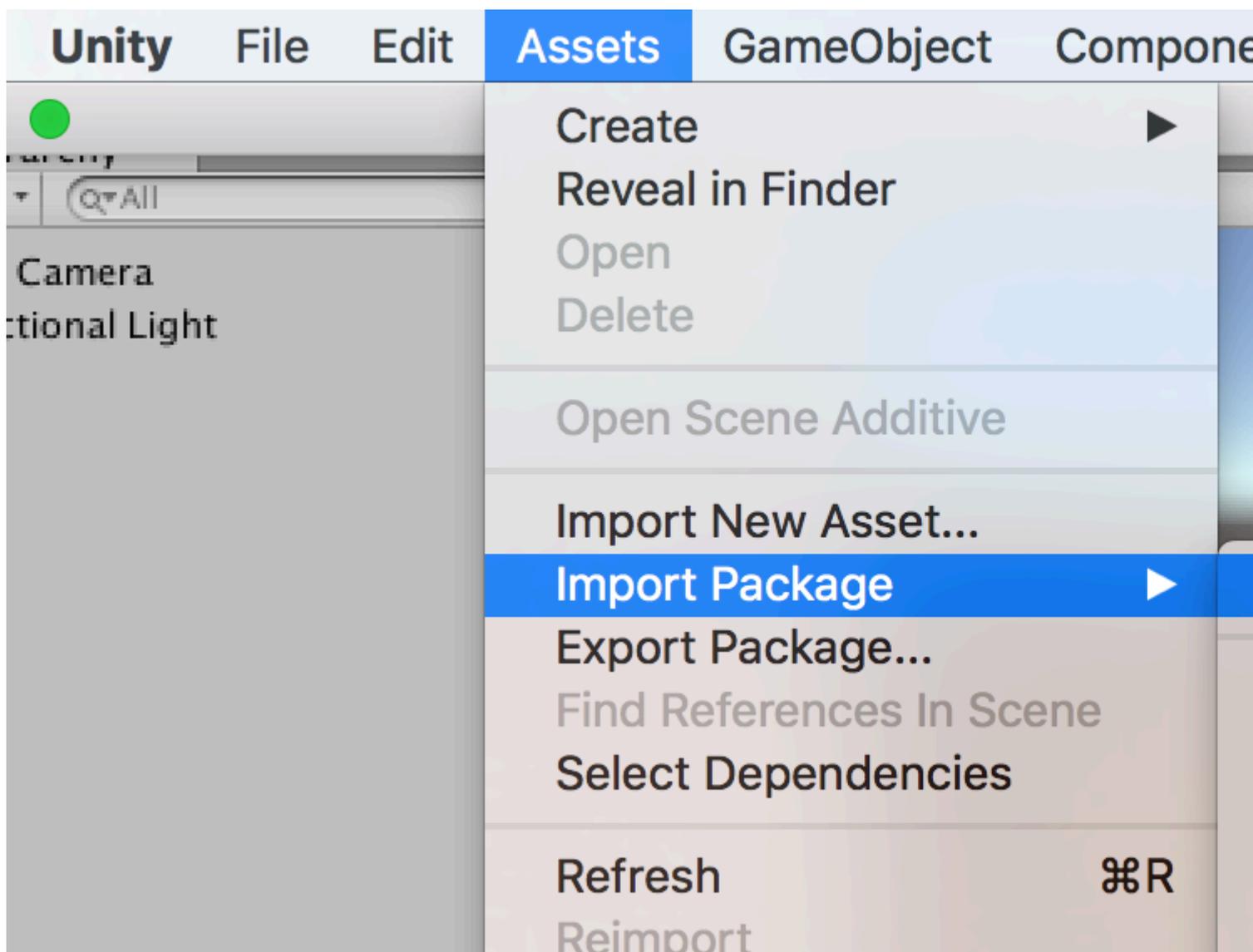
Examples

Pacchetti di attività

I **pacchetti di asset** (con il formato file di `.unitypackage`) sono un modo comunemente usato per distribuire i progetti Unity ad altri utenti. Quando si lavora con periferiche che hanno i propri SDK (ad esempio **Oculus**), è possibile che venga richiesto di scaricare e importare uno di questi pacchetti.

Importazione di un pacchetto .unity

Per importare un pacchetto, vai alla barra dei menu Unity e fai clic su `Assets > Import Package > Custom Package...`, quindi `.unitypackage` file `.unitypackage` nel Browser file visualizzato.



Leggi Come utilizzare i pacchetti di asset online: <https://riptutorial.com/it/unity3d/topic/4491/come-utilizzare-i-pacchetti-di-asset>

Capitolo 6: Comunicazione con il server

Examples

Ottenere

Ottenere sta ottenendo dati dal server web. `e new WWW("https://urlexample.com");` con un URL ma senza un secondo parametro sta facendo un **Get** .

vale a dire

```
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 semplice (campi postali)

Ogni istanza di **WWW** con un secondo parametro è un *post* .

Ecco un esempio per inviare *ID utente e password* al server.

```
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");
    }
}
```

Posta (Carica un file)

Carica un file sul server è anche un post. Puoi facilmente caricare un file tramite **WWW** , come il seguente:

Carica un file zip sul server

```
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!");
}
```

In questo esempio, usa la **coroutine** per preparare e caricare il file, se vuoi saperne di più sulle coroutine Unity, visita [Coroutines](#) .

Invio di una richiesta al server

Esistono molti modi per comunicare con i server utilizzando Unity come client (alcune metodologie sono migliori di altre a seconda del tuo scopo). Innanzitutto, è necessario determinare la necessità del server di essere in grado di inviare in modo efficace le operazioni da e verso il server. Per questo esempio, invieremo alcune parti di dati al nostro server per essere convalidate.

Molto probabilmente, il programmatore avrà impostato una sorta di gestore sul proprio server per ricevere gli eventi e rispondere di conseguenza al client, tuttavia ciò non rientra nell'ambito di questo esempio.

C #:

```
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;
    }
}
```

La prima cosa da fare è lanciare le loro istruzioni using che ci permettono di usare le classi WebClient e NameValueCollection.

Per questo esempio la funzione SendDataToServer accetta 3 parametri di stringa (facoltativi):

1. URL del server con cui comunichiamo
2. Primo pezzo di dati
3. Secondo pezzo di dati che stiamo inviando al server

Il nome utente e la password sono i dati opzionali che invio al server. Per questo esempio lo stiamo usando per essere poi ulteriormente convalidato da un database o da qualsiasi altra memoria esterna.

Ora che abbiamo impostato la nostra struttura, istanziamo un nuovo WebClient da utilizzare per inviare effettivamente i nostri dati. Ora dobbiamo caricare i nostri dati nel nostro NameValueCollection e caricare i dati sul server.

La funzione UploadValues accetta anche 3 parametri necessari:

1. Indirizzo IP del server
2. Metodo HTTP
3. Dati che stai inviando (nome utente e password nel nostro caso)

Questa funzione restituisce una matrice di byte della risposta dal server. Abbiamo bisogno di codificare il byte array restituito in una stringa corretta per poter essere in grado di manipolare e sezionare la risposta.

Si potrebbe fare qualcosa del genere:

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

Ora potresti ancora essere confuso, quindi suppongo che darò una breve spiegazione su come gestire un lato server di risposta.

Per questo esempio userò PHP per gestire la risposta dal client. Ti consiglio di utilizzare PHP come linguaggio di scripting back-end perché è super versatile, facile da usare e soprattutto veloce. Ci sono sicuramente altri modi per gestire una risposta su un server, ma a mio parere il PHP è di gran lunga la più semplice e semplice implementazione in 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";
}
```

Quindi questa è la parte più importante - l'"eco". Quando il nostro client carica i dati sul server, il client salva la risposta (o la risorsa) in quell'array di byte. Una volta che il cliente ha risposto, sai che i dati sono stati convalidati e puoi andare avanti nel client una volta che l'evento è successo. Devi anche pensare a quale tipo di dati stai inviando (in una certa misura) e come minimizzare l'importo che stai effettivamente inviando.

Quindi questo è solo un modo per inviare / ricevere dati da Unity - ci sono altri modi che potrebbero essere più efficaci per te in base al tuo progetto.

Leggi Comunicazione con il server online:

<https://riptutorial.com/it/unity3d/topic/5578/comunicazione-con-il-server>

Capitolo 7: coroutine

Sintassi

- Coroutine pubblica StartCoroutine (routine IEnumerator);
- public Coroutine StartCoroutine (string methodName, object value = null);
- public void StopCoroutine (string methodName);
- public void StopCoroutine (routine IEnumerator);
- pubblico vuoto StopAllCoroutines ();

Osservazioni

Considerazioni sulle prestazioni

È preferibile utilizzare le coroutine con moderazione in quanto la flessibilità si accompagna a un costo in termini di prestazioni.

- Le coroutine in gran numero richiedono più dalla CPU rispetto ai metodi di aggiornamento standard.
- Esiste un problema in alcune versioni di Unity in cui le coroutine producono immondizia a ogni ciclo di aggiornamento a causa della boxing Unity del valore restituito `MoveNext`. Questo è stato osservato per l'ultima volta in 5.4.0b13. ([Bug report](#))

Riduci la spazzatura memorizzando nella cache

YieldInstructions

Un trucco comune per ridurre la spazzatura generata nelle coroutine è memorizzare nella cache

`YieldInstruction`.

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

La resa `null` produce rifiuti inutili.

Examples

coroutine

Innanzitutto è essenziale capire che i motori di gioco (come Unity) lavorano su un paradigma "frame based".

Il codice viene eseguito durante ogni frame.

Ciò include il codice di Unity e il tuo codice.

Quando si pensa ai frame, è importante capire che non c'è **assolutamente** alcuna garanzia di quando i frame avvengono. **Non** succedono su un ritmo regolare. Gli spazi tra i frame potrebbero essere, ad esempio, 0,02632 quindi 0,021167 quindi 0,029778 e così via. Nell'esempio sono tutti "circa" 1/50 di secondo, ma sono tutti diversi. E in qualsiasi momento, puoi ottenere una cornice che richiede molto più tempo o più breve; e il tuo codice può essere eseguito in qualsiasi momento all'interno del frame.

Tenendolo a mente, potresti chiedere: come accedi a questi frame nel tuo codice, in Unity?

Molto semplicemente, si usa la chiamata Update () o si usa una coroutine. (In effetti - sono esattamente la stessa cosa: permettono al codice di essere eseguito su ogni fotogramma.)

Lo scopo di una coroutine è che:

è possibile eseguire un po 'di codice e quindi "fermarsi e attendere" **fino ad alcuni frame futuri** .

Puoi aspettare fino **al fotogramma successivo** , puoi aspettare **un numero di fotogrammi** , oppure puoi aspettare un po 'di tempo **approssimativo** in secondi in futuro.

Ad esempio, puoi aspettare "circa un secondo", il che significa che aspetterà per circa un secondo, e quindi metterà il tuo codice in qualche fotogramma all'incirca un secondo da ora. (E in effetti, all'interno di quel frame, il codice potrebbe essere eseguito in qualsiasi momento, in qualsiasi momento.) Per ripetere: non sarà esattamente un secondo. Il tempismo preciso non ha senso in un motore di gioco.

All'interno di una coroutine:

Per aspettare un frame:

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

Per aspettare tre frame:

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

Attendere **circa** mezzo secondo:

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

Fai qualcosa ogni singolo fotogramma:

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

Questo esempio è letteralmente identico al semplice inserimento di qualcosa all'interno della chiamata "Aggiornamento" di Unity: il codice di "fare qualcosa" viene eseguito su ogni frame.

Esempio

Allegare Ticker a un oggetto `GameObject`. Mentre quell'oggetto di gioco è attivo, verrà eseguito il segno di spunta. Si noti che lo script blocca con attenzione la coroutines, quando l'oggetto del gioco diventa inattivo; questo di solito è un aspetto importante dell'utilizzo corretto della coroutines.

```
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
        }
    }
}
```

Termina una coroutine

Spesso progetti le coroutines per terminare naturalmente quando vengono raggiunti determinati obiettivi.

```
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");
}
```

Per fermare una coroutine da "dentro" la coroutine, non puoi semplicemente "tornare" come se dovessi partire presto da una funzione ordinaria. Invece, usi la `yield break` .

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

Puoi anche forzare tutte le coroutine lanciate dallo script a fermarsi prima di finire.

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

Il metodo per interrompere una *specifica* coroutine dal chiamante varia a seconda di come è stato avviato.

Se hai avviato una coroutine per nome stringa:

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

quindi puoi fermarlo chiamando [StopCoroutine](#) con lo stesso nome stringa:

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

In alternativa, è possibile mantenere un riferimento *sia* alla `IEnumerator` restituito dal metodo `coroutine`, o l' `Coroutine` oggetto restituito da `StartCoroutine` , e chiamare `StopCoroutine` su uno di questi:

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

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

```

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

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

```

Metodi MonoBehaviour che possono essere Coroutine

Ci sono tre metodi MonoBehaviour che possono essere fatti in coroutine.

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

Questo può essere usato per creare, per esempio, script che vengono eseguiti solo quando l'oggetto è visibile a una telecamera.

```

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();
    }
}

```

Coroutine a catena

Le coroutine possono cedere dentro se stesse e attendere **altre coroutine** .

Quindi, puoi sequenze a catena - "una dopo l'altra".

Questo è molto semplice, ed è una tecnica di base, fondamentale in Unity.

È assolutamente naturale nei giochi che certe cose debbano accadere "in ordine". Quasi ogni "round" di un gioco inizia con una serie di eventi che avvengono, in un certo lasso di tempo, in un certo ordine. Ecco come potresti iniziare un gioco di corse automobilistiche:

```
IEnumerator BeginRace()
```

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

Quindi, quando chiami `BeginRace` ...

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

Gestirà la tua routine di "preparazione alla gara". (Forse, facendo lampeggiare un po' di luci e facendo rumori di folla, azzerando i punteggi e così via.) Quando avrà finito, eseguirà la sequenza "conto alla rovescia", dove animeresti forse un conto alla rovescia sull'interfaccia utente. Al termine, verrà eseguito il codice di partenza della gara, in cui potresti eseguire effetti sonori, avviare alcuni driver AI, spostare la videocamera in un determinato modo e così via.

Per chiarezza, capire che le tre chiamate

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

devono **essere in una coroutine**. Vale a dire, devono essere in una funzione del tipo `IEnumerator`. Quindi nel nostro esempio è `IEnumerator BeginRace`. Quindi, dal codice "normale", si avvia quella coroutine con la chiamata `StartCoroutine`.

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

Per comprendere meglio il concatenamento, ecco una funzione che incatena le coroutines. Passi in una serie di coroutines. La funzione esegue tutte le coroutines che si passano, in ordine, una dopo l'altra.

```
// 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;
}
```

Ecco come lo chiameresti ... diciamo che hai tre funzioni. Ricordiamo che devono essere tutti

`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 chiameresti in questo modo

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

o forse così

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

Per ripetere, uno dei requisiti di base dei giochi è che certe cose accadono una dopo l'altra "in una sequenza" nel tempo. Lo ottieni in Unity molto semplicemente, con

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

Modi per cedere

Puoi aspettare fino al fotogramma successivo.

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

È possibile avere più di queste chiamate in fila, per attendere semplicemente il numero di frame desiderato.

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

Attendere **circa** n secondi. È estremamente importante capire che questo è solo **un tempo molto approssimativo** .

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

Non è assolutamente possibile utilizzare la chiamata "WaitForSeconds" per qualsiasi forma di

sincronizzazione accurata.

Spesso vuoi concatenare azioni. Quindi, fai qualcosa, e quando questo è finito fai qualcos'altro, e quando questo è finito fai qualcos'altro. Per riuscirci, aspetta un'altra coroutine:

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

Comprendi che puoi chiamarlo solo da una coroutine. Così:

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

È così che si avvia una coroutine da un pezzo di codice "normale".

Quindi, all'interno di quella coroutine in esecuzione:

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

Ciò stamperà A, avvia il processo lungo e **stampa immediatamente B**. Non sarà attendere che il processo lungo per terminare. D'altro canto:

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

Ciò stamperà A, avvierà il processo lungo, **attenderà fino a quando non sarà finito**, e quindi stamperà B.

Vale sempre la pena ricordare che le coroutine non hanno assolutamente alcuna connessione, in alcun modo, alla discussione. Con questo codice:

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

è facile pensarlo come "simile" all'avvio di LongProcess su un altro thread in background. Ma questo è assolutamente scorretto. È solo una coroutine. I motori di gioco sono basati su frame e "coroutines" in Unity ti permettono semplicemente di accedere ai frame.

È molto semplice aspettare che una richiesta web venga completata.

```
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);  
}  
}
```

Per completezza: in casi molto rari si utilizza l'aggiornamento fisso in Unity; c'è una chiamata `WaitForFixedUpdate()` che normalmente non verrebbe mai utilizzata. Esiste una chiamata specifica (`WaitForEndOfFrame()` nella versione corrente di Unity) che viene utilizzata in determinate situazioni in relazione alla generazione di acquisizioni dello schermo durante lo sviluppo. (Il meccanismo esatto cambia leggermente con l'evoluzione di Unity, quindi google per le ultime informazioni se pertinenti).

Leggi coroutine online: <https://riptutorial.com/it/unity3d/topic/3415/coroutine>

Capitolo 8: Estendere l'editor

Sintassi

- [MenuItem (stringa itemName)]
- [MenuItem (stringa itemName, bool isValidFunction)]
- [MenuItem (stringa itemName, bool isValidFunction, int priority)]
- [ContextMenu (nome stringa)]
- [ContextMenu (nome stringa, funzione stringa)]
- [DrawGizmo (GizmoType gizmo)]
- [DrawGizmo (GizmoType gizmo, Type drawnGizmoType)]

Parametri

Parametro	Dettagli
MenuCommand	MenuCommand viene utilizzato per estrarre il contesto per un oggetto Menu
MenuCommand.context	L'oggetto che è la destinazione del comando di menu
MenuCommand.userData	Un int per il passaggio di informazioni personalizzate a una voce di menu

Examples

Ispettore doganale

L'uso di un ispettore personalizzato ti consente di cambiare il modo in cui viene disegnato uno script in Inspector. A volte si desidera aggiungere ulteriori informazioni nell'ispettore per il proprio script che non è possibile fare con un cassetto delle proprietà personalizzate.

Di seguito è riportato un semplice esempio di oggetto personalizzato che con l'utilizzo di un'ispettore personalizzato può mostrare più informazioni utili.

```
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

```

Per prima cosa definiamo il nostro comportamento personalizzato con alcuni campi

```

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

```

I campi sopra mostrati vengono disegnati automaticamente (senza un ispettore personalizzato) quando stai visualizzando lo script nella finestra dell'Inspector.

```

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

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

```

Queste proprietà non vengono automaticamente disegnate da Unity. Per mostrare queste proprietà nella vista Inspector dobbiamo usare il nostro Custom Inspector.

Per prima cosa dobbiamo definire il nostro ispettore personalizzato come questo

```

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

```

L'ispettore personalizzato deve derivare da *Editor* e necessita dell'attributo *CustomEditor* . Il parametro dell'attributo è il tipo di oggetto per cui deve essere usato questo ispettore personalizzato.

Il prossimo è il metodo `OnInspectorGUI`. Questo metodo viene chiamato ogni volta che lo script viene mostrato nella finestra di ispezione.

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

Effettuiamo una chiamata a `base.OnInspectorGUI()` per consentire a Unity di gestire gli altri campi presenti nello script. Se non lo chiamassimo dovremmo fare più lavoro da soli.

Poi ci sono le nostre proprietà personalizzate che vogliamo mostrare

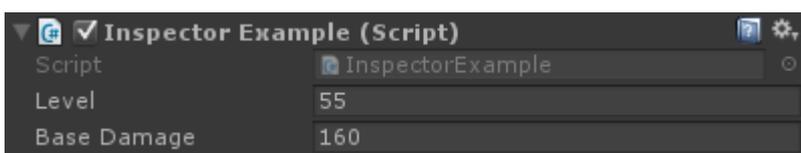
```
var ie = (InspectorExample)target;  
  
EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );  
EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
```

Dobbiamo creare una variabile temporanea che tratti il target castato nel nostro tipo personalizzato (l'obiettivo è disponibile perché deriviamo dall'*Editor*).

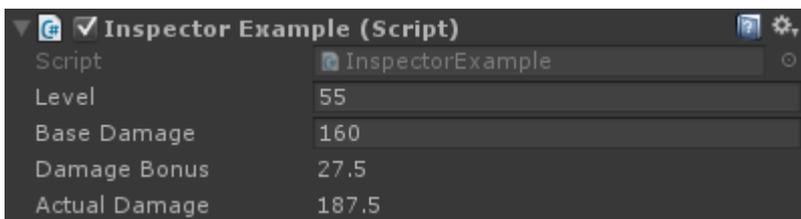
Successivamente possiamo decidere come disegnare le nostre proprietà, in questo caso sono sufficienti due etichette di riferimento, poiché vogliamo solo mostrare i valori e non essere in grado di modificarli.

Risultato

Prima



Dopo



Cassetto delle proprietà personalizzate

A volte hai oggetti personalizzati che contengono dati ma che non derivano da *MonoBehaviour*. L'aggiunta di questi oggetti come campo in una classe che è *MonoBehaviour* non avrà alcun effetto visivo a meno che non si scriva il proprio cassetto delle proprietà personalizzate per il tipo

dell'oggetto.

Di seguito è riportato un semplice esempio di oggetto personalizzato, aggiunto a `MonoBehaviour` e un cassetto delle proprietà personalizzate per l'oggetto personalizzato.

```
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();
    }
}
```

Per prima cosa definiamo l'oggetto personalizzato con tutti i suoi requisiti. Solo una semplice classe che descrive un utente. Questa classe viene utilizzata nella nostra classe `PropertyDrawerExample` che è possibile aggiungere a un oggetto `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 classe personalizzata necessita dell'attributo `Serializable`, altrimenti `CustomPropertyDrawer` non verrà utilizzato

Il prossimo è `CustomPropertyDrawer`

Per prima cosa dobbiamo definire una classe che deriva da `PropertyDrawer`. La definizione della classe richiede anche l'attributo `CustomPropertyDrawer`. Il parametro passato è il tipo di oggetto per il quale si desidera utilizzare questo cassetto.

```

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

```

Quindi sostituiamo la funzione `GetPropertyHeight`. Questo ci consente di definire un'altezza personalizzata per la nostra proprietà. In questo caso sappiamo che la nostra proprietà avrà quattro parti: etichetta, nome, età e sesso. Quindi usiamo `EditorGUIUtility.singleLineHeight * 4`, aggiungiamo altri 6 pixel perché vogliamo distanziare ogni campo con due pixel in mezzo.

```

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

```

Il prossimo è il metodo `OnGUI` effettivo. Iniziamo con `EditorGUI.BeginProperty ([...])` e terminiamo la funzione con `EditorGUI.EndProperty ()`. Facciamo questo in modo che se questa proprietà fosse parte di un prefabbricato, l'effettiva logica di sostituzione prefabbricata funzionerebbe per tutto ciò che si trova tra questi due metodi.

```

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

```

Dopodiché mostriamo un'etichetta contenente il nome del campo e già definiamo i rettangoli per i nostri campi.

```

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 );
```

Ogni campo è distanziato di $16 + 2$ pixel e l'altezza è 16 (che è uguale a `EditorGUIUtility.singleLineHeight`)

Successivamente indentiamo l'interfaccia utente con una scheda per un layout un po' più bello, visualizziamo le proprietà, *annulliamo la rientranza* della GUI e *terminiamo* 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();
```

Visualizziamo i campi utilizzando `EditorGUI.PropertyField` che richiede un rettangolo per la posizione e una proprietà serializzata per la proprietà da mostrare. *Acquisiamo* la proprietà chiamando `FindPropertyRelative` ("...") sulla proprietà passata nella funzione `OnGUI`. Nota che queste sono case-sensitive e proprietà non pubbliche non possono essere trovate!

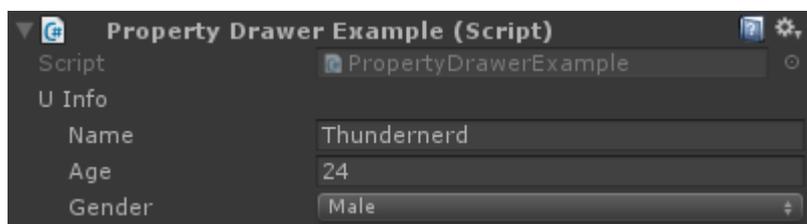
Per questo esempio non sto salvando il ritorno delle proprietà da `property.FindPropertyRelative` ("..."). Dovresti salvare questi in campi privati della classe per evitare chiamate inutili

Risultato

Prima



Dopo



Voci del menu

Le voci di menu sono un ottimo modo per aggiungere azioni personalizzate all'editor. È possibile aggiungere voci di menu alla barra dei menu, averli come clic-contesto su componenti specifici o anche come clic-clic su campi nei propri script.

Di seguito è riportato un esempio di come è possibile applicare le voci di menu.

```

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();
    }
}

```

Che assomiglia a questo

Example	Window	Help
DoOne		Ctrl+A
DoTwo		Shift+B
DoFurther		Alt+C
DoSomething	Ctrl+Shift+Alt+D	
DoAnotherThing		PgUp

Andiamo oltre la voce di menu di base. Come puoi vedere di seguito, devi definire una funzione statica con un attributo *MenuItem*, che ti consente di passare una stringa come titolo per la voce di menu. Puoi mettere la tua voce di menu più livelli in profondità aggiungendo un / nel nome.

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

Non puoi avere una voce di menu al primo livello. Le voci del tuo menu devono essere in un sottomenu!

I caratteri speciali alla fine del nome di MenuItem sono per i tasti di scelta rapida, questi non sono un requisito.

Ci sono caratteri speciali che puoi usare per i tuoi tasti di scelta rapida, questi sono:

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

Ciò significa che la scorciatoia % # & d sta per ctrl + shift + alt + D su Windows e cmd + shift + alt + D su OS X.

Se si desidera utilizzare una scorciatoia senza tasti speciali, ad esempio solo il tasto 'D', è possibile anteporre il carattere _ (carattere di sottolineatura) al tasto di scelta rapida che si desidera utilizzare.

Esistono altre chiavi speciali supportate, che sono:

- SINISTRA, DESTRA, SU, GIÙ - per i tasti freccia
- F1..F12 - per i tasti funzione
- HOME, END, PGUP, PGDN - per i tasti di navigazione

I tasti di scelta rapida devono essere separati da qualsiasi altro testo con uno spazio

Successivamente ci sono voci di menu del validator. Le voci del menu Validator consentono di disabilitare le voci di menu (disattivate, non selezionabili) quando la condizione non viene soddisfatta. Un esempio potrebbe essere che la voce del menu agisce sulla selezione corrente di GameObjects, che è possibile verificare nella voce di menu del validator.

```
[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
}
```

Perché una voce di menu del validatore funzioni è necessario creare due funzioni statiche, entrambe con l'attributo MenuItem e lo stesso nome (il tasto di scelta rapida non ha importanza). La differenza tra loro è che li stai contrassegnando come una funzione di validazione o meno passando un parametro booleano.

È inoltre possibile definire l'ordine delle voci di menu aggiungendo una priorità. La priorità è definita da un intero che si passa come terzo parametro. Più piccolo è il numero più in alto nell'elenco, più grande è il numero più in basso nell'elenco. È possibile aggiungere un separatore tra due voci di menu assicurandosi che vi siano almeno 10 cifre tra la priorità delle voci di menu.

```
[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
}
```

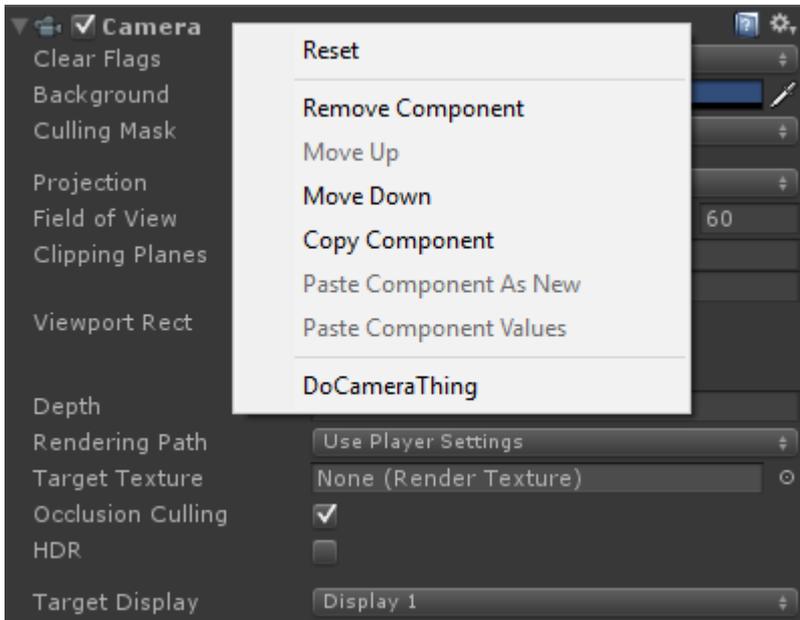
Se si dispone di un elenco di menu che ha una combinazione di elementi prioritari e non prioritari, i non prioritari verranno separati dagli elementi prioritari.

Il prossimo è aggiungere una voce di menu al menu di scelta rapida di un componente già esistente. È necessario avviare il nome di MenuItem con CONTEXT (con distinzione tra maiuscole e minuscole) e impostare la funzione in un parametro MenuCommand.

Il seguente frammento aggiungerà una voce di menu contestuale al componente Fotocamera.

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

Che assomiglia a questo

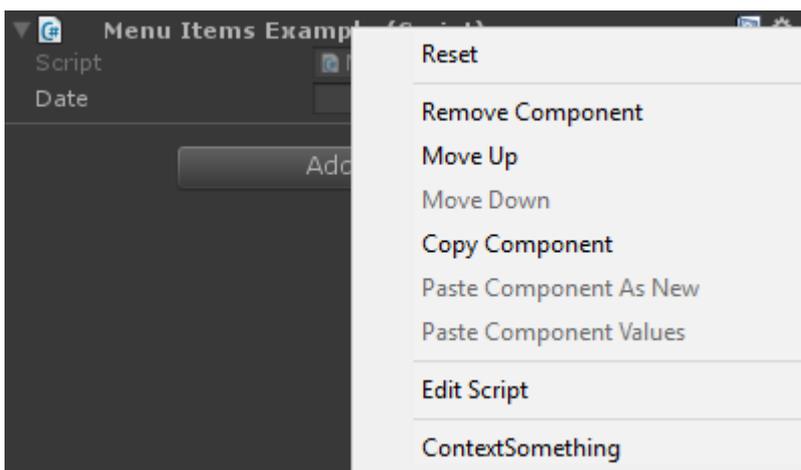


Il parametro MenuCommand consente di accedere al valore del componente ea tutti i dati utente che vengono inviati con esso.

Puoi anche aggiungere una voce di menu contestuale ai tuoi componenti utilizzando l'attributo ContextMenu. Questo attributo richiede solo un nome, nessuna convalida o priorità e deve essere parte di un metodo non statico.

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

Che assomiglia a questo



È inoltre possibile aggiungere voci del menu di scelta rapida ai campi nel proprio componente. Queste voci di menu vengono visualizzate quando fai clic con il pulsante destro del mouse sul campo a cui appartengono e puoi eseguire i metodi che hai definito in quel componente. In questo modo è possibile aggiungere, ad esempio, valori predefiniti o la data corrente, come mostrato di seguito.

```

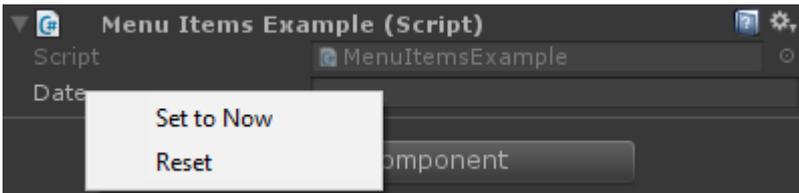
[ContextMenuItem( "Reset", "ResetDate" )]
[ContextMenuItem( "Set to Now", "SetDateToNow" )]
public string Date = "";

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

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

```

Che assomiglia a questo



aggeggi

I gizmo sono usati per disegnare forme nella vista scena. Puoi utilizzare queste forme per ottenere informazioni aggiuntive sui tuoi oggetti GameObjects, ad esempio il trofeo che hanno o il raggio di rilevamento.

Di seguito sono riportati due esempi su come farlo

Esempio 1

Questo esempio utilizza i *metodi* (magici) *OnDrawGizmos* e *OnDrawGizmosSelected*.

```

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() );
}
}

```

In questo esempio abbiamo due metodi per disegnare i gizmo, uno che disegna quando l'oggetto è attivo (`OnDrawGizmos`) e uno per quando l'oggetto è selezionato nella gerarchia (`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;
}

```

Per prima cosa salviamo la matrice gizmo e il colore perché lo cambieremo e vogliamo ripristinarlo quando avremo finito per non influenzare nessun altro disegno gizmo.

Poi vogliamo disegnare il trofeo che il nostro oggetto ha, tuttavia, abbiamo bisogno di cambiare la matrice dei Gizmos in modo che corrisponda alla posizione, alla rotazione e alla scala. Abbiamo anche impostato il colore dei Gizmos in rosso per enfatizzare il trofeo. Quando questo è fatto, possiamo chiamare `Gizmos.DrawFrustum` per disegnare il troncone nella scena.

Quando abbiamo finito di disegnare ciò che vogliamo disegnare, ripristiniamo la matrice e il colore dei Gizmos.

```

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

```

Vogliamo anche disegnare un intervallo di rilevamento quando selezioniamo il nostro `GameObject`. Questo viene fatto attraverso la classe `Handles` poiché la classe `Gizmos` non ha alcun metodo per i dischi.

L'utilizzo di questa forma di disegno di gizmo risulta nell'output mostrato di seguito.

Esempio due

Questo esempio utilizza l'attributo *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()
);
        }
    }
}
```

In questo modo puoi separare le chiamate gizmo dal tuo script. La maggior parte di questo utilizza lo stesso codice dell'altro esempio ad eccezione di due cose.

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

È necessario utilizzare l'attributo *DrawGizmo* che prende l'enumerazione *GizmoType* come primo parametro e un tipo come secondo parametro. Il tipo dovrebbe essere il tipo che si desidera utilizzare per disegnare il gizmo.

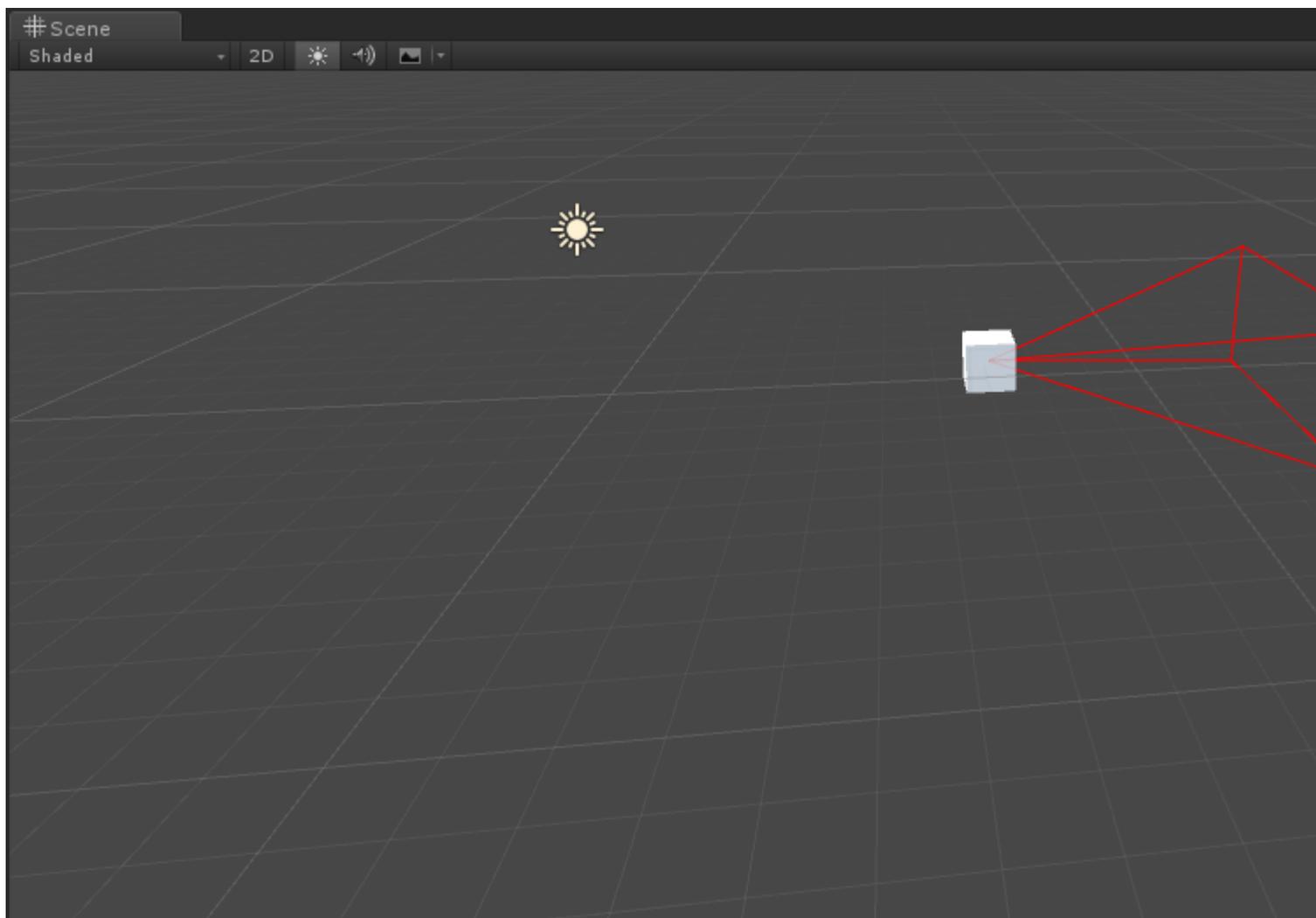
Il metodo per disegnare il gizmo deve essere statico, pubblico o non pubblico e può essere chiamato come vuoi. Il primo parametro è il tipo, che deve corrispondere al tipo passato come secondo parametro nell'attributo e il secondo parametro è l'enum *GizmoType* che descrive lo stato corrente dell'oggetto.

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

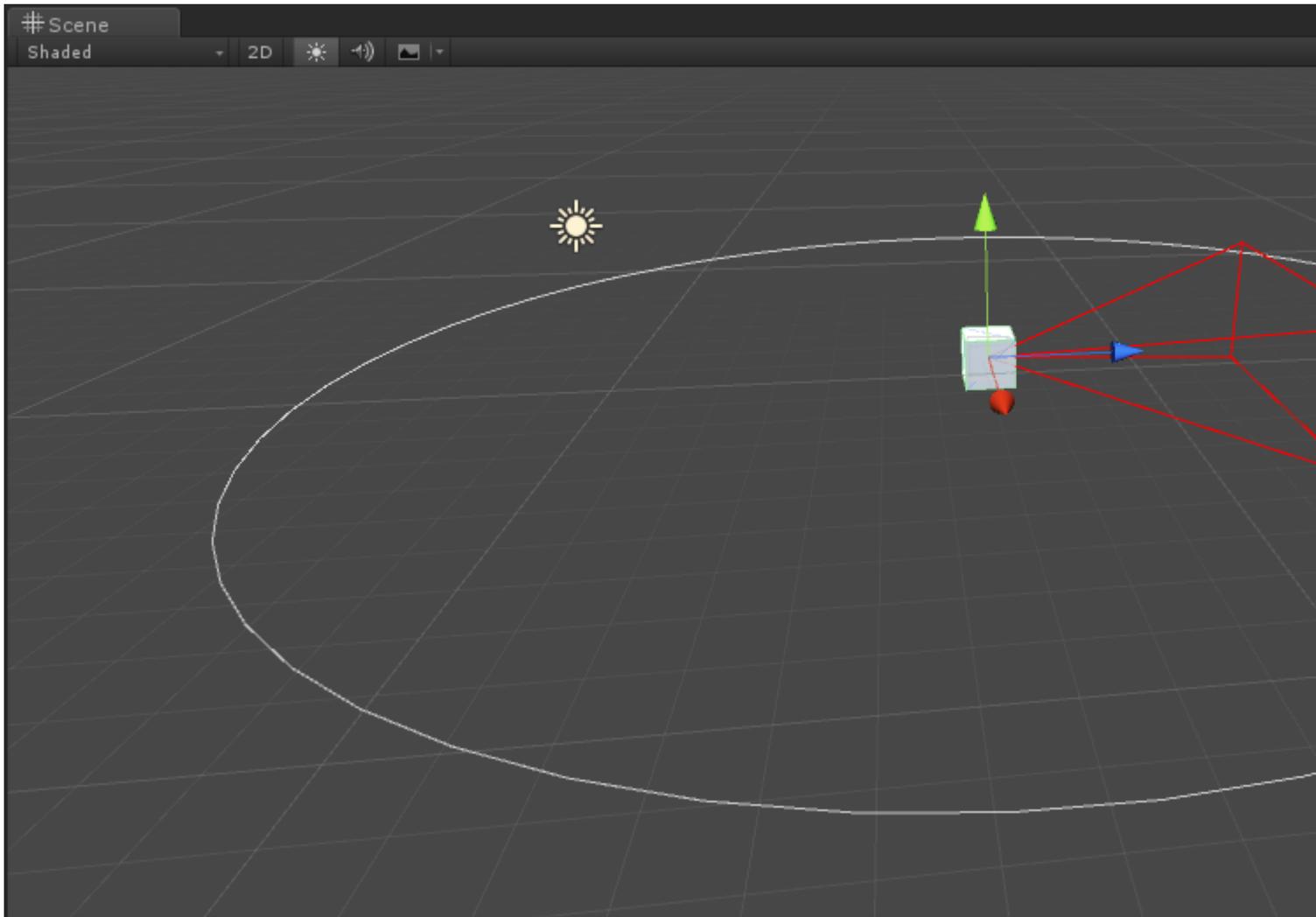
L'altra differenza è che per verificare qual è il *GizmoType* dell'oggetto, è necessario eseguire un controllo AND sul parametro e sul tipo desiderato.

Risultato

Non selezionato



Selezionato



Finestra dell'editor

Perché una finestra dell'editor?

Come potresti aver visto, puoi fare un sacco di cose in un ispettore personalizzato (se non sai cos'è un ispettore personalizzato, controlla l'esempio qui: <http://www.Scriptutorial.com/unity3d/topic/2506/estendere-l-editor>, ma a un certo punto potresti voler implementare un pannello di configurazione o una tavolozza delle risorse personalizzata, in questi casi utilizzerai una **finestra Editor**. L'interfaccia utente di Unity stessa è composta da Windows Editor, puoi aprirli (di solito attraverso la barra in alto), scheda, ecc.

Crea una finestra Editor di base

Semplice esempio

La creazione di una finestra di editor personalizzato è abbastanza semplice. Tutto quello che devi fare è estendere la classe `EditorWindow` e utilizzare i metodi `Init()` e `OnGUI()`. Qui c'è un semplice esempio :

```
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();
    }

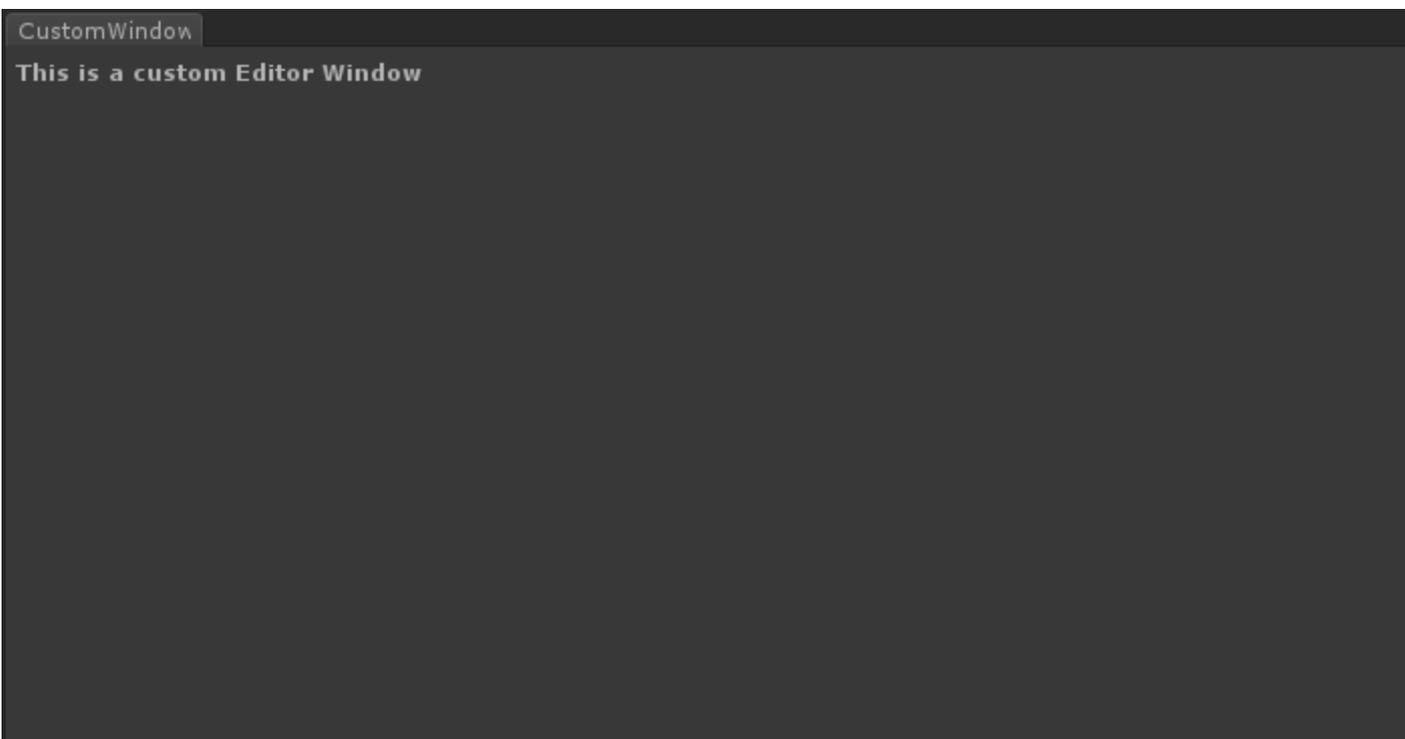
    void OnGUI()
    {
        GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);
    }
}

```

I 3 punti importanti sono:

1. Non dimenticare di estendere `EditorWindow`
2. Utilizzare `Init ()` come previsto nell'esempio. `EditorWindow.GetWindow` sta controllando se una `CustomWindow` è già stata creata. In caso contrario, creerà una nuova istanza. Usando questo ti assicuri di non avere più istanze della tua finestra allo stesso tempo
3. Usa `OnGUI ()` come al solito per visualizzare le informazioni nella tua finestra

Il risultato finale sarà simile a questo:



CustomWindow
This is a custom Editor Window

Andando più a fondo

Ovviamente probabilmente vorrai gestire o modificare alcune risorse usando questa `EditorWindow`. Ecco un esempio che utilizza la classe `Selection` (per ottenere la selezione attiva) e

modifica delle proprietà delle risorse selezionate tramite [SerializedObject](#) e [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();
}

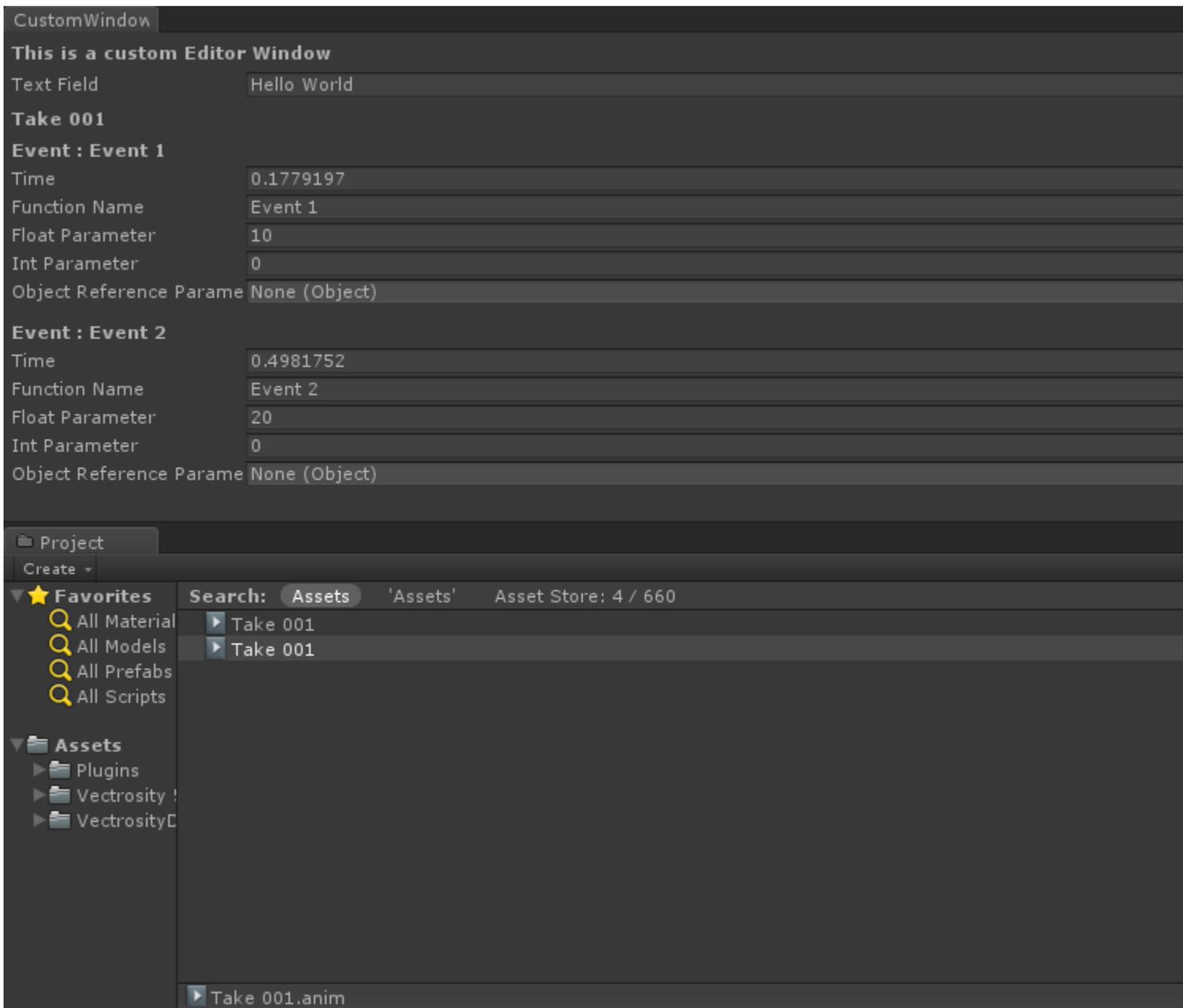
// Of course we need to Apply the modified properties. We don't our changes won't
be saved
_serializedClip.ApplyModifiedProperties();
}

/// 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();
}
}

```

Ecco il risultato:



Argomenti avanzati

Nell'editor puoi fare alcune cose davvero avanzate e la classe `EditorWindow` è perfetta per la visualizzazione di grandi quantità di informazioni. Le risorse più avanzate di Unity Asset Store (come `NodeCanvas` o `PlayMaker`) utilizzano `EditorWindow` per la visualizzazione per visualizzazioni personalizzate.

Disegnare in `SceneView`

Una cosa interessante da fare con `EditorWindow` è visualizzare le informazioni direttamente in `SceneView`. In questo modo è possibile creare un editor di mappe / mondo completamente personalizzato, ad esempio, utilizzando l'`EditorWindow` personalizzato come una tavolozza delle risorse e ascoltando i clic in `SceneView` per creare un'istanza di nuovi oggetti. Ecco un esempio:

```
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();
    }

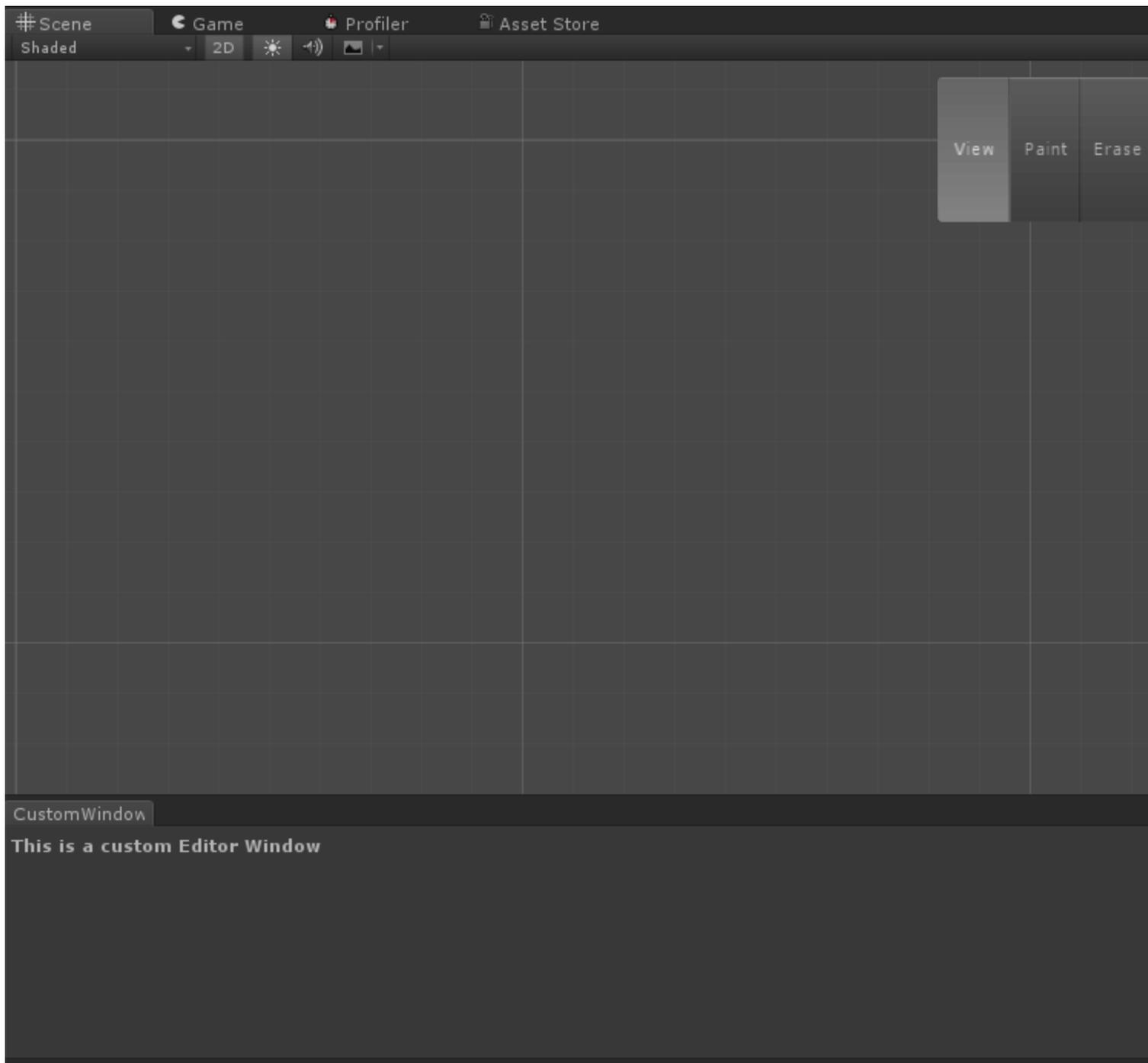
    void OnGUI () {
        GUILayout.Label ("This is a custom Editor Window", EditorStyles.boldLabel);
    }

    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();
    }
}

```

Questo mostrerà la barra degli strumenti direttamente in SceneView



Ecco una rapida panoramica di quanto lontano puoi andare:

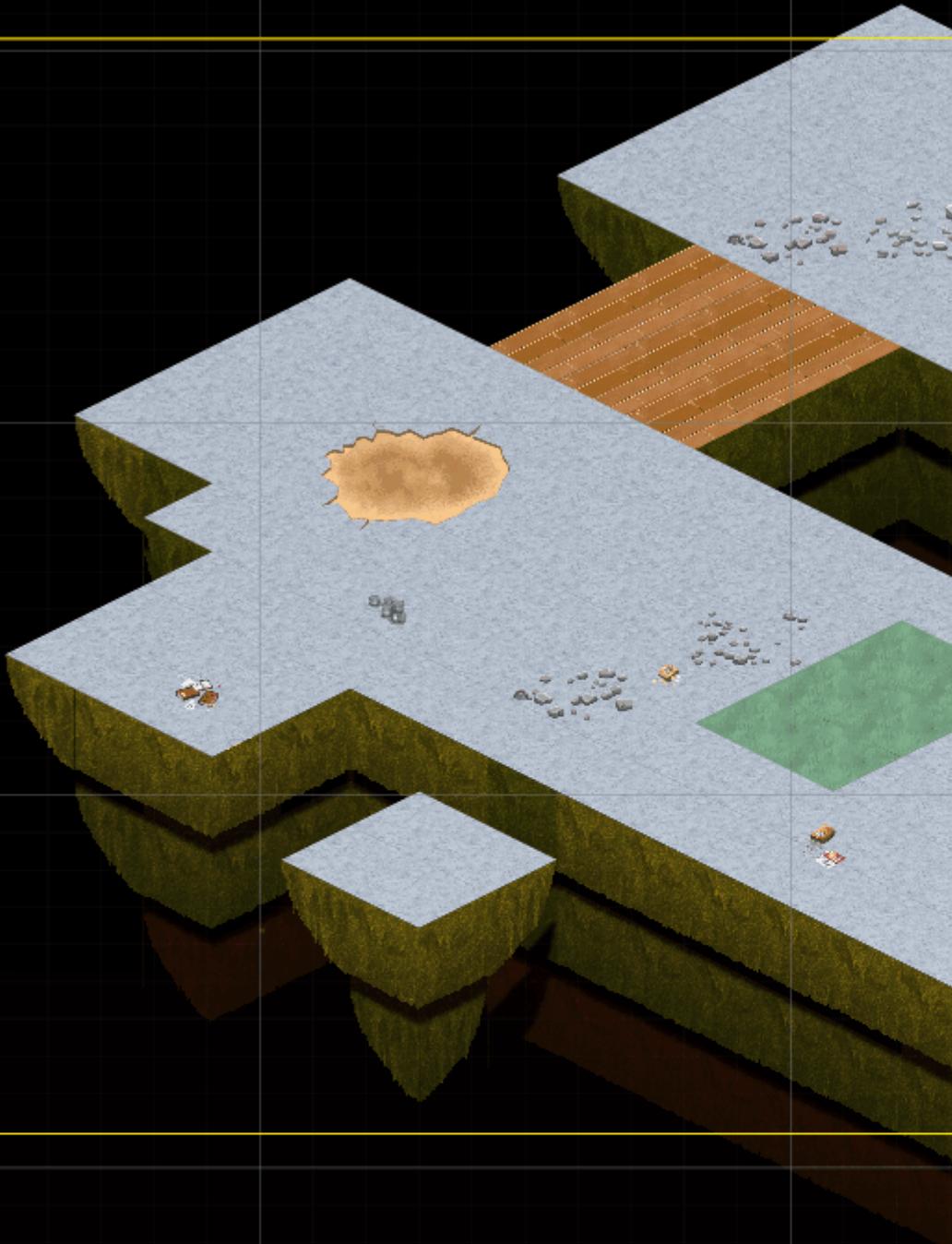
Position sceneView camera

- 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_

Capitolo 9: Fisica

Examples

Rigidbody

Panoramica

Il componente Rigidbody conferisce a GameObject una *presenza fisica* nella scena in quanto è in grado di rispondere alle forze. Potresti applicare le forze direttamente al GameObject o permetterle di reagire a forze esterne come la gravità o un altro Rigidbody che la colpisce.

Aggiunta di un componente Rigidbody

È possibile aggiungere un corpo rigido facendo clic su **Componente > Fisica > Corpo rigido**

Spostare un oggetto Rigidbody

Si raccomanda che se applichi un Rigidbody a un GameObject, usi le forze o la coppia per spostarlo invece di manipolarlo. Utilizzare i `AddForce()` o `AddTorque()` per questo:

```
// 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);
```

Massa

È possibile modificare la massa di un GameObject Rigidbody per influenzare il modo in cui reagisce con altri Rigid e forze. Una massa più alta significa che GameObject avrà più influenza su altri oggetti GameObjects basati sulla fisica e richiederà una forza maggiore per spostarsi. Oggetti con massa diversa cadranno alla stessa velocità se hanno gli stessi valori di resistenza. Per modificare la massa nel codice:

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

Trascinare

Maggiore è il valore di trascinamento, più un oggetto rallenta durante lo spostamento. Pensala come una forza opposta. Per modificare il trascinamento nel codice:

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

isKinematic

Se contrassegni un Rigidbody come **Kinematic**, questo non può essere influenzato da altre forze ma può comunque influenzare altri GameObjects. Per modificare il codice:

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

vincoli

È anche possibile aggiungere vincoli a ciascun asse per congelare la posizione o la rotazione del corpo rigido nel locale. L'impostazione predefinita è `RigidbodyConstraints.None` come mostrato qui:



Un esempio di vincoli nel codice:

```
// 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
```

È possibile utilizzare l'operatore OR bit a bit `|` combinare più vincoli in questo modo:

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

collisioni

Se vuoi un `GameObject` con un `Rigidbody` su di esso per rispondere alle collisioni dovrai anche aggiungere un collisore ad esso. I tipi di collisore sono:

- Box collider
- Spider Collider
- Capsule collider
- Volante
- Mesh collider

Se si applica più di un collisore a un `GameObject`, viene chiamato collisore composto.

È possibile creare un collider in **Trigger** per utilizzare i `OnTriggerEnter()`, `OnTriggerStay()` e `OnTriggerExit()`. Un collisore del grilletto non reagisce fisicamente alle collisioni, altri `GameObject` semplicemente lo attraversano. Sono utili per rilevare quando un altro `GameObject` si trova in una determinata area o no, ad esempio, quando si raccoglie un oggetto, potremmo essere in grado di scorrelo ma rilevare quando ciò accade.

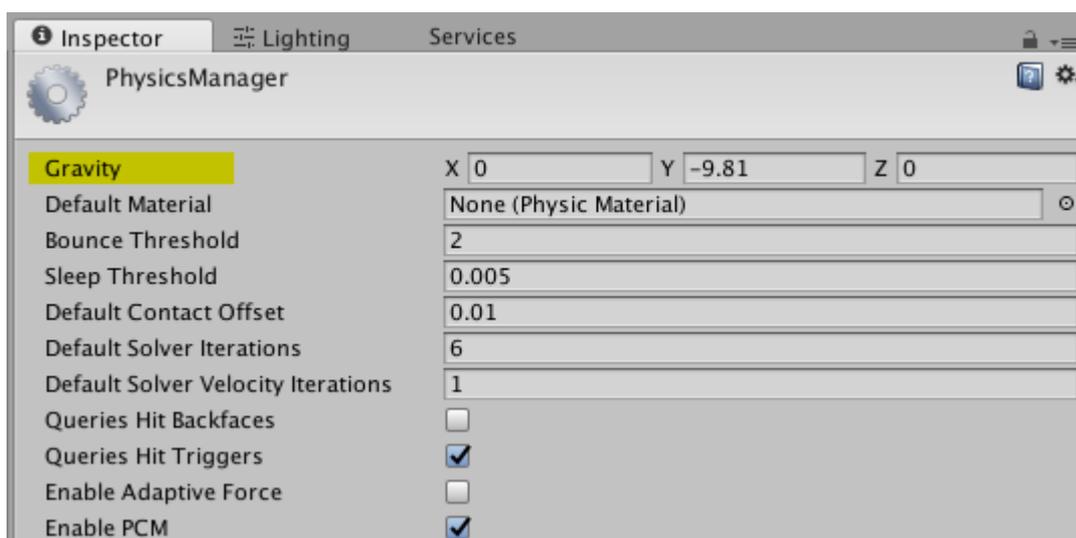
Gravità nel corpo rigido

La proprietà `useGravity` di un `Rigidbody` controlla se la gravità lo influenza o meno. Se impostato su `false` il `Rigidbody` si comporterà come se nello spazio esterno (senza una forza costante applicata ad esso in qualche direzione).

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

È molto utile nelle situazioni in cui hai bisogno di tutte le altre proprietà di `Rigidbody` eccetto il movimento controllato dalla gravità.

Quando abilitato, il `Rigidbody` sarà influenzato da una forza gravitazionale, impostata in `Physics Settings`:



La gravità è definita in unità del mondo al secondo al quadrato, ed è qui inserita come un vettore tridimensionale: significa che con le impostazioni nell'immagine di esempio, tutti i `RigidBodies` con la proprietà `useGravity` impostata su `True` sperimenteranno una forza di 9,81 unità mondiali al

secondo *al secondo* nella direzione verso il basso (poiché i valori Y negativi nel sistema di coordinate di Unity sono rivolti verso il basso).

Leggi Fisica online: <https://riptutorial.com/it/unity3d/topic/3680/fisica>

Capitolo 10: Implementazione della classe MonoBehaviour

Examples

Nessun metodo sottoposto a override

La ragione per cui non è necessario sovrascrivere `Awake`, `Start`, `Update` e altri metodi è perché non sono metodi virtuali definiti in una classe base.

La prima volta che si accede allo script, il runtime di script esamina lo script per vedere se sono definiti alcuni metodi. Se lo sono, tali informazioni vengono memorizzate nella cache e i metodi vengono aggiunti al rispettivo elenco. Queste liste sono quindi semplicemente passate in loop in momenti diversi.

Il motivo per cui questi metodi non sono virtuali dipende dalle prestazioni. Se tutti gli script avessero `Awake`, `Start`, `OnEnable`, `OnDisable`, `Update`, `LateUpdate` e `FixedUpdate`, questi sarebbero tutti aggiunti ai loro elenchi, il che significa che tutti questi metodi verranno eseguiti. Normalmente questo non sarebbe un grosso problema, tuttavia, tutte queste chiamate di metodo sono dal lato nativo (C++) al lato gestito (C#) che viene fornito con un costo di prestazioni.

Ora immagina questo, tutti questi metodi sono nelle loro liste e alcuni / molti di loro potrebbero non avere nemmeno un corpo di metodo reale. Ciò significherebbe che c'è un'enorme quantità di prestazioni sprecate nel chiamare metodi che non fanno nulla. Per evitare ciò, Unity ha scelto di non utilizzare metodi virtuali e ha creato un sistema di messaggistica che si assicura che questi metodi vengano richiamati solo quando vengono effettivamente definiti, salvando chiamate di metodo non necessarie.

Puoi leggere ulteriori informazioni sull'argomento su un blog Unity qui sopra: [10000 Update \(\) Chiamate](#) e altro su IL2CPP qui: [Introduzione agli interni IL2CPP](#)

Leggi Implementazione della classe MonoBehaviour online:

<https://riptutorial.com/it/unity3d/topic/2304/implementazione-della-classe-monobehaviour>

Capitolo 11: Importatori e (Post) Processori

Sintassi

- `AssetPostprocessor.OnPreprocessTexture ()`

Osservazioni

Utilizzare `String.Contains ()` per elaborare solo gli asset con una determinata stringa nei relativi percorsi delle risorse.

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

Examples

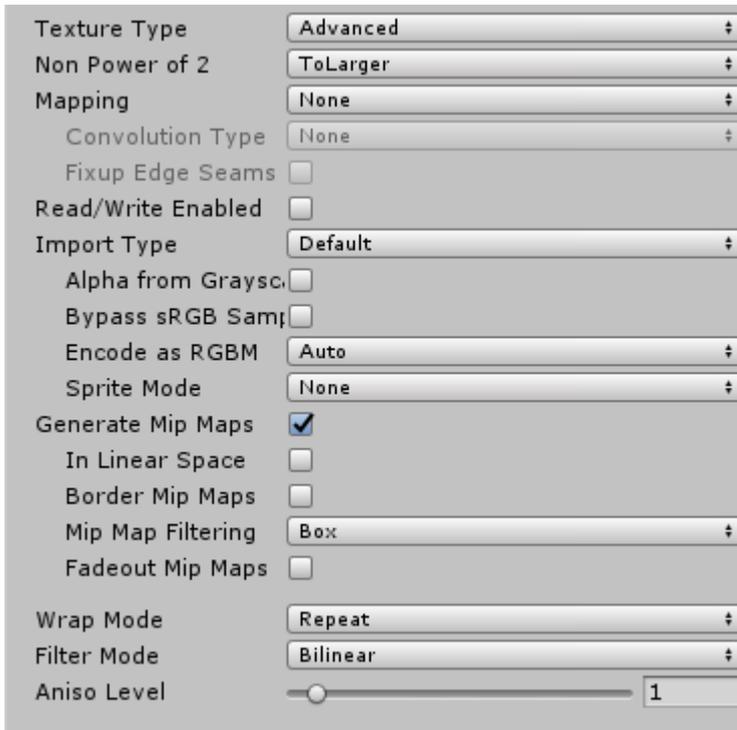
Postprocessor Texture

Creare il file `TexturePostProcessor.cs` ovunque nella cartella **Assets** :

```
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;
    }
}
```

Ora, ogni volta che Unity importa una texture avrà i seguenti parametri:



Se si utilizza il postprocessore, non è possibile modificare i parametri di trama modificando le **impostazioni di importazione** nell'editor.

Quando premi il pulsante **Applica**, la trama verrà reimportata e il codice del postprocessore verrà eseguito di nuovo.

Un importatore di base

Si supponga di avere un file personalizzato per il quale si desidera creare un importatore. Potrebbe essere un file .xls o qualsiasi altra cosa. In questo caso utilizzeremo un file JSON perché è facile, ma selezioneremo un'estensione personalizzata per rendere più semplice stabilire quali sono i nostri file?

Supponiamo che il formato del file JSON sia

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

Salviamolo come `Example.test` da qualche parte *al di fuori* delle risorse per ora.

Quindi crea un `MonoBehaviour` con una classe personalizzata solo per i dati. La classe personalizzata serve esclusivamente a deserializzare il JSON. **NON** è necessario utilizzare una classe personalizzata, ma rende questo esempio più breve. Lo salveremo in `TestData.cs`

```
using UnityEngine;
using System.Collections;

public class TestData : MonoBehaviour {
```

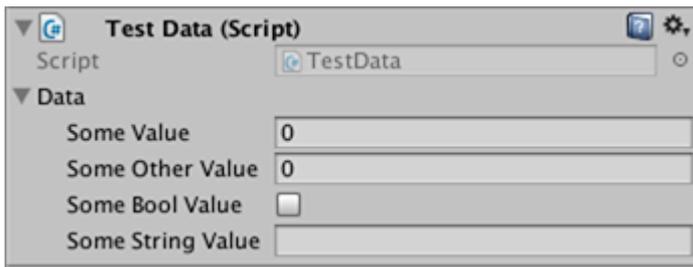
```

[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();
}

```

Se dovessi aggiungere manualmente lo script a un GameObject vedresti qualcosa di simile



Quindi crea una cartella `Editor` da qualche parte sotto `Assets` . Posso essere a qualsiasi livello All'interno della cartella dell'Editor, crea un file `TestDataAssetPostprocessor.cs` e inserisci questo.

```

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 quello salvato dovresti essere in grado di trascinare e rilasciare il file `Example.test` che

abbiamo creato sopra nella tua cartella Unity Assets e dovresti vedere il corrispondente prefabbricato creato. Se modifichi `Example.test` vedrai i dati nel prefabbricato che vengono aggiornati immediatamente. Se trascini il prefabbricato nella gerarchia delle scene, lo vedrai aggiornare così come le modifiche a `Example.test`. Se sposti `Example.test` in un'altra cartella, il prefabbricato corrispondente si muoverà con esso. Se modifichi un campo su un'istanza, quindi modifica il file `Example.test`, vedrai solo i campi che non hai modificato sull'istanza da aggiornare.

Miglioramenti: nell'esempio sopra, dopo aver trascinato `Example.test` nella cartella `Assets`, vedrai sia un `Example.test` che un `Example.test.prefab`. Sarebbe bello sapere che funziona più come gli importatori di modelli funzionano, ma magicamente vedi solo `Example.test` ed è un `AssetBundle` o qualcosa del genere. Se sai come si prega di fornire quell'esempio

Leggi [Importatori e \(Post\) Processor online](https://riptutorial.com/it/unity3d/topic/5279/importatori-e-post-processor): <https://riptutorial.com/it/unity3d/topic/5279/importatori-e-post-processor>

Capitolo 12: Integrazione annunci

introduzione

Questo argomento riguarda l'integrazione di servizi pubblicitari di terze parti, come Unity Ads o Google AdMob, in un progetto Unity.

Osservazioni

Questo vale per [Unity Ads](#) .

Assicurarsi che la modalità di test per Unity Ads sia abilitata durante lo sviluppo

Tu, in quanto sviluppatore, non puoi generare impressioni o installazioni facendo clic sugli annunci nel tuo gioco. Ciò viola i [Termini di servizio di Unity Ads](#) e verrai escluso dalla rete Unity Ads per tentata frode.

Per ulteriori informazioni, leggi l'accordo sui [Termini di servizio di Unity Ads](#) .

Examples

Nozioni sugli annunci Unity in 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
        }
    }
}
```

Nozioni sugli annunci Unity in 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
    }
}
```

Leggi Integrazione annunci online: <https://riptutorial.com/it/unity3d/topic/9796/integrazione-annunci>

Capitolo 13: Livelli

Examples

Utilizzo dei livelli

I livelli Unity sono simili ai tag, in quanto possono essere utilizzati per definire oggetti che devono essere interagiti con o dovrebbero comportarsi in un certo modo, tuttavia, i livelli vengono principalmente utilizzati con le funzioni della classe `Physics` : [Unity Documentation - Physics](#)

I livelli sono rappresentati da un numero intero e possono essere passati alle funzioni in questo modo:

```
using UnityEngine;
class LayerExample {

    public int layer;

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

L'utilizzo di un layer in questo modo includerà solo Collider i cui GameObject hanno il layer specificato nei calcoli eseguiti. Ciò rende più semplice la logica e migliora le prestazioni.

Struttura LayerMask

La struttura `LayerMask` è un'interfaccia che funziona quasi esattamente come passare un intero alla funzione in questione. Tuttavia, il suo più grande vantaggio è consentire all'utente di selezionare il livello in questione da un menu a discesa nell'ispettore.

```
using UnityEngine;
class LayerMaskExample{

    public LayerMask mask;
    public Vector3 direction;

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

Dispone inoltre di più funzioni statiche che consentono di convertire i nomi dei livelli in indici o indici in nomi di layer.

```
using UnityEngine;
class NameToLayerExample{

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

Per rendere più semplice il controllo dei livelli, definire il seguente metodo di estensione.

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

    return result;
}
```

Questo metodo ti consentirà di verificare se un oggetto di gioco si trova in una maschera (selezionata nell'editor) o meno.

Leggi Livelli online: <https://riptutorial.com/it/unity3d/topic/4762/livelli>

Capitolo 14: Modelli di progettazione

Examples

Modello di progettazione Model View Controller (MVC)

Il controller della vista del modello è un modello di progettazione molto comune che esiste da un po' di tempo. Questo modello si concentra sulla riduzione del codice degli *spaghetti* separando le classi in parti funzionali. Recentemente ho sperimentato questo modello di design in Unity e vorrei presentare un esempio di base.

Un design MVC è costituito da tre parti principali: Modello, Vista e Controller.

Modello: il modello è una classe che rappresenta la porzione di dati del tuo oggetto. Questo potrebbe essere un giocatore, inventario o un intero livello. Se programmato correttamente, dovresti essere in grado di prendere questo script e usarlo al di fuori di Unity.

Nota alcune cose sul modello:

- Non dovrebbe ereditare da MonoBehaviour
- Non dovrebbe contenere codice Unity specifico per la portabilità
- Dal momento che stiamo evitando le chiamate Unity API, questo può ostacolare cose come convertitori impliciti nella classe Model (sono necessari workaround)

Player.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 classe Vector3 personalizzata da utilizzare con il nostro modello di dati.

```
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;
    }
}
```

Visualizza: la vista è una classe che rappresenta la porzione di visualizzazione legata al modello. Questa è una classe appropriata per derivare da MonoBehaviour. Questo dovrebbe contenere codice che interagisce direttamente con API specifiche di Unity, tra cui `OnCollisionEnter`, `Start`, `Update`, ecc ...

- Tipicamente eredita da MonoBehaviour
- Contiene codice specifico di unità

PlayerView.cs

```
using UnityEngine;

public class PlayerView : MonoBehaviour
{
    public void SetPosition(Vector3 position)
    {
        transform.position = position;
    }
}
```

Controller: il controller è una classe che unisce sia il modello che la vista. I controller mantengono sia il modello che la vista sincronizzati così come l'interazione tra le unità. Il controller può ascoltare gli eventi di entrambi i partner e aggiornarli di conseguenza.

- Lega sia il modello che la vista sincronizzando lo stato
- Può guidare l'interazione tra i partner
- I controller possono essere o meno portatili (potrebbe essere necessario utilizzare il codice Unity qui)
- Se decidi di non rendere portatile il tuo controller, considera la possibilità di renderlo un MonoBehaviour per aiutare nell'ispezione degli 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 finale

Ora che abbiamo tutti i pezzi principali, possiamo creare una fabbrica che genererà tutte e tre le parti.

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);
    }
}

```

E finalmente possiamo chiamare la fabbrica da un manager ...

Manager.cs

```
using UnityEngine;

public class Manager : MonoBehaviour
{
    [ContextMenu("Load Player")]
    private void LoadPlayer()
    {
        new PlayerFactory().Load();
    }
}
```

Allega lo script Manager a un GameObject vuoto nella scena, fai clic con il tasto destro del mouse sul componente e seleziona "Carica Player".

Per una logica più complessa è possibile introdurre l'ereditarietà con classi di base e interfacce astratte per un'architettura migliorata.

Leggi Modelli di progettazione online: <https://riptutorial.com/it/unity3d/topic/10842/modelli-di-progettazione>

Capitolo 15: Negozio di beni

Examples

Accedere al negozio di beni

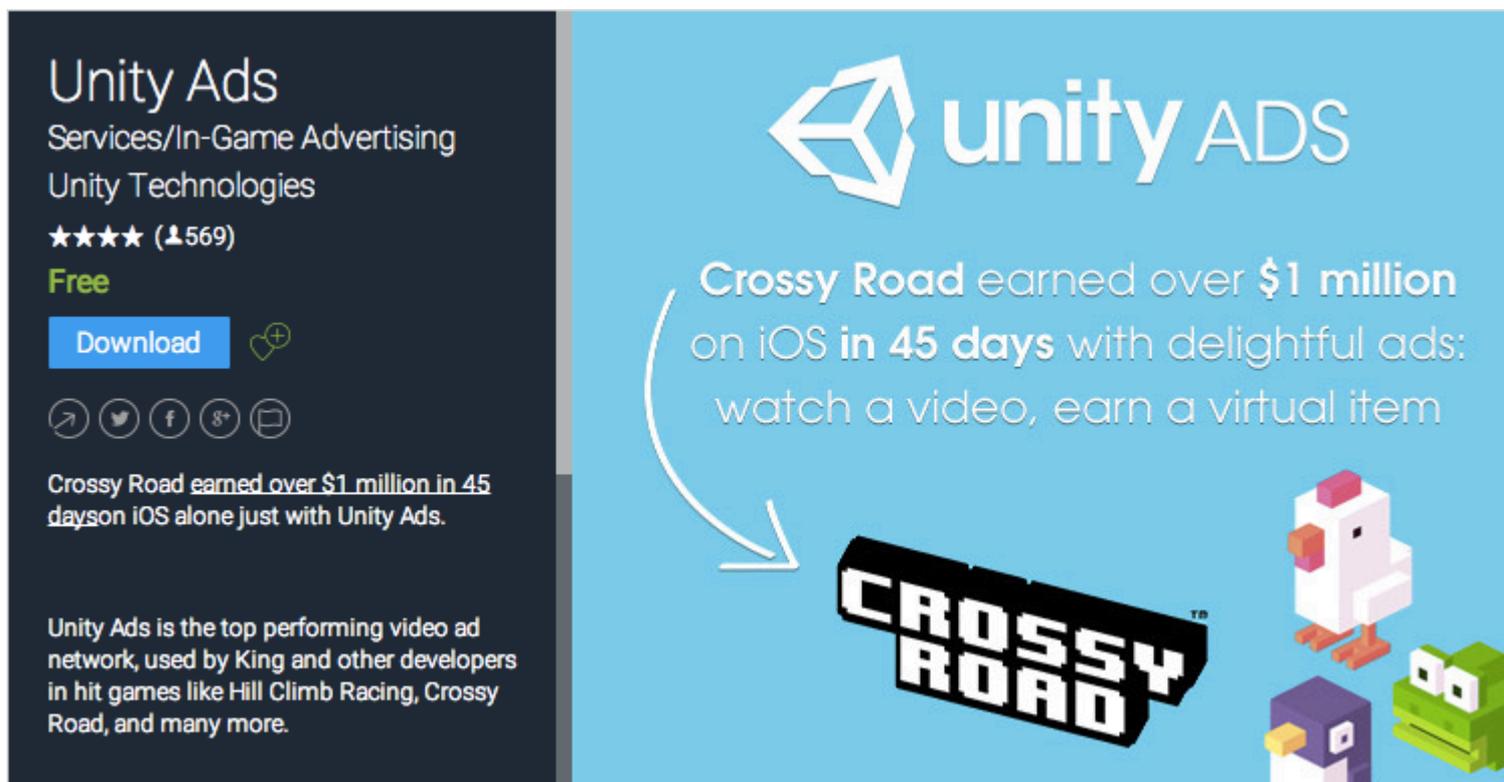
Esistono tre modi per accedere a Unity Asset Store:

- Aprire la finestra Asset Store selezionando Finestra → Archivio risorse dal menu principale in Unity.
- Usa il tasto di scelta rapida (Ctrl + 9 su Windows / 9 su Mac OS)
- Sfoglia l'interfaccia web: <https://www.assetstore.unity3d.com/>

Potrebbe essere richiesto di creare un account utente gratuito o di accedere se è la prima volta che accedi a Unity Asset Store.

Acquisti di beni

Dopo aver effettuato l'accesso all'Asset Store e visualizzato la risorsa che desideri scaricare, fai semplicemente clic sul pulsante **Scarica** . Il testo del pulsante potrebbe anche essere **Acquista ora** se la risorsa ha un costo associato.



The image shows two parts: a dark-themed app store listing for 'Unity Ads' and a light blue promotional banner for the game 'Crossy Road'. The app store listing includes the text 'Unity Ads Services/In-Game Advertising Unity Technologies', a 4-star rating with 569 reviews, and a 'Free' price tag. A blue 'Download' button is visible. Below the button are social media icons and a quote: 'Crossy Road earned over \$1 million in 45 days on iOS alone just with Unity Ads.' The banner features the Unity logo and 'unity ADS' text, followed by the same quote. At the bottom of the banner, the 'CROSSY ROAD' logo is shown in a 3D, blocky font, with a white chicken, a purple penguin, and a green frog character nearby.

Se stai visualizzando Unity Asset Store tramite l'interfaccia web, il testo del pulsante **Download** potrebbe invece essere visualizzato come **Aperto in Unity** . Selezionando questo pulsante si avvia un'istanza di Unity e si visualizza l'asset all'interno della *finestra Asset Store* .

È possibile che venga richiesto di creare un account utente gratuito o effettuare l'accesso se è la

prima volta che acquisti da Unity Asset Store.

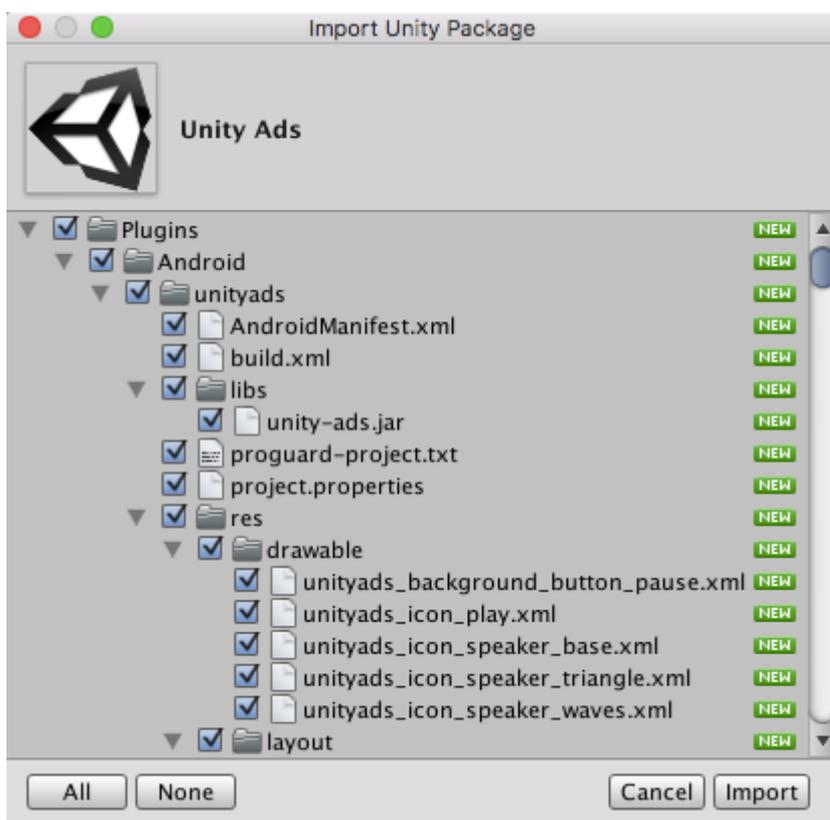
Unity procederà quindi con l'accettazione del pagamento, se applicabile.

Importazione di beni

Dopo che la risorsa è stata scaricata in Unity, il pulsante **Download** o **Compra ora** cambierà in **Importa**.

Selezionando questa opzione, all'utente verrà visualizzata una finestra *Importa pacchetto unità*, in cui l'utente può selezionare i file di asset di cui desidera importare all'interno del proprio progetto.

Selezionare **Importa** per confermare il processo, collocando i file di asset selezionati all'interno della cartella Risorse mostrata nella *finestra Vista progetto*.



Attività editoriali

1. crea un account editore
2. aggiungi una risorsa nell'account editore
3. scarica gli strumenti del deposito risorse (dall'archivio risorse)
4. vai su "Asset Store Tools"> "Package Upload"
5. selezionare il pacchetto e la cartella del progetto corretti nella finestra degli strumenti del magazzino degli asset
6. fai clic su carica
7. invia la tua risorsa online

TODO: aggiungi immagini, maggiori dettagli

Conferma il numero di fattura di un acquisto

Il numero di fattura viene utilizzato per verificare la vendita per gli editori. Molti editori di asset pagati o plugin chiedono il numero di fattura su richiesta di supporto. Il numero di fattura viene anche utilizzato come chiave di licenza per attivare alcuni asset o plug-in.

Il numero della fattura può essere trovato in due punti:

1. Dopo aver acquistato la risorsa, ti verrà inviata una email il cui oggetto è "conferma acquisto di Unity Asset Store ...". Il numero di fattura si trova nell'allegato PDF di questa e-mail.



UNITY3D.COM

Unity Technologies ApS

Vendersgade 28
1363 København K
Danmark

INVOICE

Invoice No.	[REDACTED]
Date	[REDACTED]
Due Date	[REDACTED]
Order No.	[REDACTED]

2. Aprire <https://www.assetstore.unity3d.com/#!/account/transactions> , quindi è possibile trovare il numero di fattura nella colonna *Descrizione* .

Credit Card / PayPal		
Date	Action	Description
[REDACTED]	CREDIT CARD / PAYPAL	# [REDACTED] 30 Mesh Terrain Editor Pro

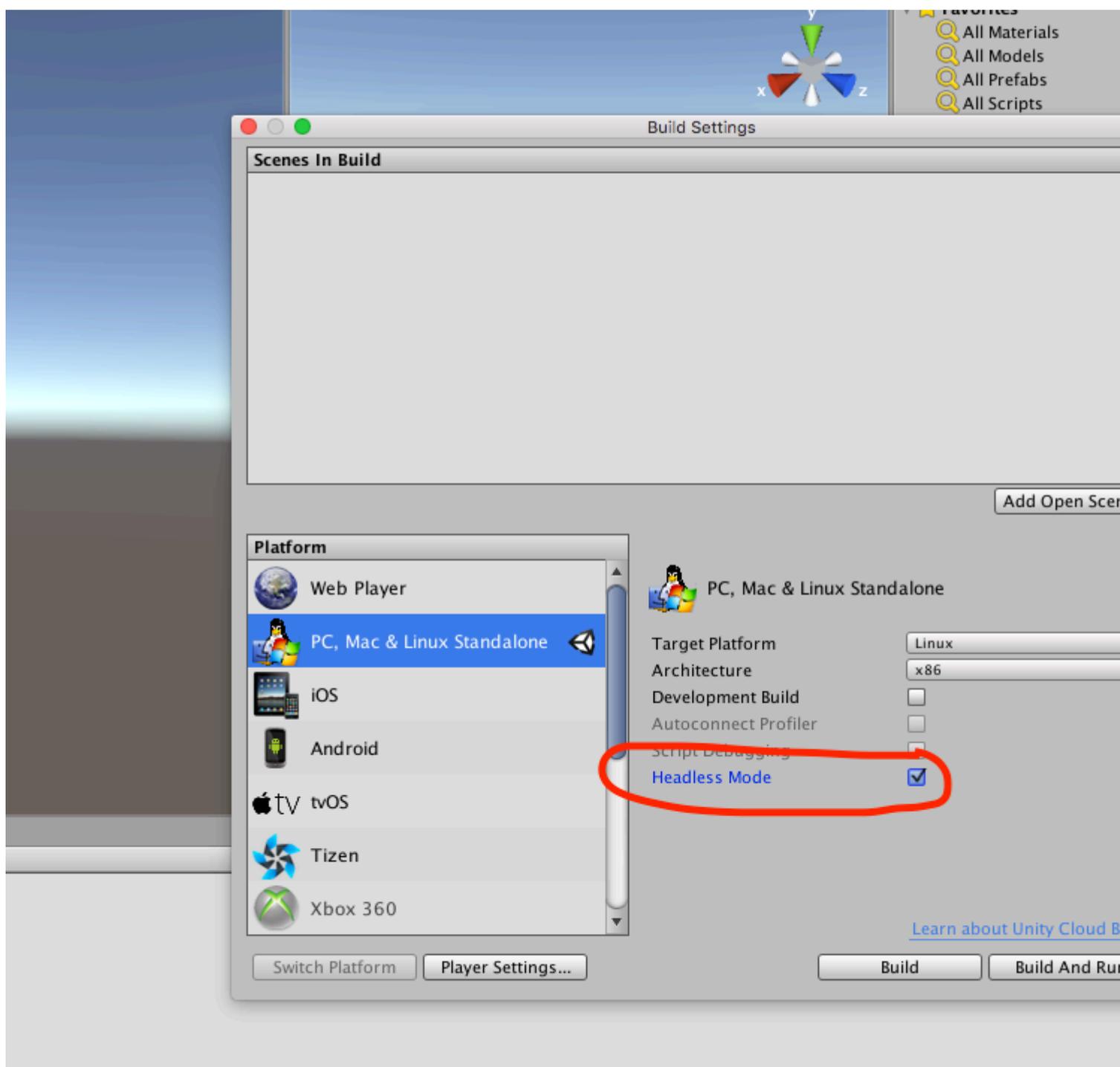
Leggi Negozio di beni online: <https://riptutorial.com/it/unity3d/topic/5705/negozio-di-beni>

Capitolo 16: Networking

Osservazioni

Modalità senza testa in Unity

Se stai creando un server da distribuire in Linux, le impostazioni di Build hanno un'opzione "Modalità senza testa". Una build dell'applicazione con questa opzione non visualizza nulla e non legge l'input dell'utente, che di solito è ciò che vogliamo per un server.



Examples

Creazione di un server, un client e invio di un messaggio.

Unity networking fornisce l'High Level API (HLA) per gestire l'astrazione delle comunicazioni di rete da implementazioni di basso livello.

In questo esempio vedremo come creare un server in grado di comunicare con uno o più client.

L'HLA ci consente di serializzare facilmente una classe e inviare oggetti di questa classe sulla rete.

La classe che stiamo usando per serializzare

Questa classe deve ereditare da `MessageBase`, in questo esempio invieremo semplicemente una stringa all'interno di questa classe.

```
using System;
using UnityEngine.Networking;

public class MyNetworkMessage : MessageBase
{
    public string message;
}
```

Creazione di un server

Creiamo un server che ascolta la porta 9999, consente un massimo di 10 connessioni e legge gli oggetti dalla rete della nostra classe personalizzata.

L'HLA associa diversi tipi di messaggi a un ID. Esistono tipi di messaggi predefiniti definiti nella classe `MsgType` da Unity Networking. Ad esempio il tipo di connessione ha ID 32 e viene chiamato nel server quando un client si connette ad esso o nel client quando si connette a un server. È possibile registrare i gestori per gestire i diversi tipi di messaggio.

Quando invii una classe personalizzata, come nel nostro caso, definiamo un gestore con un nuovo ID associato alla classe che stiamo inviando tramite la rete.

```
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 MessageType
    NetworkServer.RegisterHandler (MessageType.Connect, OnClientConnected);
    NetworkServer.RegisterHandler (MessageType.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 conected to the server";

// Broadcast a message a to everyone connected
NetworkServer.SendToAll(messageID,messageContainer);
}

void OnClientDisconnected(NetworkMessage netMessage)
{
    // Do stuff when a client disssconnects
}

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);
}
}
}

```

Il cliente

Ora creiamo 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);
}
}

```

Leggi Networking online: <https://riptutorial.com/it/unity3d/topic/5671/networking>

Capitolo 17: Ottimizzazione

Osservazioni

1. Se possibile, disabilita gli script sugli oggetti quando non sono necessari. Ad esempio, se hai uno script su un oggetto nemico che cerca e spara al giocatore, considera di disabilitare questo script quando il nemico è troppo lontano ad esempio dal giocatore.

Examples

Controlli veloci ed efficienti

Evita le operazioni non necessarie e le chiamate ai metodi ovunque tu sia, specialmente in un metodo chiamato molte volte al secondo, come `Update`.

Controllo distanza / intervallo

Usa `sqrMagnitude` invece di `magnitude` quando confronti le distanze. Ciò evita operazioni `sqrt` non necessarie. Nota che quando usi `sqrMagnitude`, anche il lato destro deve essere quadrato.

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

Controlli limitati

Le intersezioni degli oggetti possono essere controllate in modo grossolano controllando se i loro limiti di `Collider` / `Renderer` intersecano. La struttura `Bounds` ha anche un pratico metodo `Intersects` che aiuta a determinare se due limiti si intersecano.

`Bounds` ci aiutano anche a calcolare approssimativamente la distanza *effettiva* (da superficie a superficie) tra gli oggetti (vedere `Bounds.SqrDistance`).

Avvertenze

Il controllo dei limiti funziona molto bene per gli oggetti convessi, ma i controlli dei limiti sugli oggetti concavi possono portare a imprecisioni molto più elevate a seconda della forma dell'oggetto.

L'utilizzo di `Mesh.bounds` non è raccomandato in quanto restituisce limiti di spazio locali. Utilizzare invece `MeshRenderer.bounds`.

Coroutine Power

USO

Se hai una lunga esecuzione che si basa [sull'API Unity non thread-safe](#), usa [Coroutines](#) per suddividerla su più frame e mantenere la tua applicazione reattiva.

[Le Coroutine](#) aiutano anche a eseguire costose azioni ogni ennesimo fotogramma invece di eseguire quell'azione ogni fotogramma.

Divisione di routine a esecuzione prolungata su più frame

Le Coroutine aiutano a distribuire operazioni a lungo termine su più fotogrammi per mantenere il framerate della tua applicazione.

Le routine che dipingono o generano terreno proceduralmente o generano rumore sono esempi che potrebbero richiedere il trattamento di 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
    }
}
```

Il codice sopra è un esempio facile da capire. Nel codice di produzione è meglio evitare il controllo per pixel che controlla quando `yield return` (forse farlo ogni 2-3 righe) e pre-calcolare `for` lunghezza loop in anticipo.

Esecuzione di azioni costose meno frequentemente

Le coroutine ti aiutano a eseguire azioni costose meno frequentemente, in modo che non sia il più grande successo di prestazioni che sarebbe se eseguiessi ogni fotogramma.

Prendendo il seguente esempio direttamente dal [Manuale](#) :

```
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);
    }
}
}

```

I test di prossimità possono essere ulteriormente ottimizzati utilizzando l' [API CullingGroup](#) .

Insidie comuni

Un errore comune che gli sviluppatori creano è l'accesso ai risultati o agli effetti collaterali delle coroutine *al di fuori* della coroutine. Le coroutine restituiscono il controllo al chiamante non appena viene rilevata una dichiarazione di `yield return` e il risultato o l'effetto collaterale non possono ancora essere eseguiti. Per aggirare i problemi in cui è *necessario* utilizzare il risultato / l'effetto collaterale al di fuori della coroutine, controllare [questa risposta](#) .

stringhe

Si potrebbe sostenere che ci sono più risorse per le risorse in Unity rispetto alla stringa umile, ma è uno degli aspetti più facili da risolvere in anticipo.

Le operazioni con le stringhe costruiscono garbage

La maggior parte delle operazioni con le stringhe costruisce piccole quantità di spazzatura, ma se quelle operazioni vengono chiamate più volte nel corso di un singolo aggiornamento, si accumula. Col tempo, attiverà la Garbage Collection automatica, che potrebbe determinare un picco della CPU visibile.

Memorizza le tue operazioni sulle stringhe

Considera il seguente esempio.

```

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");
}

```

Può sembrare sciocco e ridondante, ma se stai lavorando con Shader, potresti imbatterti in situazioni come queste. La memorizzazione nella cache delle chiavi farà la differenza.

Si prega di notare che le *stringhe* e le *costanti* non generano rifiuti, in quanto vengono iniettati in modo statico nello spazio stack del programma. Se si *generano* stringhe in fase di esecuzione e si *garantisce* che generano sempre le **stesse** stringhe come nell'esempio precedente, il caching sarà sicuramente di aiuto.

Per altri casi in cui la stringa generata non è la stessa ogni volta, non c'è altra alternativa alla generazione di quelle stringhe. In quanto tale, il picco di memoria con stringhe di generazione manuale ogni volta è generalmente trascurabile, a meno che non vengano generate decine di migliaia di stringhe alla volta.

La maggior parte delle operazioni con le stringhe sono messaggi di debug

Fare operazioni con le stringhe per i messaggi di debug, es. `Debug.Log("Object Name: " + obj.name)` va bene e non può essere evitato durante lo sviluppo. Tuttavia, è importante assicurarsi che i messaggi di debug non pertinenti non finiscano nel prodotto rilasciato.

Un modo è utilizzare l' [attributo Conditional](#) nelle chiamate di debug. Questo non solo rimuove le chiamate al metodo, ma anche tutte le operazioni sulle stringhe che vi entrano.

```

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);
    }
}

```

Questo è un esempio semplificato. Potresti voler investire un po' di tempo nella progettazione di una routine di registrazione più completa.

Confronto di stringhe

Questa è una piccola ottimizzazione, ma vale la pena menzionarla. Il confronto delle stringhe è leggermente più complicato di quanto si possa pensare. Il sistema proverà a tenere conto delle differenze culturali per impostazione predefinita. Puoi scegliere di utilizzare un semplice confronto binario, che esegue più velocemente.

```

// 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>();

```

Riferimenti di cache

Riferimenti della cache per evitare le costose chiamate, specialmente nella funzione di aggiornamento. Questo può essere fatto inserendo nella cache questi riferimenti all'avvio se disponibili o quando disponibili e controllando null / bool flat per evitare di ottenere nuovamente il riferimento.

Esempi:

Riferimenti ai componenti della cache

modificare

```
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);
}
```

Riferimenti all'oggetto cache

modificare

```
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));
}
```

Inoltre, memorizza nella cache chiamate costose come le chiamate a `Mathf` ove possibile.

Evita di chiamare metodi usando stringhe

Evita di chiamare metodi usando stringhe che possono accettare metodi. Questo approccio farà uso di riflessioni che possono rallentare il tuo gioco, specialmente se usato nella funzione di aggiornamento.

Esempi:

```
//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 i metodi di unità vuoti

Evita i metodi di unità vuoti. Oltre ad essere un cattivo stile di programmazione, c'è un piccolo overhead coinvolto nello scripting di runtime. In molti casi, questo può accumularsi e influire sulle prestazioni.

```
void Update
{
}

void FixedUpdate
{
}
```

Leggi Ottimizzazione online: <https://riptutorial.com/it/unity3d/topic/3433/ottimizzazione>

Capitolo 18: Piattaforme mobili

Sintassi

- `public static int Input.touchCount`
- tocco statico pubblico `Input.GetTouch (int index)`

Examples

Rilevazione del tocco

Per rilevare un tocco in Unity è abbastanza semplice, dobbiamo solo usare `Input.GetTouch()` e passargli un indice.

```
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
            }
        }
    }
}
```

Questi esempi hanno il tocco dell'ultimo frame del gioco.

TouchPhase

All'interno dell'enumer `TouchPhase` ci sono 5 diversi tipi di `TouchPhase`

- Iniziatò: un dito toccò lo schermo
- Spostato - un dito spostato sullo schermo
- Stazionario: un dito è sullo schermo ma non si muove
- Finito - un dito fu sollevato dallo schermo
- Annullato: il sistema ha annullato il tracciamento per il tocco

Ad esempio, per spostare l'oggetto questo script è collegato a tutto lo schermo in base al tocco.

```
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);
        }
    }
}
```

Leggi Piattaforme mobili online: <https://riptutorial.com/it/unity3d/topic/6285/piattaforme-mobili>

Capitolo 19: Plugin Android 101 - Un'introduzione

introduzione

Questo argomento è la prima parte di una serie su come creare plugin Android per Unity. Inizia da qui se hai poca o nessuna esperienza nella creazione di plugin e / o del sistema operativo Android.

Osservazioni

Attraverso questa serie, utilizzo estesamente link esterni che consiglio di leggere. Mentre le versioni parafrasate del contenuto pertinente saranno incluse qui, ci possono essere momenti in cui la lettura aggiuntiva aiuterà.

A partire dai plugin Android

Al momento, Unity offre due modi per chiamare il codice Android nativo.

1. Scrivi codice Android nativo in Java e chiama queste funzioni Java utilizzando C #
2. Scrivi il codice C # per chiamare direttamente le funzioni che fanno parte del sistema operativo Android

Per interagire con il codice nativo, Unity fornisce alcune classi e funzioni.

- [AndroidJavaObject](#) - Questa è la classe base che Unity fornisce per interagire con il codice nativo. Quasi tutti gli oggetti restituiti dal codice nativo possono essere archiviati come e [AndroidJavaObject](#)
 - [AndroidJavaClass](#) : eredita [AndroidJavaObject](#). Questo è usato per referenziare le classi nel tuo codice nativo
 - [Get](#) / [Set](#) valori di un'istanza di un oggetto nativo, e le statiche [GetStatic](#) / [SetStatic](#) versioni
 - [Chiama](#) / [CallStatic](#) per chiamare funzioni native non statiche e statiche
-

Cenni sulla creazione di un plug-in e una terminologia

1. Scrivi codice Java nativo in [Android Studio](#)
2. Esportare il codice in un file JAR / AAR (passaggi qui per [file JAR](#) e [file AAR](#))
3. Copia il file JAR / AAR nel tuo progetto Unity in **Risorse / Plugin / Android**
4. Scrivi il codice in Unity (C # è sempre stato il modo di andare qui) per chiamare le funzioni nel plugin

Nota che i primi tre passaggi si applicano SOLO se desideri avere un plugin nativo!

Da qui in poi, mi riferirò al file JAR / AAR come al **plug-in nativo** e allo script C # come al **wrapper C #**

Scegliere tra i metodi di creazione del plugin

È ovvio che il primo modo di creare plug-in è lungo, quindi scegliere la tua rotta sembra discutibile. Tuttavia, il metodo 1 è l'unico modo per chiamare il codice personalizzato. Quindi, come si sceglie?

In poche parole, fa il tuo plugin

1. Coinvolgi codice personalizzato: scegli il metodo 1
2. Richiama solo le funzioni Android native? - Scegli il metodo 2

Per favore **NON** provare a "mixare" (cioè una parte del plugin usando il metodo 1 e l'altra usando il metodo 2) i due metodi! Anche se interamente possibile, è spesso poco pratico e doloroso da gestire.

Examples

UnityAndroidPlugin.cs

Crea un nuovo script C # in Unity e sostituiscilo con il seguente

```
using UnityEngine;
using System.Collections;

public static class UnityAndroidPlugin {

}
```

UnityAndroidNative.java

Crea una nuova classe Java in Android Studio e sostituiscile con i seguenti contenuti

```
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

Crea un nuovo script C # in Unity e incolla questi contenuti

```
using UnityEngine;
using System.Collections;

public class UnityAndroidPluginGUI : MonoBehaviour {

    void OnGUI () {

    }

}
```

Leggi Plugin Android 101 - Un'introduzione online:

<https://riptutorial.com/it/unity3d/topic/10032/plugin-android-101---un-introduzione>

Capitolo 20: Pooling di oggetti

Examples

Pool di oggetti

A volte quando realizzi un gioco devi creare e distruggere molti oggetti dello stesso tipo più e più volte. Puoi farlo semplicemente creando un prefabbricato e istanziato / distruggi questo quando ne hai bisogno, tuttavia, farlo è inefficiente e può rallentare il tuo gioco.

Un modo per aggirare questo problema è il pool di oggetti. Fondamentalmente ciò significa che hai una piscina (con o senza un limite per la quantità) di oggetti che dovrai riutilizzare ogni volta che puoi per evitare un'inutile istanziazione o distruzione.

Di seguito è riportato un esempio di un pool di oggetti semplice

```
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;
}
}

```

Passiamo prima alle variabili

```

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

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

```

- `GameObject prefab` : questo è il prefabbricato che il pool di oggetti utilizzerà per istanziare nuovi oggetti nel pool.
- `int amount` : questa è la quantità massima di articoli che possono essere nel pool. Se si desidera creare un'istanza di un altro elemento e il pool ha già raggiunto il limite, verrà utilizzato un altro elemento del pool.
- `bool populateOnStart` : puoi scegliere di popolare il pool all'avvio oppure no. Così facendo riempirai il pool con istanze del prefabbricato in modo che la prima volta che chiami `Instantiate` otterrai un oggetto già esistente
- `bool growOverAmount` : l'impostazione su `true` consente al pool di crescere ogni volta che l'importo è richiesto in un determinato intervallo di tempo. Non si è sempre in grado di prevedere con precisione la quantità di articoli da inserire nella propria piscina, in modo tale da aggiungere di più alla propria piscina quando necessario.
- `List<GameObject> pool` : questo è il pool, il luogo in cui sono memorizzati tutti gli oggetti istanziati / distrutti.

Ora diamo un'occhiata alla funzione `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);
        }
    }
}

```

Nella funzione di avvio controlliamo se dovremmo compilare la lista all'avvio e farlo se il `prefab` è stato impostato e la quantità è maggiore di 0 (altrimenti la creazione sarebbe indefinita).

Questo è solo un semplice ciclo per istanziare nuovi oggetti e metterli in piscina. Una cosa a cui prestare attenzione è che impostiamo tutte le istanze su inattive. In questo modo non sono ancora

visibili nel gioco.

Successivamente, c'è la funzione di `Instantiate`, che è dove la maggior parte della magia accade

```
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 funzione `Instantiate` la stessa funzione di `Instantiate` Unity, ad eccezione del fatto che il prefabbricato è già stato fornito sopra come membro della classe.

Il primo passo della funzione `Instantiate` è controllare se c'è un oggetto inattivo nel pool in questo momento. Ciò significa che possiamo riutilizzare quell'oggetto e restituirlo al richiedente. Se è presente un oggetto inattivo nel pool, impostiamo la posizione e la rotazione, impostandolo come attivo (altrimenti potrebbe essere riutilizzato per errore se si dimentica di attivarlo) e restituirlo al richiedente.

Il secondo passaggio avviene solo se non ci sono elementi inattivi nel pool e il pool può crescere oltre l'importo iniziale. Quello che succede è semplice: un'altra istanza del prefabbricato viene creata e aggiunta al pool. Permettere la crescita della piscina ti aiuta ad avere la giusta quantità di oggetti nella piscina.

Il terzo "passo" si verifica solo se non ci sono elementi inattivi nel pool e il pool *non* è autorizzato a crescere. Quando ciò accade, il richiedente riceverà un `GameObject` null, il che significa che non è disponibile nulla e che dovrebbe essere gestito correttamente per prevenire

`NullReferenceExceptions`.

Importante!

Per assicurarsi che le voci tornare in piscina **non** si deve distruggere gli oggetti di gioco. L'unica cosa che devi fare è impostarli su inattivo e questo li renderà disponibili per il riutilizzo attraverso il pool.

Pool di oggetti semplice

Di seguito è riportato un esempio di un pool di oggetti che consente di affittare e restituire un determinato tipo di oggetto. Per creare il pool di oggetti è necessario utilizzare Func per la funzione di creazione e un'azione per distruggere l'oggetto per garantire all'utente flessibilità. Alla richiesta di un oggetto quando il pool è vuoto verrà creato un nuovo oggetto e su richiesta quando il pool ha oggetti, gli oggetti verranno rimossi dal pool e restituiti.

Pool di oggetti

```
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;
        }
    }
}
```

Esempio di utilizzo

```
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.
    }
}
```

Un altro pool di oggetti semplice

Un altro esempio: un'arma che spara proiettili.

L'arma funge da pool di oggetti per i proiettili che 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);
    }
}
}

```

Leggi Pooling di oggetti online: <https://riptutorial.com/it/unity3d/topic/2276/pooling-di-oggetti>

Capitolo 21: prefabbricati

Sintassi

- oggetto statico pubblico PrefabUtility.InstantiatePrefab (Object target);
- oggetto statico pubblico Object AssetDatabase.LoadAssetAtPath (string assetPath, Type type);
- oggetto statico pubblico Object.Instantiate (oggetto originale);
- public static Object Resources. Load (string path);

Examples

introduzione

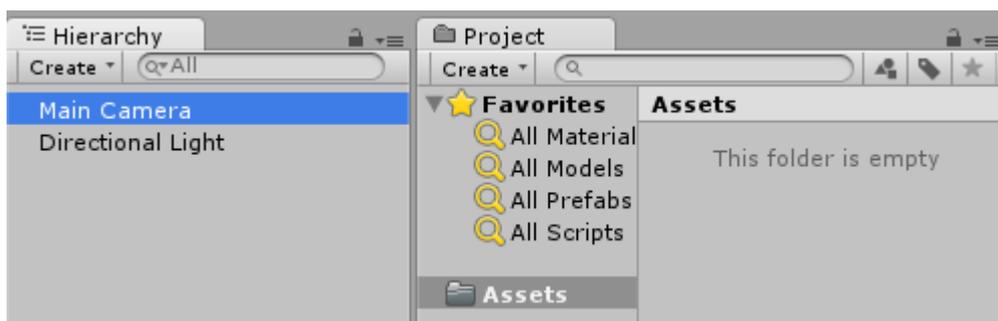
Le **prefabbricate** sono un tipo di risorsa che consente di archiviare un GameObject completo con i suoi componenti, proprietà, componenti allegati e valori di proprietà serializzati. Ci sono molti scenari in cui questo è utile, tra cui:

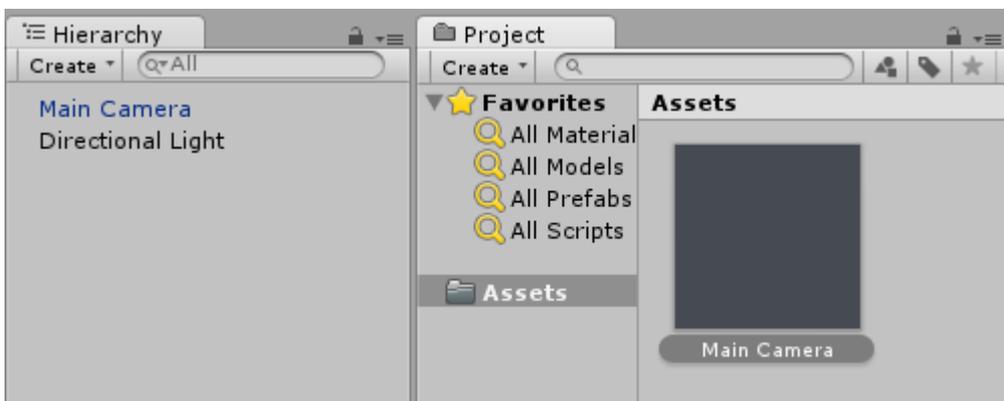
- Duplicazione di oggetti in una scena
- Condivisione di un oggetto comune su più scene
- Essere in grado di modificare una prefabbricazione una volta e applicando le modifiche su più oggetti / scene
- Creazione di oggetti duplicati con modifiche minori, pur avendo gli elementi comuni modificabili da un prefabbricato di base
- Creazione istantanea di GameObjects in fase di runtime

C'è una regola empirica in Unity che dice "tutto dovrebbe essere prefabbricato". Anche se questa è probabilmente un'esagerazione, incoraggia il riutilizzo del codice e la costruzione di GameObjects in un modo riutilizzabile, che è sia efficiente in termini di memoria che di buon design.

Creazione di prefabbricati

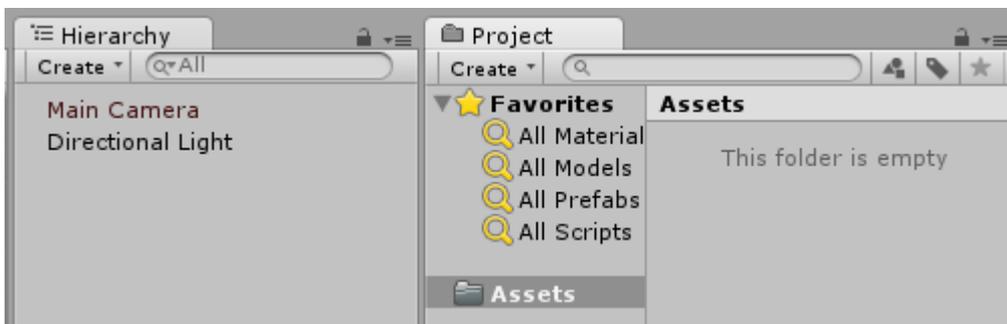
Per creare un prefabbricato, trascina un oggetto di gioco dalla gerarchia delle scene nella cartella o sottocartella **Risorse** :





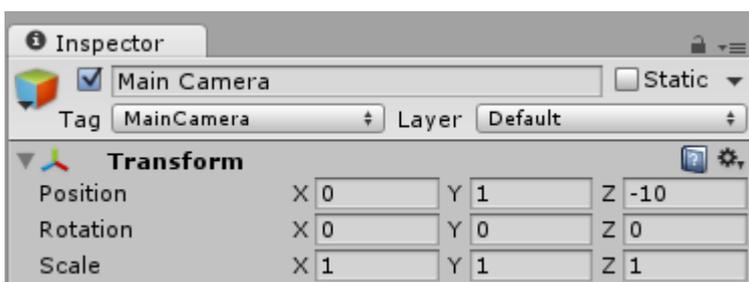
Il nome dell'oggetto del gioco diventa blu, a indicare che è **collegato a un prefabbricato** . Ora questo oggetto è **un'istanza prefabbricata** , proprio come un'istanza di un oggetto di una classe.

Un prefabbricato può essere cancellato dopo l'istanziamento. In tal caso il nome dell'oggetto di gioco precedentemente collegato diventa rosso:

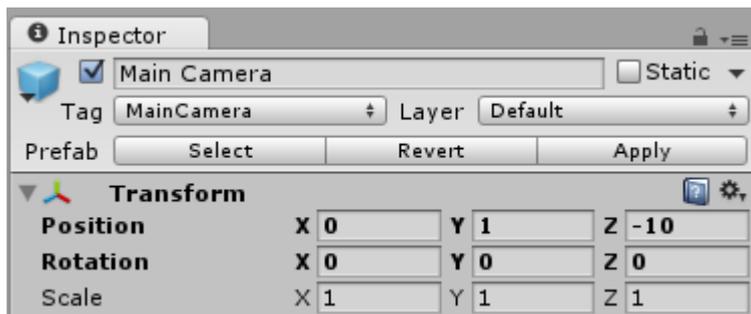


Ispettore prefabbricato

Se selezioni un prefabbricato nella vista della gerarchia, noterai che l'ispettore è leggermente diverso da un normale oggetto di gioco:



VS



Le proprietà grassetto significano che i loro valori differiscono dai valori prefabbricati. È possibile modificare qualsiasi proprietà di un prefabbricato istanziato senza influire sui valori prefabbricati originali. Quando un valore viene modificato in un'istanza prefabbricata, diventa in grassetto e tutte le successive modifiche dello stesso valore nel prefabbricato non si rifletteranno nell'istanza modificata.

È possibile ripristinare i valori prefabbricati originali facendo clic sul pulsante **Ripristina**, il quale conterrà anche le modifiche dei valori nell'istanza. Inoltre, per ripristinare un singolo valore, puoi fare clic con il pulsante destro del mouse e premere **Ripristina valore su Prefab**. Per ripristinare un componente, fai clic con il pulsante destro del mouse e premi **Ripristina prefabbricato**.

Facendo clic sul pulsante **Applica** sovrascrive i valori delle proprietà prefabbricate con i valori delle proprietà dell'oggetto di gioco corrente. Non c'è il pulsante "Annulla" o conferma la finestra di dialogo, quindi mantieni questo pulsante con cura.

Seleziona pulsante evidenzia collegato prefabbricato nella struttura delle cartelle del progetto.

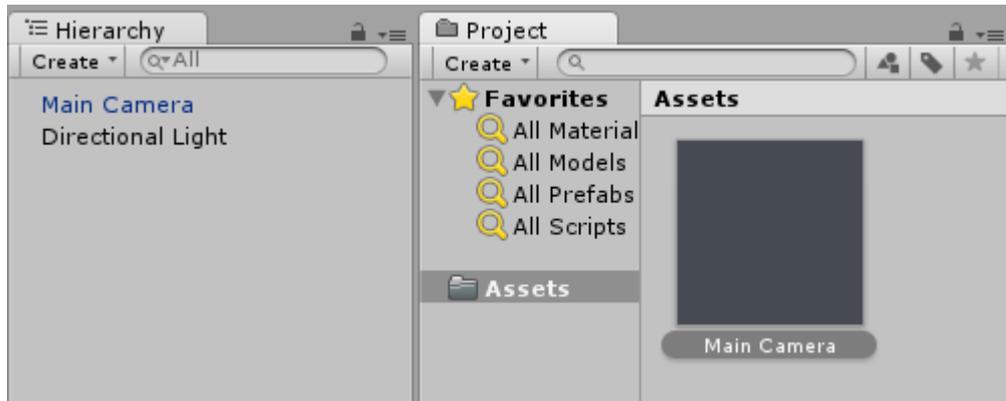
Prefabbricati istanziati

Esistono 2 modi di **creare** istanze prefabbricate: in fase di **progettazione** o in **fase di esecuzione**.

Creazione di istanze temporali

Istanziare i prefabbricati in fase di progettazione è utile per posizionare visivamente più istanze dello stesso oggetto (ad es. *Posizionare alberi quando si progetta un livello del gioco*).

- Per istanziare visivamente un prefabbricato trascinalo dalla vista del progetto alla gerarchia delle scene.



- Se stai scrivendo un'estensione editor , puoi anche creare un'istanza prefabbricata che chiama a `PrefabUtility.InstantiatePrefab()` programmazione il metodo

`PrefabUtility.InstantiatePrefab()` :

```
GameObject gameObject =
  (GameObject)PrefabUtility.InstantiatePrefab(AssetDatabase.LoadAssetAtPath("Assets/MainCamera.prefab",
  typeof(GameObject)));
```

Istanziamento runtime

Istanziare i prefabbricati in fase di esecuzione è utile per creare istanze di un oggetto secondo una logica (es. *Generare un nemico ogni 5 secondi*).

Per istanziare un prefabbricato è necessario un riferimento all'oggetto prefabbricato. Questo può essere fatto avendo un campo `public GameObject` nel tuo script `MonoBehaviour` (e impostandone il valore usando l'ispettore nell'editor di Unity):

```
public class SomeScript : MonoBehaviour {
    public GameObject prefab;
}
```

O inserendo il prefabbricato nella cartella [Risorse](#) e utilizzando `Resources.Load` . `Resources.Load` :

```
GameObject prefab = Resources.Load("Assets/Resources/MainCamera");
```

Una volta che hai un riferimento all'oggetto prefabbricato, puoi istanziarlo usando la funzione `Instantiate` qualsiasi punto del tuo codice (ad esempio *all'interno di un loop per creare più oggetti*):

```
GameObject gameObject = Instantiate<GameObject>(prefab, new Vector3(0,0,0),
Quaternion.identity);
```

Nota: il termine *prefabbricato* non esiste in fase di esecuzione.

Prefabbricati annidati

I prefabbricati annidati non sono disponibili in Unity al momento. È possibile trascinare un prefabbricato in un altro e applicarlo, ma eventuali modifiche sul prefabbricato figlio non verranno applicate a quello annidato.

Ma c'è una soluzione semplice: **devi aggiungere al genitore prefabbricato un semplice script, che istanzia un bambino.**

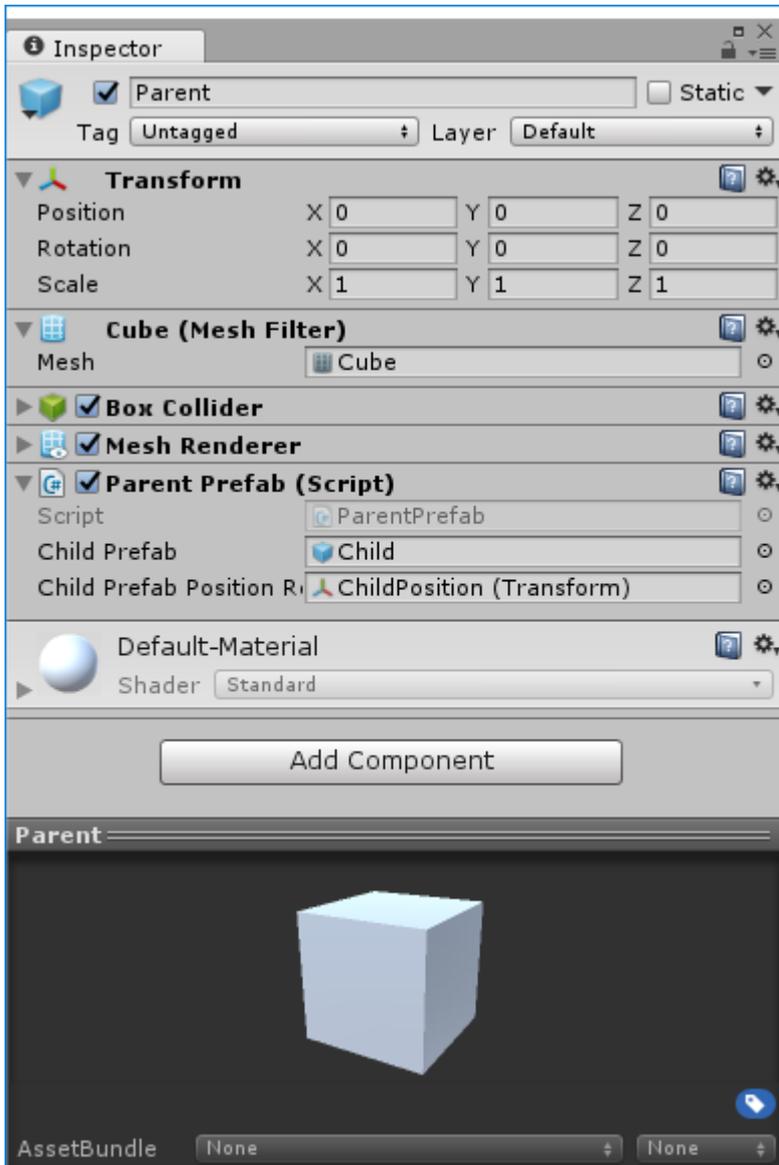
```
using UnityEngine;

public class ParentPrefab : MonoBehaviour {

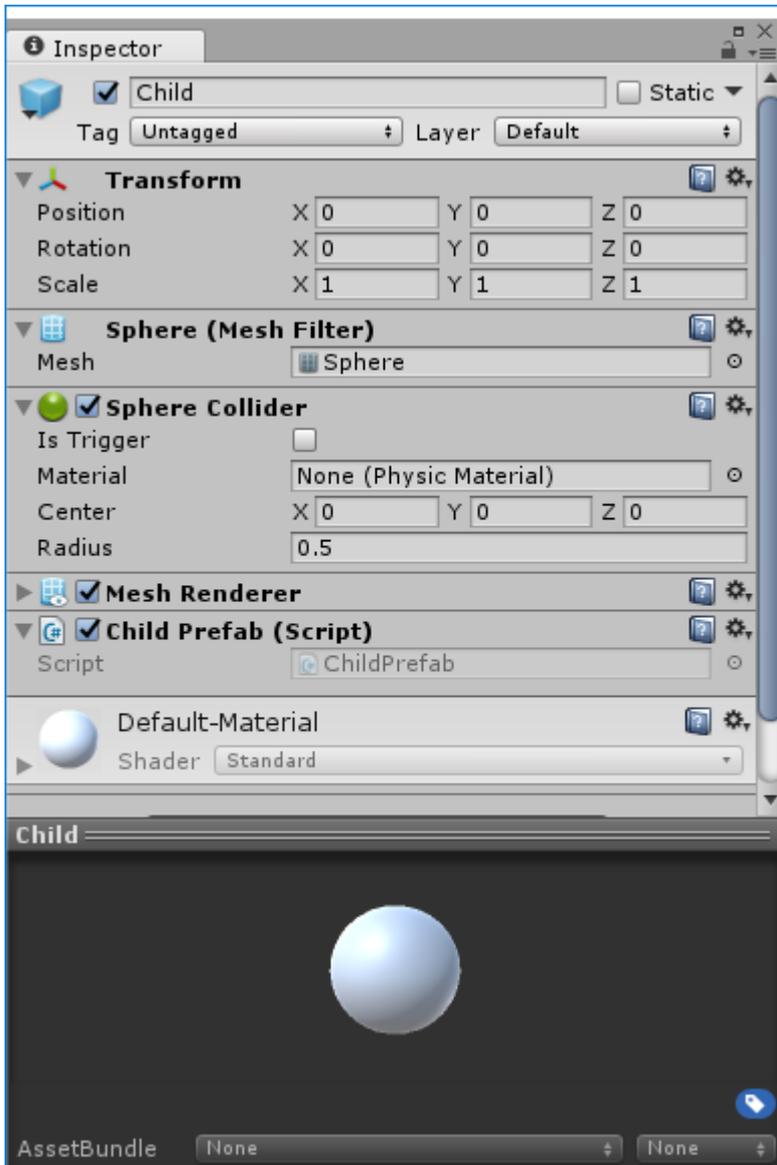
    [SerializeField] GameObject childPrefab;
    [SerializeField] Transform childPrefabPositionReference;

    // Use this for initialization
    void Start () {
        print("Hello, I'm a parent prefab!");
        Instantiate(
            childPrefab,
            childPrefabPositionReference.position,
            childPrefabPositionReference.rotation,
            gameObject.transform
        );
    }
}
```

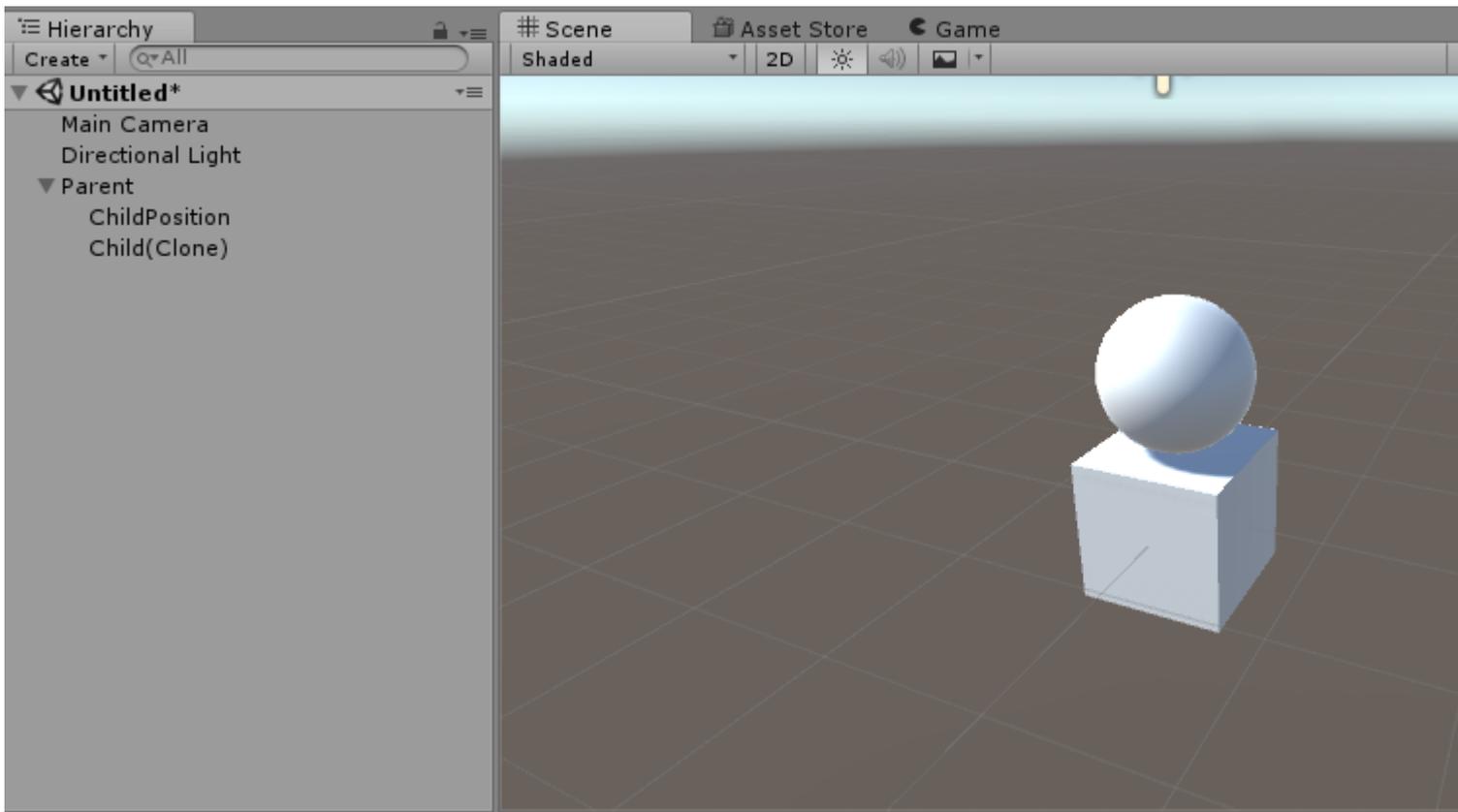
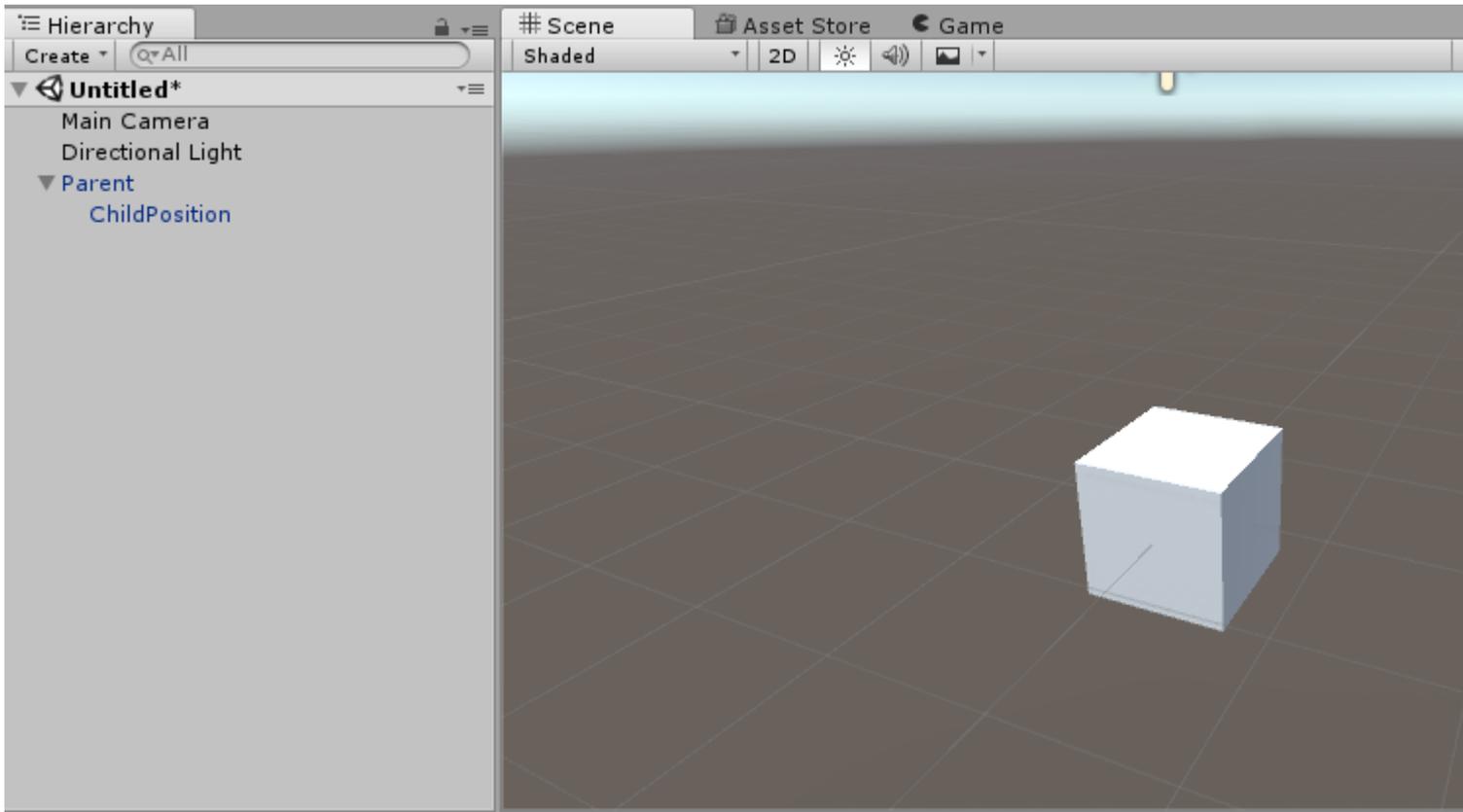
Prefabbricato dei genitori:



Prefabbricato per bambini:



Scena prima e dopo l'inizio:



Leggi prefabbricati online: <https://riptutorial.com/it/unity3d/topic/2133/prefabbricati>

Capitolo 22: quaternions

Sintassi

- Quaternion.LookRotation (Vector3 avanti [, Vector3 su]);
- Quaternion.AngleAxis (angoli flottanti, Vector3 axisOfRotation);
- float angleBetween = Quaternion.Angle (Quaternion rotation1, Quaternion rotation2);

Examples

Introduzione a Quaternion vs Eulero

Gli angoli di Eulero sono "angoli di grado" come 90, 180, 45, 30 gradi. I quaternioni differiscono dagli angoli di Eulero in quanto rappresentano un punto su una sfera unitaria (il raggio è 1 unità). Puoi pensare a questa sfera come una versione 3D del cerchio dell'Unità che impari in trigonometria. I quaternioni differiscono dagli angoli di Eulero in quanto usano numeri immaginari per definire una rotazione 3D.

Anche se questo può sembrare complicato (e probabilmente lo è), Unity ha grandi funzioni incorporate che ti permettono di cambiare tra gli angoli di Eulero e quaternioni, oltre alle funzioni per modificare i quaternioni, senza conoscere una sola cosa sulla matematica che sta dietro di loro.

Conversione tra Eulero e Quaternione

```
// 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 transform's Quaternion angles to Euler angles
Quaternion quaternionAngles = transform.rotation;
Vector3 eulerAngles = quaternionAngles.eulerAngles;
```

Perché usare un Quaternion?

I quaternioni risolvono un problema noto come blocco cardanico. Ciò si verifica quando l'asse di rotazione primario diventa collineare con l'asse di rotazione terziario. Ecco un [esempio visivo](#) @ 2:09

Quaternion Look Rotation

Quaternion.LookRotation(Vector3 forward [, Vector3 up]) creerà una rotazione Quaternion che guarda avanti 'verso il basso' il vettore in avanti e ha l'asse Y allineato con il vettore 'su'. Se il vettore su non viene specificato, verrà utilizzato Vector3.up.

Ruota questo oggetto di gioco per guardare un oggetto di gioco di destinazione

```
// 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;
```

Leggi quaternions online: <https://riptutorial.com/it/unity3d/topic/1782/quaternions>

Capitolo 23: raycast

Parametri

Parametro	Dettagli
origine	Il punto di partenza del raggio nelle coordinate del mondo
direzione	La direzione del raggio
maxDistance	La distanza massima che il raggio dovrebbe controllare per le collisioni
layerMask	Una maschera di livello che viene utilizzata per ignorare selettivamente i Collider durante il casting di un raggio.
queryTriggerInteraction	Specifica dove questa query deve colpire i trigger.

Examples

Raggio di fisica

Questa funzione lancia un raggio dal punto `origin` in direzione `direction` della lunghezza `maxDistance` contro tutti acceleratori nella scena.

La funzione prende la `direction origin maxDistance` e calcola se c'è un collisore di fronte al `GameObject`.

```
Physics.Raycast(origin, direction, maxDistance);
```

Ad esempio, questa funzione stamperà `Hello World` sulla console se c'è qualcosa all'interno di 10 unità del `GameObject` collegato ad esso:

```
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

È possibile utilizzare i raycast per verificare se un AI può camminare senza cadere dal bordo di un livello.

```
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
        }
    }
}
```

In questo esempio la direzione è giusta. La variabile `raycastRightPart` è la parte giusta del personaggio, quindi il raycast avverrà nella parte destra del personaggio. La distanza è 0,6f volte l'altezza del personaggio in modo che il raycast non dia un colpo quando colpisce il terreno che è molto più basso del terreno su cui si trova in quel momento. Assicurati che `LayerMask` sia impostato a terra, altrimenti rileverà anche altri tipi di oggetti.

`RaycastHit2D` stesso è una struttura e non una classe, quindi `hit` non può essere nullo; questo significa che devi controllare il collider di una variabile `RaycastHit2D`.

Incapsulare chiamate Raycast

Avere gli script in modo che chiamino `Raycast` direttamente può portare a problemi se è necessario cambiare le matrici di collisione in futuro, poiché dovrai tenere traccia di ogni campo di `LayerMask` per sistemare le modifiche. A seconda delle dimensioni del tuo progetto, questo potrebbe diventare un'impresa enorme.

Incapsulare le chiamate a `Raycast` può semplificarti la vita.

Guardandolo da un principio [SoC](#), un oggetto di gioco non dovrebbe davvero conoscere o preoccuparsi di `LayerMask`. Ha solo bisogno di un metodo per scansionare ciò che lo circonda. Se il risultato di `Raycast` restituisce questo o quello non dovrebbe importare nell'oggetto di gioco. Dovrebbe agire solo sulle informazioni che riceve e non fare ipotesi sull'ambiente in cui esiste.

Un modo per avvicinarsi a questo è spostare il valore `LayerMask` in istanze [ScriptableObject](#) e utilizzarli come una forma di servizi raycast da iniettare negli script.

```
// 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);
    }
}
}
```

Ciò consente di creare numerosi servizi raycast, tutti con combinazioni LayerMask diverse per situazioni diverse. Potresti averne uno che colpisce solo i collideri di terra, e un altro che colpisce collideri di terra e piattaforme a senso unico.

Se è necessario apportare modifiche drastiche alle impostazioni di LayerMask, è sufficiente aggiornare queste risorse RaycastService.

Ulteriori letture

- [Inversione di controllo](#)
- [Iniezione di dipendenza](#)

Leggi raycast online: <https://riptutorial.com/it/unity3d/topic/2826/raycast>

Capitolo 24: Realtà virtuale (VR)

Examples

Piattaforme VR

Ci sono due piattaforme principali in VR, una è la piattaforma mobile, come **Google Cardboard** , **Samsung GearVR** , l'altra è la piattaforma PC, come **HTC Vive**, **Oculus**, **PS VR** ...

Unity supporta ufficialmente **Oculus Rift** , **Google Carboard** , **Steam VR** , **Playstation VR** , **Gear VR** e **Microsoft Hololens** .

La maggior parte delle piattaforme ha il proprio supporto e sdk. Di solito, è necessario scaricare il sdk come estensione in primo luogo per l'unità.

SDK:

- [Google Cardboard](#)
- [Piattaforma Daydream](#)
- [Samsung GearVR](#) (integrato da Unity 5.3)
- [Oculus Rift](#)
- [HTC Vive / Open VR](#)
- [Microsoft Hololens](#)

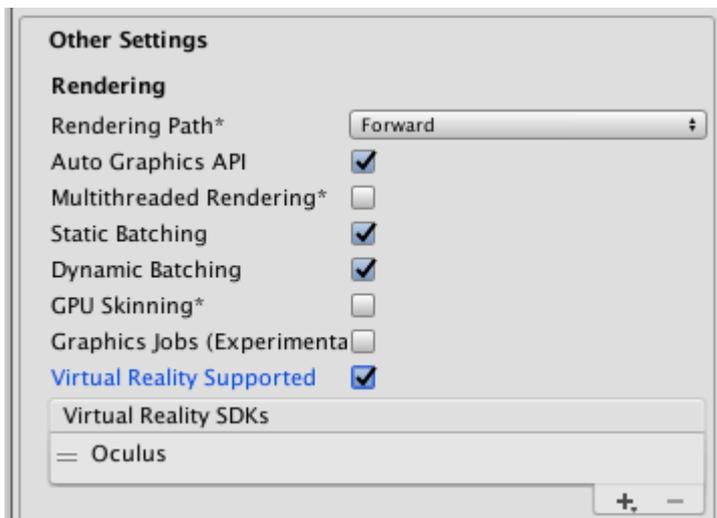
Documentazione:

- [Google Cardboard / Daydream](#)
- [Samsung GearVR](#)
- [Oculus Rift](#)
- [HTC Vive](#)
- [Microsoft Hololens](#)

Abilitazione del supporto VR

In Unity Editor, apri **Impostazioni Player** (Modifica> Impostazioni progetto> Player).

In **Altre impostazioni** , selezionare *Realtà virtuale supportata* .



Aggiungi o rimuovi dispositivi VR per ciascun target di build nell'elenco *SDK di realtà virtuale* sotto la casella di controllo.

Hardware

C'è una dipendenza hardware necessaria per un'applicazione VR, che di solito dipende dalla piattaforma che stai creando. Esistono 2 categorie generali per i dispositivi hardware in base alle loro capacità di movimento:

1. 3 DOF (gradi di libertà)
2. 6 DOF (gradi di libertà)

3 DOF significa che il movimento del display testa-testa (HMD) è costretto a operare in 3 dimensioni che ruotano attorno ai tre assi ortogonali centrati sul centro di gravità dell'HMD: gli assi longitudinale, verticale e orizzontale. Il movimento attorno all'asse longitudinale è chiamato rollio, il movimento attorno all'asse laterale è chiamato passo e il movimento intorno all'asse perpendicolare è chiamato imbardata, principi simili che governano il movimento di qualsiasi oggetto in movimento come un aereo o un'auto, il che significa che anche se sarai in grado di vedere in tutte le direzioni X, Y, Z dal movimento del tuo HMD nell'ambiente virtuale, ma non saresti in grado di muovere o toccare nulla (il movimento da parte di un controller bluetooth aggiuntivo non è lo stesso).

Tuttavia, 6 DOF consente un'esperienza in scala stanza in cui è anche possibile spostarsi attorno all'asse X, Y e Z oltre ai movimenti di rollio, beccheggio e imbardata attorno al proprio centro di gravità, quindi il 6 grado di libertà.

Attualmente una VR su scala ambiente facilitata per 6 DOF richiede elevate prestazioni di calcolo con una scheda grafica e RAM di fascia alta che probabilmente non otterrete dai laptop standard e richiederà un computer desktop con prestazioni ottimali e almeno almeno 6 piedi x 6 piedi spazio libero, mentre un'esperienza di 3 DOF può essere raggiunta solo da uno smartphone standard con un giroscopio integrato (che è integrato nella maggior parte dei moderni telefoni intelligenti che costano circa \$ 200 o più).

Alcuni dispositivi comuni disponibili sul mercato oggi sono:

- [Oculus Rift](#) (6 DOF)
- [HTC Vive](#) (6 DOF)
- [Daydream](#) (3 DOF)
- [Gear VR con tecnologia Oculus](#) (3 DOF)
- [Google Cardboard](#) (3 DOF)

Leggi Realtà virtuale (VR) online: <https://riptutorial.com/it/unity3d/topic/5787/realta-virtuale--vr->

Capitolo 25: risorse

Examples

introduzione

Con la classe `Resources` è possibile caricare dinamicamente asset che non fanno parte della scena. È molto utile quando devi utilizzare le risorse su richiesta, ad esempio localizzare audio, testi, ecc. [Multilingue](#).

Le risorse devono essere collocate in una cartella denominata **Risorse**. È possibile avere più cartelle di risorse distribuite lungo la gerarchia del progetto. La classe `Resources` ispezionerà tutte le cartelle `Risorse` che potresti avere.

Ogni risorsa inserita nelle `Risorse` sarà inclusa nella compilazione anche se non è referenziata nel tuo codice. Quindi non inserire risorse in `Risorse` indiscriminatamente.

```
//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 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();
    }
}
```

Risorse 101

introduzione

Unity ha alcune cartelle "appositamente denominate" che consentono una varietà di usi. Una di queste cartelle è chiamata "Risorse"

La cartella "Risorse" è uno dei soli due modi per caricare le risorse in fase di runtime in Unity (l'altra è [AssetBundles \(Unity Docs\)](#))

La cartella "Risorse" può risiedere ovunque all'interno della cartella `Risorse` e puoi avere più cartelle denominate `Risorse`. Il contenuto di tutte le cartelle "Risorse" viene unito durante la compilazione.

Il modo principale per caricare un asset da una cartella Risorse è utilizzare la funzione [Resources.Load](#) . Questa funzione accetta un parametro stringa che consente di specificare il percorso del file **relativo** alla cartella Risorse. Si noti che **NON** è necessario specificare le estensioni di file durante il caricamento di una risorsa

```
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));
    }
}
```

Gli oggetti che sono composti da più oggetti possono anche essere caricati da Risorse. Esempi sono tali oggetti sono modelli 3D con trame in forno, o un folletto multiplo.

```
//This example will load a multiple sprite texture from Resources named "A_Multiple_Sprite"
var sprites = Resources.LoadAll("A_Multiple_Sprite") as Sprite[];
```

Mettere tutto insieme

Ecco una delle mie classi di supporto che uso per caricare tutti i suoni per qualsiasi gioco. Puoi allegarlo a qualsiasi GameObject in una scena e caricherà i file audio specificati dalla cartella "Risorse / Suoni"

```
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;
        }
    }
}
```

Note finali

1. Unity è intelligente quando si tratta di includere risorse nella tua build. Qualsiasi risorsa non serializzata (cioè utilizzata in una scena inclusa in una build) è esclusa da una build. TUTTAVIA, questo NON si applica a nessun bene all'interno della cartella Risorse. Pertanto, non esagerare nell'aggiungere risorse a questa cartella
2. Le risorse caricate utilizzando `Resources.Load` o `Resources.LoadAll` possono essere scaricate in futuro utilizzando `Resources.UnloadUnusedAssets` o `Resources.UnloadAsset`

Leggi risorse online: <https://riptutorial.com/it/unity3d/topic/4070/risorse>

Capitolo 26: ScriptableObject

Osservazioni

ScriptableObjects con AssetBundles

Prestare attenzione quando si aggiungono prefabbricati a AssetBundles se contengono riferimenti a ScriptableObjects. Poiché ScriptableObjects è essenzialmente un asset, Unity crea dei duplicati prima di aggiungerli ad AssetBundles, il che potrebbe comportare comportamenti indesiderati durante il runtime.

Quando si carica un GameObject da un AssetBundle, potrebbe essere necessario reinserire le risorse ScriptableObject negli script caricati, sostituendo quelli in bundle. Vedi [Iniezione di dipendenza](#)

Examples

introduzione

ScriptableObjects sono oggetti serializzati che non sono legati a scene o oggetti di gioco come i MonoBehaviours. Per dirla in un modo, sono dati e metodi legati a file di risorse all'interno del progetto. Queste risorse ScriptableObject possono essere passate a MonoBehaviours o ad altri oggetti Scriptable, dove è possibile accedere ai loro metodi pubblici.

A causa della loro natura come risorse serializzate, costituiscono eccellenti classi di gestori e origini dati.

Creazione di risorse ScriptableObject

Di seguito è riportata una semplice implementazione di 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);
    }
}
```

```
}  
}
```

Aggiungendo l'attributo `CreateAssetMenu` alla classe, Unity lo elencherà nel sottomenu **Risorse / Crea** . In questo caso è in **Attività / Crea / StackOverflow / Esempi** .

Una volta create, le istanze di `ScriptableObject` possono essere passate ad altri script e `ScriptableObjects` tramite Inspector.

```
using UnityEngine;  
  
public class SampleScript : MonoBehaviour {  
  
    [SerializeField]  
    MyScriptableObject myScriptableObject;  
  
    void OnEnable()  
    {  
        myScriptableObject.HelloWorld();  
    }  
}
```

Crea istanze `ScriptableObject` tramite codice

Crei nuove istanze `ScriptableObject.CreateInstance<T>()` tramite `ScriptableObject.CreateInstance<T>()`

```
T obj = ScriptableObject.CreateInstance<T>();
```

Dove `T` estende `ScriptableObject` .

Non creare `ScriptableObjects` chiamando i loro costruttori, es. `new ScriptableObject()` .

La creazione di `ScriptableObjects` tramite codice durante il runtime è richiesta raramente perché il loro uso principale è la serializzazione dei dati. Potresti anche usare le lezioni standard a questo punto. È più comune quando si eseguono script delle estensioni degli editor.

`ScriptableObjects` sono serializzati nell'editor anche in `PlayMode`

Prestare particolare attenzione quando si accede a campi serializzati in un'istanza `ScriptableObject`.

Se un campo è contrassegnato come `public` o serializzato tramite `SerializeField` , la modifica del suo valore è permanente. Non si resettano quando si esce dalla modalità di gioco come fanno i `MonoBehaviours`. Questo può essere utile a volte, ma può anche creare confusione.

Per questo motivo è meglio rendere i campi serializzati di sola lettura ed evitare del tutto i campi pubblici.

```
public class MyScriptableObject : ScriptableObject  
{
```

```
[SerializeField]
int mySerializedValue;

public int MySerializedValue
{
    get { return mySerializedValue; }
}
}
```

Se si desidera memorizzare valori pubblici in `ScriptableObject` che vengono ripristinati tra le sessioni di gioco, considerare l'utilizzo del seguente modello.

```
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; }
    }
}
```

Trova `ScriptableObjects` esistente durante il runtime

Per trovare `ScriptableObjects` *attivo* durante il runtime, è possibile utilizzare

```
Resources.FindObjectsOfTypeAll()
```

```
T[] instances = Resources.FindObjectsOfTypeAll<T>();
```

Dove `T` è il tipo di istanza `ScriptableObject` che stai cercando. *Attivo* significa che è stato caricato in memoria in una qualche forma prima.

Questo metodo è molto lento, quindi ricorda di memorizzare nella cache il valore restituito ed evita di chiamarlo frequentemente. Fare riferimento a `ScriptableObjects` direttamente negli script dovrebbe essere l'opzione preferita.

Suggerimento: puoi mantenere le tue raccolte di istanze per ricerche più veloci.

`OnEnable()` `ScriptableObjects` si registri su una raccolta condivisa durante `OnEnable()`.

Leggi `ScriptableObject` online: <https://riptutorial.com/it/unity3d/topic/3434/scriptableObject>

Capitolo 27: Singletons in Unity

Osservazioni

Mentre ci sono scuole di pensiero che fanno argomentazioni convincenti per cui l'uso non vincolato di Singletons è una cattiva idea, ad esempio [Singleton su gameprogrammingpatterns.com](#), ci sono occasioni in cui potresti voler mantenere un GameObject in Unity su più Scene (ad esempio per musica di sottofondo senza interruzioni) assicurando che non esista più di una istanza; un caso d'uso perfetto per un Singleton.

Aggiungendo questo script a un oggetto GameObject, una volta che è stato istanziato (ad esempio includendolo ovunque in una scena) rimarrà attivo in tutte le scene e solo un'istanza sarà mai esistita.

Le istanze [ScriptableObject](#) ([UnityDoc](#)) forniscono un'alternativa valida a Singletons per alcuni casi d'uso. Anche se non impongono implicitamente la regola dell'istanza singola, mantengono il loro stato tra le scene e giocano bene con il processo di serializzazione Unity. Promuovono anche [Inversion of Control](#) mentre le dipendenze vengono [iniettate tramite l'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();
    }
}
```

Ulteriori letture

- [Implementazione Singleton in C #](#)

Examples

Implementazione utilizzando RuntimeInitializeOnLoadMethodAttribute

Da **Unity 5.2.5** è possibile utilizzare [RuntimeInitializeOnLoadMethodAttribute](#) per eseguire la logica di inizializzazione bypassando l'ordine di esecuzione di [MonoBehaviour](#). Fornisce un modo per creare un'implementazione più pulita e 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);
    }
}
```

L'ordine di esecuzione risultante:

1. `GameDirector.OnRuntimeMethodLoad()` **avviato ...**
2. `GameDirector.Awake()`
3. `GameDirector.OnRuntimeMethodLoad()` **completato.**
4. `OtherMonoBehaviour1.Awake()`
5. `OtherMonoBehaviour2.Awake()` , **ecc.**

Un semplice Singleton MonoBehaviour in Unity C

In questo esempio, un'istanza statica privata della classe viene dichiarata all'inizio.

Il valore di un campo statico è condiviso tra le istanze, quindi se viene creata una nuova istanza di

questa classe, `if` troverà un riferimento al primo oggetto Singleton, distruggendo la nuova istanza (o il suo oggetto di gioco).

```
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
}
```

Advanced Unity Singleton

Questo esempio combina più varianti di singleton `MonoBehaviour` trovati su Internet in uno e consente di modificarne il comportamento in base ai campi statici globali.

Questo esempio è stato testato utilizzando Unity 5. Per utilizzare questo singleton, è sufficiente estenderlo come segue: `public class MySingleton : Singleton<MySingleton> {}`. Potrebbe anche essere necessario sostituire `AwakeSingleton` per usarlo al posto del solito `Awake`. Per ulteriori modifiche, modificare i valori predefiniti dei campi statici come descritto di seguito.

1. Questa implementazione utilizza l'attributo [DisallowMultipleComponent](#) per mantenere un'istanza per `GameObject`.
2. Questa classe è astratta e può essere estesa solo Contiene anche un metodo virtuale `AwakeSingleton` che deve essere sovrascritto anziché implementare il normale `Awake`.
3. Questa implementazione è thread-safe.
4. Questo singleton è ottimizzato. Utilizzando il flag `instantiated` invece del controllo nullo dell'istanza, evitiamo il sovraccarico derivante dall'implementazione dell'operatore `==` di Unity. ([Leggi di più](#))
5. Questa implementazione non consente alcuna chiamata all'istanza singleton quando sta per essere distrutta da Unity.
6. Questo singleton viene fornito con le seguenti opzioni:
 - `FindInactive` : se cercare altre istanze di componenti dello stesso tipo associate a `GameObject` inattivo.

- **Persist** : se mantenere il componente vivo tra le scene.
- **DestroyOthers** : se distruggere qualsiasi altro componente dello stesso tipo e mantenerne uno solo.
- **Lazy** : se impostare l'istanza singleton "al volo" (in `Awake`) o solo "su richiesta" (quando viene chiamato `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 destroy 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;
}
}
}

```

Implementazione di Singleton attraverso la classe base

Nei progetti che presentano più classi di singleton (come spesso accade), può essere pulito e

comodo astrarre il comportamento di singleton in una classe di 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 può quindi implementare il modello singleton estendendo MonoBehaviourSingleton. Questo approccio consente di utilizzare il modello con un ingombro minimo sul Singleton stesso:

```
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>();
    }
}
```

Si noti che uno dei vantaggi del modello singleton è che si può accedere staticamente a un riferimento all'istanza:

```
// Logs: String Instance
Debug.Log(SingletonImplementation.Instance.Text);
```

Tenete a mente però, questa pratica dovrebbe essere minimizzata al fine di ridurre l'accoppiamento. Questo approccio ha anche un leggero costo di prestazioni dovuto all'uso del dizionario, ma poiché questa raccolta può contenere solo un'istanza di ogni classe di singleton, il compromesso in termini di principio di DRY (non ripetersi), leggibilità e la convenienza è piccola.

Singleton Pattern che utilizza il sistema Entity-Component di Unitys

L'idea di base è usare GameObjects per rappresentare singleton, che ha molteplici vantaggi:

- Mantiene la complessità al minimo ma supporta concetti come l'iniezione di dipendenza
- Singletons hanno un normale ciclo di vita di Unity come parte del sistema Entity-Component
- Singletons può essere pigro caricato e memorizzato nella cache dove necessario (ad esempio nei cicli di aggiornamento)
- Nessun campo statico necessario
- Non è necessario modificare MonoBehaviours / Componenti esistenti per usarli come Singletons
- Facile da resettare (basta distruggere il Singletons GameObject), sarà pigro nuovamente caricato al prossimo utilizzo
- Facile da iniettare mock (basta inicializzarlo con la simulazione prima di usarlo)
- Ispezione e configurazione usando l'editor di Unity normale e possono accadere già nel momento dell'editor ([Schermata di un Singleton accessibile nell'editor di Unity](#))

Test.cs (che utilizza l'esempio 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 (che contiene un esempio e la classe Singleton effettiva):

```
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 MonoBehaviour 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;
    }
}
}

```

I due metodi di estensione per GameObject sono utili anche in altre situazioni, se non ne hai bisogno spostali all'interno della classe Singleton e rendili privati.

Classe Singleton basata su MonoBehaviour & ScriptableObject

La maggior parte degli esempi Singleton usa MonoBehaviour come classe base. Il principale svantaggio è che questa classe Singleton vive solo durante il periodo di esecuzione. Questo ha alcuni inconvenienti:

- Non c'è modo di modificare direttamente i campi Singleton oltre a modificare il codice.
- Nessun modo per memorizzare un riferimento ad altre risorse su Singleton.
- Impossibile impostare il singleton come destinazione di un evento UI Unity. Finisco per utilizzare ciò che chiamo "Componenti proxy" che la sua unica proposta è di avere 1 metodi di linea che chiamano "GameManager.Instance.SomeGlobalMethod ()".

Come notato nelle osservazioni, esistono implementazioni che tentano di risolvere questo problema utilizzando ScriptableObjects come classe base, ma perdono i benefici del tempo di esecuzione del MonoBehaviour. Questa implementazione risolve questi problemi utilizzando un oggetto ScriptableObject come classe base e un MonoBehaviour associato durante il runtime:

- È un bene quindi le sue proprietà possono essere aggiornate sull'editor come qualsiasi altra risorsa Unity.
- Funziona bene con il processo di serializzazione Unity.
- È possibile assegnare riferimenti al singleton ad altre risorse dall'editor (le dipendenze vengono iniettate tramite l'editor).

- Gli eventi Unity possono chiamare direttamente i metodi su Singleton.
- Puoi chiamarlo da qualsiasi posizione nella base di codice usando "SingletonClassName.Instance"
- Ha accesso agli eventi e ai metodi del tempo di esecuzione di MonoBehaviour come: Aggiornamento, Sveglia, Inizio, FixedUpdate, StartCoroutine, ecc.

```

/*****
 * 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 UnityEngine.ScriptableObjectNamespace;

public class SingletonScriptableObject<T> :
    UnityEngine.ScriptableObjectNamespace.BehaviourScriptableObject where T :
    UnityEngine.ScriptableObjectNamespace.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() { }
}
}
}

```

Qui c'è un esempio di classe Singleton di GameManager che utilizza SingletonScriptableObject (con molti commenti):

```

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 MonoBehaviour 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 attached 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  
*/
```

Leggi Singletons in Unity online: <https://riptutorial.com/it/unity3d/topic/2137/singletons-in-unity>

Capitolo 28: Sistema audio

introduzione

Questa è una documentazione sulla riproduzione audio in Unity3D.

Examples

Classe audio - Riproduci 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();
    }
}
```

Leggi Sistema audio online: <https://riptutorial.com/it/unity3d/topic/8064/sistema-audio>

Capitolo 29: Sistema di input

Examples

Lettura della chiave Stampa e differenza tra `GetKey`, `GetKeyDown` e `GetKeyUp`

L'input deve essere letto dalla funzione di aggiornamento.

Riferimento per tutti i [codici di accesso](#) disponibili.

1. Leggere il tasto premere con `Input.GetKey` :

`Input.GetKey` restituirà **ripetutamente** `true` mentre l'utente tiene premuto il tasto specificato. Questo può essere usato per sparare **ripetutamente** un'arma tenendo premuto il tasto specificato. Di seguito è riportato un esempio di attivazione automatica dei proiettili quando viene tenuto premuto il tasto Spazio. Il giocatore non deve premere e rilasciare il tasto più e più volte.

```
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. Premere la pressione del tasto con `Input.GetKeyDown` :

`Input.GetKeyDown` sarà `true` solo **una volta** quando viene premuto il tasto specificato. Questa è la differenza chiave tra `Input.GetKey` e `Input.GetKeyDown` . Un esempio di utilizzo del suo utilizzo è l'attivazione / disattivazione di un'interfaccia utente o di una torcia elettrica o di 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. Premere la pressione del tasto con `Input.GetKeyUp` :

Questo è l'esatto opposto di `Input.GetKeyDown` . È usato per rilevare quando il tasto-pressione viene rilasciato / sollevato. Proprio come `Input.GetKeyDown` , restituisce `true` solo **una volta** . Ad esempio, è possibile `enable` luce quando si tiene premuto il tasto con `Input.GetKeyDown` quindi disabilitare la luce quando viene rilasciato il tasto 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!");
    }
}
}

```

Leggi sensore accelerometro (base)

`Input.acceleration` viene utilizzato per leggere il sensore dell'accelerometro. Restituisce `Vector3` come risultato che contiene i valori dell'asse `x` , `y` e `z` nello spazio 3D.

```

void Update()
{
    Vector3 acclerometerValue = rawAccelValue();
    Debug.Log("X: " + acclerometerValue.x + " Y: " + acclerometerValue.y + " Z: " +
acclerometerValue.z);
}

Vector3 rawAccelValue()
{
    return Input.acceleration;
}

```

Leggi sensore accelerometro (Advance)

L'uso di valori grezzi direttamente dal sensore dell'accelerometro per spostare o ruotare un oggetto `GameObject` può causare problemi quali movimenti a scatti o vibrazioni. Si consiglia di appianare i valori prima di utilizzarli. In effetti, i valori dal sensore dell'accelerometro dovrebbero essere sempre livellati prima dell'uso. Questo può essere ottenuto con un filtro passa-basso ed è qui che `Vector3.Lerp` entra in azione.

```
//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;
}
```

Leggi sensore accelerometro (precisione)

Leggere l'accelerometro Sensore con precisione.

Questo esempio alloca la 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;
    }
}

```

Questo esempio **non** alloca la 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;
}

```

Si noti che questo non è filtrato. Si prega di guardare [qui](#) per come regolare i valori dell'accelerometro per rimuovere il rumore.

Leggi il pulsante del mouse (sinistra, metà, destra) Clic

Queste funzioni sono utilizzate per controllare i clic del pulsante del mouse.

- `Input.GetMouseButton(int button);`
- `Input.GetMouseButtonDown(int button);`
- `Input.GetMouseButtonUp(int button);`

Prendono tutti lo stesso parametro.

- 0 = Clic del mouse sinistro.
- 1 = tasto destro del mouse.
- 2 = clic del mouse centrale.

`GetMouseButton` viene utilizzato per rilevare quando il pulsante del mouse viene *tenuto* premuto *continuamente*. Restituisce `true` mentre il pulsante del mouse specificato viene tenuto premuto.

```

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` viene utilizzato per rilevare quando c'è il clic del mouse. Restituisce `true` se viene premuto **una volta** . Non ritorna più `true` fino a quando il pulsante del mouse non viene rilasciato e premuto di nuovo.

```

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` viene utilizzato per rilevare quando viene rilasciato il pulsante del mouse specificato. Questo è restituito solo `true` una volta rilasciato il pulsante del mouse specificato. Per tornare di nuovo vero, deve essere premuto e rilasciato di nuovo.

```

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");  
}
```

Leggi Sistema di input online: <https://riptutorial.com/it/unity3d/topic/3413/sistema-di-input>

Capitolo 30: Sistema di interfaccia utente (UI)

Examples

Iscrivendosi all'evento nel codice

Per impostazione predefinita, uno dovrebbe iscriversi all'evento utilizzando l'ispettore, ma a volte è meglio farlo in codice. In questo esempio ci iscriviamo all'evento click di un pulsante per gestirlo.

```
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);
    }
}
```

I componenti dell'interfaccia utente di solito forniscono facilmente l'ascoltatore principale:

- Pulsante: [onClick](#)
- Dropdown : [onValueChanged](#)
- InputField: [onEndEdit](#) , [onValidateInput](#) , [onValueChanged](#)
- Barra di scorrimento: [onValueChanged](#)
- ScrollRect: [onValueChanged](#)
- Slider: [onValueChanged](#)
- Attiva: [onValueChanged](#)

Aggiungere i listener del mouse

A volte, si desidera aggiungere listener su eventi particolari non forniti in modo nativo dai componenti, in particolare eventi del mouse. Per fare ciò, dovrai aggiungerli da solo usando 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." );
}
}

```

Sono possibili varie eventID:

- PointerEnter
- PointerExit
- PointerDown
- PointerUp
- PointerClick
- Trascinare
- Far cadere
- Scorrere
- UpdateSelected
- Selezionare
- Deseleziona
- Mossa
- InitializePotentialDrag
- beginDrag
- EndDrag
- Sottoscrivi
- Annulla

Leggi Sistema di interfaccia utente (UI) online: <https://riptutorial.com/it/unity3d/topic/2296/sistema-di-interfaccia-utente--ui->

Capitolo 31: Sistema di interfaccia utente grafica in modalità immediata (IMGUI)

Sintassi

- vuoto statico pubblico `GUILayout.Label` (testo stringa, opzioni `Params GUILayoutOption []`)
- `public static bool GUILayout.Button` (string text, params `GUILayoutOption [] options`)
- stringa statica pubblica `GUILayout.TextArea` (testo stringa, opzioni `Params GUILayoutOption []`)

Examples

GUILayout

Vecchio strumento di sistema dell'interfaccia utente, ora utilizzato per la prototipazione rapida e semplice o il debug nel gioco.

```
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 funzione **GUILayout** funziona all'interno della funzione **OnGUI** .

Leggi [Sistema di interfaccia utente grafica in modalità immediata \(IMGUI\) online](https://riptutorial.com/it/unity3d/topic/6947/sistema-di-interfaccia-utente-grafica-in-modalita-immediata--imgui-):
<https://riptutorial.com/it/unity3d/topic/6947/sistema-di-interfaccia-utente-grafica-in-modalita-immediata--imgui->

Capitolo 32: Sviluppo multiplatforma

Examples

Definizioni del compilatore

Le definizioni del compilatore eseguono il codice specifico della piattaforma. Usandoli puoi fare piccole differenze tra varie piattaforme.

- Trigger Game Center risultati su dispositivi Apple e risultati di google play su dispositivi Android.
- Cambia le icone nei menu (logo windows in windows, pinguino Linux in Linux).
- Possibilmente hanno una meccanica specifica della piattaforma a seconda della piattaforma.
- E altro ancora...

```
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  
  
}
```

[Un elenco completo delle definizioni del compilatore Unity può essere trovato qui](#)

Organizzazione di metodi specifici per piattaforma per classi parziali

[Le classi parziali](#) forniscono un modo pulito per separare la logica principale degli script dai metodi specifici della piattaforma.

Classi e metodi parziali sono contrassegnati con la parola chiave `partial`. Questo segnala al compilatore di lasciare la classe "aperta" e cercare altri file per il resto dell'implementazione.

```
// ExampleClass.cs  
using UnityEngine;  
  
public partial class ExampleClass : MonoBehaviour  
{  
    partial void PlatformSpecificMethod();  
  
    void OnEnable()  
    {  
        PlatformSpecificMethod();  
    }  
}
```

Ora possiamo creare file per gli script specifici della piattaforma che implementano il metodo parziale. I metodi parziali possono avere parametri (anche `ref`) ma devono restituire `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
```

Se un metodo parziale non è implementato, il compilatore ometterà la chiamata.

Suggerimento: questo modello è utile quando si creano anche metodi specifici dell'editor.

Leggi Sviluppo multiplatforma online: <https://riptutorial.com/it/unity3d/topic/4816/sviluppo-multiplatforma>

Capitolo 33: tag

introduzione

Un tag è una stringa che può essere applicata per contrassegnare i tipi `GameObject`. In questo modo, è più facile identificare determinati oggetti `GameObject` tramite codice.

Un tag può essere applicato a uno o più oggetti di gioco, ma un oggetto di gioco avrà sempre solo un tag. Per impostazione predefinita, il tag "Untagged" è utilizzato per rappresentare un oggetto `GameObject` che non è stato etichettato intenzionalmente.

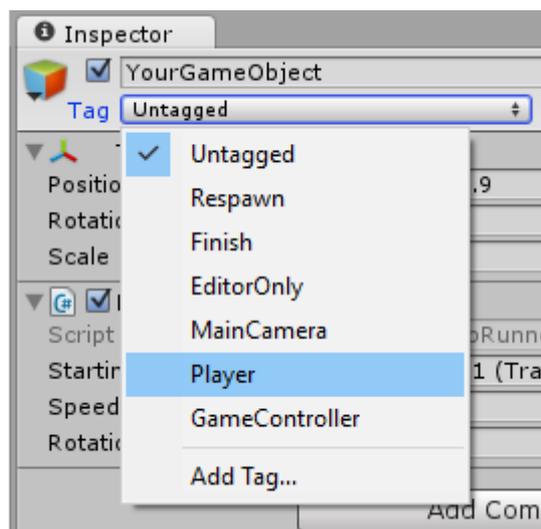
Examples

Creazione e applicazione di tag

I tag vengono in genere applicati tramite l'editor; tuttavia, puoi anche applicare i tag tramite script. Qualsiasi tag personalizzato deve essere creato tramite la finestra *Tag e livelli* prima di essere applicato a un oggetto di gioco.

Impostazione dei tag nell'editor

Con uno o più oggetti di gioco selezionati, puoi selezionare un tag dall'ispettore. Gli oggetti di gioco porteranno sempre un singolo tag; per impostazione predefinita, gli oggetti del gioco verranno contrassegnati come "Non contrassegnati". Puoi anche passare alla finestra *Tag e livelli*, selezionando "Aggiungi tag ..."; tuttavia, è importante notare che questo porta solo alla finestra *Tag e livelli*. Qualsiasi tag che crei non verrà automaticamente applicato all'oggetto del gioco.



Impostazione dei tag tramite Script

Puoi modificare direttamente un tag di oggetti di gioco tramite codice. È importante notare che è *necessario* fornire un tag dall'elenco dei tag correnti; se fornisci un tag che non è già stato creato, si verificherà un errore.

Come illustrato in altri esempi, l'utilizzo di una serie di variabili `static string` anziché la scrittura manuale di ciascun tag può garantire coerenza e affidabilità.

Il seguente script dimostra come possiamo cambiare una serie di tag di oggetti di gioco, usando riferimenti di `static string` per garantire coerenza. Notare che ogni `static string` rappresenta un tag che è già stato creato nella finestra *Tag e livelli*.

```
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;

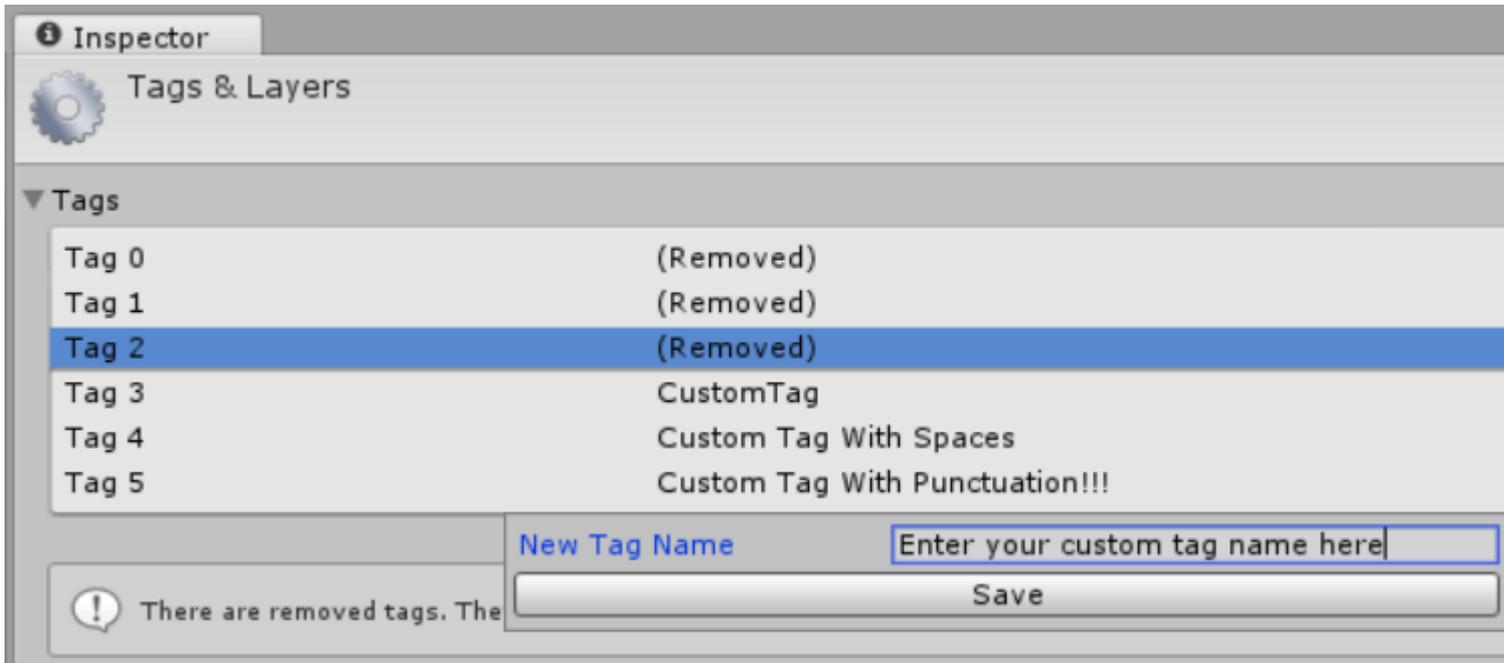
    void Start ()
    {
        // We ensure that the game object this script is attached to
        // is left untagged by using the default "Untagged" tag.
        gameObject.tag = tagUntagged;

        // 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;
        }
    }
}
```

Creazione di tag personalizzati

Indipendentemente dall'impostazione dei tag tramite Inspector o tramite script, i tag *devono* essere dichiarati tramite la finestra *Tag e livelli* prima dell'utilizzo. È possibile accedere a questa finestra selezionando *"Aggiungi tag ..."* dal menu a discesa dei tag degli oggetti di gioco. In alternativa, puoi trovare la finestra sotto **Modifica > Impostazioni progetto > Tag e livelli**.



Basta selezionare il pulsante + , inserire il nome desiderato e selezionare `Salva` per creare un tag. Selezionando il pulsante - si rimuoverà il tag attualmente evidenziato. Si noti che in questo modo, il tag verrà immediatamente visualizzato come "(Rimosso)" e verrà completamente rimosso al prossimo riavvio del progetto.

Selezionando la ruota dentata / ingranaggio dalla parte superiore destra della finestra, potrai ripristinare tutte le opzioni personalizzate. Questo rimuoverà immediatamente tutti i tag personalizzati, insieme a qualsiasi livello personalizzato che puoi avere sotto "Ordinamento livelli" e "Livelli" .

Ricerca di oggetti di gioco per tag:

I tag rendono particolarmente facile localizzare oggetti di gioco specifici. Possiamo cercare un singolo oggetto di gioco o cercare più oggetti.

Trovare un singolo `GameObject`

Possiamo utilizzare la funzione statica `GameObject.FindGameObjectWithTag(string tag)` per cercare singoli oggetti di gioco. È importante notare che, in questo modo, gli oggetti di gioco non vengono interrogati in alcun ordine particolare. Se cerchi un tag che viene utilizzato su più oggetti di gioco nella scena, questa funzione non sarà in grado di garantire *quale* oggetto di gioco viene restituito. Come tale, è più appropriato sapere che solo *un* oggetto di gioco usa tale tag, o quando non siamo preoccupati per l'esatta istanza di `GameObject` che viene restituita.

```
///
```

Trovare una matrice di istanze GameObject

Possiamo utilizzare la funzione statica `GameObject.FindGameObjectsWithTag(string tag)` per cercare *tutti gli* oggetti di gioco che utilizzano un determinato tag. Questo è utile quando vogliamo iterare attraverso un gruppo di oggetti di gioco particolari. Questo può anche essere utile se vogliamo trovare un *singolo* oggetto di gioco, ma potrebbe avere *più* oggetti di gioco che usano lo stesso tag. Come non possiamo garantire l'esatta istanza restituita dal `GameObject.FindGameObjectWithTag(string tag)`, dobbiamo invece recuperare una matrice di tutti i potenziali `GameObject` istanze con `GameObject.FindGameObjectsWithTag(string tag)`, e analizzare ulteriormente la matrice risultante per trovare l'istanza siamo cercando.

```
///
```

Confronto di tag

Quando si confrontano due oggetti `GameObject` per tag, si dovrebbe notare che quanto segue causerebbe un sovraccarico di Garbage Collector ogni volta che viene creata una stringa:

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

Quando si eseguono tali confronti all'interno di `Update ()` e di altri callback di Unity regolari (o di un ciclo), è necessario utilizzare questo metodo di allocazione heap-free:

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

Inoltre è più facile mantenere i tag in una classe statica.

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

Quindi puoi confrontare in modo sicuro

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

in questo modo, le stringhe di tag vengono generate in fase di compilazione e si limitano le implicazioni degli errori di ortografia.

Proprio come mantenere i tag in una classe statica, è anche possibile archivarli in un'enumerazione:

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

e quindi puoi confrontarlo usando il metodo `enum toString()` :

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

Leggi tag online: <https://riptutorial.com/it/unity3d/topic/5534/tag>

Capitolo 34: Trasformazioni

Sintassi

- void Transform.Translate (traduzione Vector3, Spazio relativoTo = Spazio.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 (asse Vector3, angolo flottante, Spazio relativoTo = Spazio.Self)
- void Transform.RotateAround (punto Vector3, asse Vector3, angolo flottante)
- void Transform.LookAt (Trasforma target, Vector3 worldUp = Vector3.up)
- void Transform.LookAt (Vector3 worldPosition, Vector3 worldUp = Vector3.up)

Examples

Panoramica

Le trasformazioni contengono la maggior parte dei dati relativi a un oggetto in unità, inclusi genitore (i), figlio (i), posizione, rotazione e scala. Ha anche funzioni per modificare ciascuna di queste proprietà. Ogni GameObject ha una trasformazione.

Traduzione (spostamento) di un oggetto

```
// 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);
```

Rotazione di un oggetto

```
// 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));
```

Ulteriori informazioni ed esempi possono essere visualizzati nella [documentazione di Unity](#) .

Si noti inoltre che se il gioco utilizza corpi rigidi, la trasformazione non deve essere interagita direttamente (a meno che il corpo rigido non abbia `isKinematic == true`). In tal caso, utilizzare [AddForce](#) o altri metodi simili per agire direttamente sul corpo rigido.

Parenting e bambini

Unity lavora con le gerarchie per mantenere il tuo progetto organizzato. Puoi assegnare agli oggetti un posto nella gerarchia usando l'editor ma puoi farlo anche attraverso il codice.

Parenting

È possibile impostare il genitore di un oggetto con i seguenti metodi

```
var other = GetOtherGameObject();
other.transform.SetParent( transform );
other.transform.SetParent( transform, worldPositionStays );
```

Ogni volta che si imposta un genitore di trasformazioni, manterrà la posizione degli oggetti come una posizione mondiale. Puoi scegliere di rendere questa posizione relativa passando *false* per il parametro *worldPositionStays*.

È anche possibile verificare se l'oggetto è figlio di un'altra trasformazione con il seguente metodo

```
other.transform.IsChildOf( transform );
```

Ottenere un bambino

Poiché gli oggetti possono essere controllati l'un l'altro, puoi anche trovare i bambini nella gerarchia. Il modo più semplice per farlo è utilizzare il seguente metodo

```
transform.Find( "other" );
transform.FindChild( "other" );
```

Nota: FindChild chiama Trova sotto il cofano

Puoi anche cercare bambini più in basso nella gerarchia. Lo fai aggiungendo in un "/" per specificare di andare ad un livello più profondo.

```
transform.Find( "other/another" );
transform.FindChild( "other/another" );
```

Un altro modo di andare a prendere un bambino sta usando il `GetChild`

```
transform.GetChild( index );
```

`GetChild` richiede un numero intero come indice che deve essere inferiore al numero totale di figli

```
int count = transform.childCount;
```

Modifica dell'indice di Sibling

Puoi cambiare l'ordine dei bambini di un GameObject. Puoi farlo per definire l'ordine di disegno dei bambini (supponendo che siano sullo stesso livello Z e lo stesso ordine).

```
other.transform.SetSiblingIndex( index );
```

È anche possibile impostare rapidamente l'indice di pari livello sul primo o sull'ultimo utilizzando i seguenti metodi

```
other.transform.SetAsFirstSibling();  
other.transform.SetAsLastSibling();
```

Staccare tutti i bambini

Se vuoi rilasciare tutti i figli di una trasformazione, puoi farlo:

```
foreach(Transform child in transform)  
{  
    child.parent = null;  
}
```

Inoltre, Unity fornisce un metodo per questo scopo:

```
transform.DetachChildren();
```

Fondamentalmente, sia il looping che `DetachChildren()` impostano i genitori dei bambini di primo livello a null - il che significa che non avranno genitori.

(bambini di prima profondità: le trasformazioni che sono direttamente figli di trasformare)

Leggi Trasformazioni online: <https://riptutorial.com/it/unity3d/topic/2190/trasformazioni>

Capitolo 35: Trovare e collezionare GameObjects

Sintassi

- `public static GameObject Find (nome stringa);`
- `public static GameObject FindGameObjectWithTag (tag stringa);`
- `public static GameObject [] FindGameObjectsWithTag (tag stringa);`
- oggetto statico pubblico `FindObjectOfType (Tipo tipo);`
- oggetto statico pubblico `[] FindObjectsOfType (Type type);`

Osservazioni

Quale metodo usare

Prestare attenzione durante la ricerca di GameObjects in fase di esecuzione, in quanto ciò può essere il consumo di risorse. In particolare: non eseguire `FindObjectOfType` o `Trova` in `Update`, `FixedUpdate` o più in generale in un metodo chiamato una o più volte per frame.

- Chiama i metodi di runtime `FindObjectOfType` e `Find` solo quando necessario
- `FindGameObjectWithTag` ha prestazioni molto buone rispetto ad altri metodi basati su stringhe. Unity mantiene schede separate su oggetti taggati e interroga quelle invece dell'intera scena.
- Per GameObjects "statici" (come elementi dell'interfaccia utente e prefabbricati) creati nell'editor, utilizzare il [riferimento serializzabile GameObject](#) nell'editor
- Mantieni i tuoi elenchi di GameObjects in List o Array che gestisci tu stesso
- In generale, se istanziate molti GameObjects dello stesso tipo date un'occhiata a [Pooling degli oggetti](#)
- Memorizza i risultati della ricerca nella cache per evitare di eseguire ripetutamente i costosi metodi di ricerca.

Andando più a fondo

Oltre ai metodi forniti con Unity, è relativamente facile progettare i propri metodi di ricerca e raccolta.

- In caso di `FindObjectsOfType()`, è possibile che gli script mantengano un elenco di se stessi in una raccolta `static`. È molto più rapido iterare un elenco di oggetti pronto piuttosto che cercare e ispezionare oggetti dalla scena.
- Oppure crea uno script che memorizza le loro istanze in un `Dictionary` basato su stringhe e disponi di un semplice sistema di codifica che puoi espandere.

Examples

Ricerca per nome di GameObject

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

Professionisti	Contro
Facile da usare	Le prestazioni si degradano lungo il numero di oggetti di gioco in scena
	Le stringhe sono <i>referimenti deboli</i> e sospetti di errori dell'utente

Ricerca per tag di GameObject

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

Professionisti	Contro
È possibile cercare sia singoli oggetti che interi gruppi	Le stringhe sono riferimenti deboli e sospetti di errori dell'utente.
Relativamente veloce ed efficiente	Il codice non è portatile in quanto i tag sono codificati in modo sicuro negli script.

Inserito negli script in modalità Modifica

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

Professionisti	Contro
Grande esibizione	La raccolta di oggetti è statica
Codice portatile	Può solo riferirsi a GameObjects dalla stessa scena

Trovare gli oggetti GameObjects tramite MonoBehaviour

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

`FindObjectOfType()` restituisce `null` se non viene trovato alcuno.

Professionisti	Contro
Fortemente tipizzato	Le prestazioni si degradano lungo il numero di oggetti

Professionisti	Contro
di gioco necessari per la valutazione	
È possibile cercare sia singoli oggetti che interi gruppi	

Trova GameObjects per nome dagli oggetti figlio

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

`Find` restituisce `null` se non viene trovato alcuno

Professionisti	Contro
Ambito di ricerca limitato e ben definito	Le stringhe sono riferimenti deboli

Leggi [Trovare e collezionare GameObjects online](https://riptutorial.com/it/unity3d/topic/3793/trovare-e-collezionare-gameobjects):

<https://riptutorial.com/it/unity3d/topic/3793/trovare-e-collezionare-gameobjects>

Capitolo 36: Unity Animation

Examples

Animazione di base per l'esecuzione

Questo codice mostra un semplice esempio di animazione in Unity.

Per questo esempio, dovresti avere 2 clip di animazione; Corri e pigia. Quelle animazioni dovrebbero essere movimenti Stand-in-Place. Una volta selezionate le clip di animazione, crea un controller di animazione. Aggiungi questo controller al giocatore o all'oggetto di gioco che desideri animare.

Apri la finestra Animator dall'opzione Windows. Trascina le 2 clip di animazione nella finestra di Animator e verranno creati 2 stati. Una volta creato, usa la scheda dei parametri di sinistra per aggiungere 2 parametri, entrambi come bool. Chiamane uno come "PerformRun" e altri come "PerformIdle". Impostare "PerformIdle" su true.

Effettua le transizioni dallo stato inattivo a Esegui e Esegui inattivo (fare riferimento all'immagine). Fare clic su Inattività-> Esegui transizione e nella finestra Impostazioni, deselezionare HasExit. Fai lo stesso per l'altra transizione. Per Inattività-> Esegui transizione, aggiungi una condizione: PerformIdle. Per Esegui-> In attesa, aggiungi una condizione: Esegui. Aggiungi lo script C # indicato di seguito all'oggetto del gioco. Dovrebbe essere eseguito con l'animazione usando il pulsante Su e ruotando con i pulsanti Sinistra e Destra.

```
using UnityEngine;
using System.Collections;

public class RootMotion : MonoBehaviour {

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

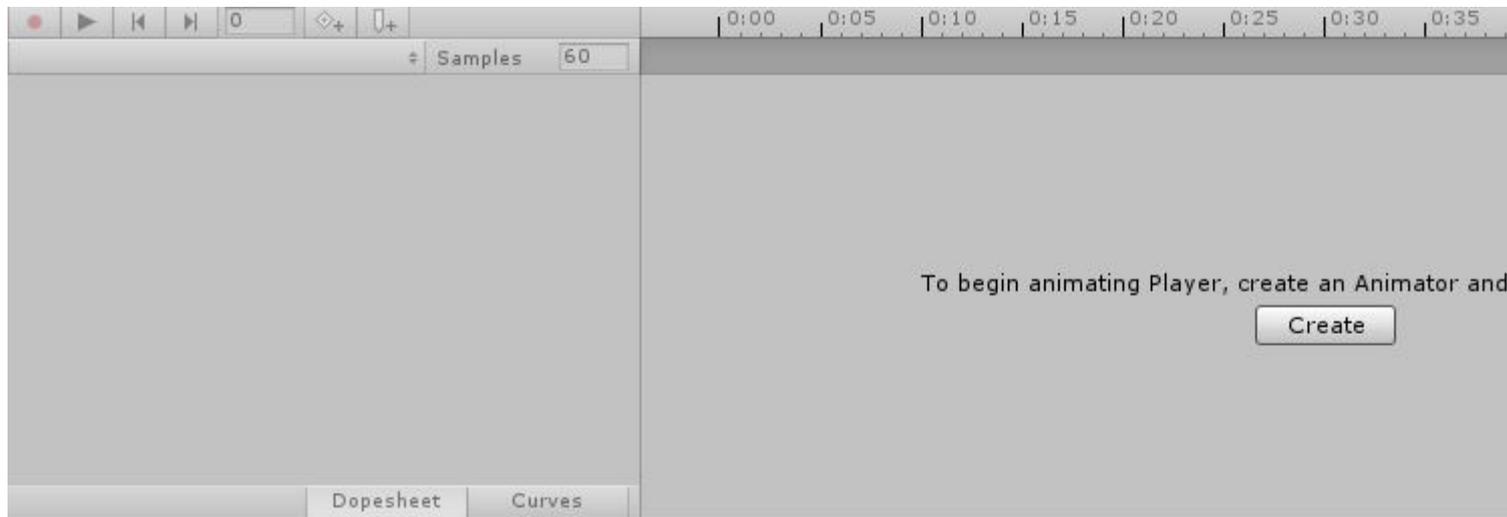
    Animator animator;

    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>();
    }

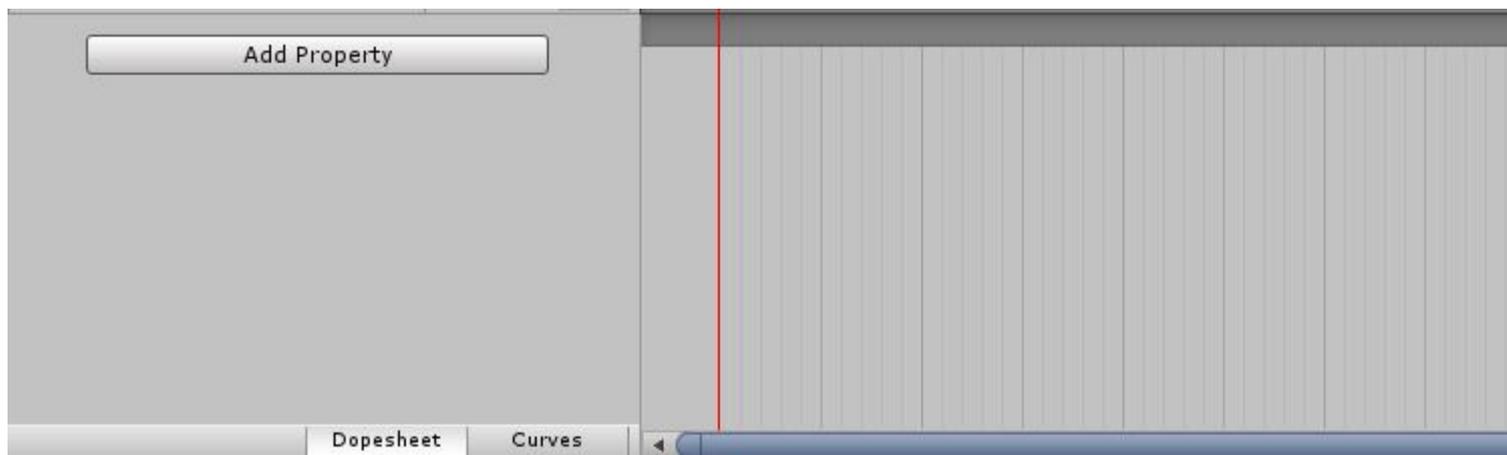
    void Update()
    {
        /**
         * The Update() function will get the bool parameters from the animator state machine and
         set the values provided by the user.
         */
    }
}
```


Nota, i modelli utilizzati in questo esempio sono scaricati da Unity Asset Store. Il lettore è stato scaricato dal seguente link: <https://www.assetstore.unity3d.com/en/#!/content/21874> .

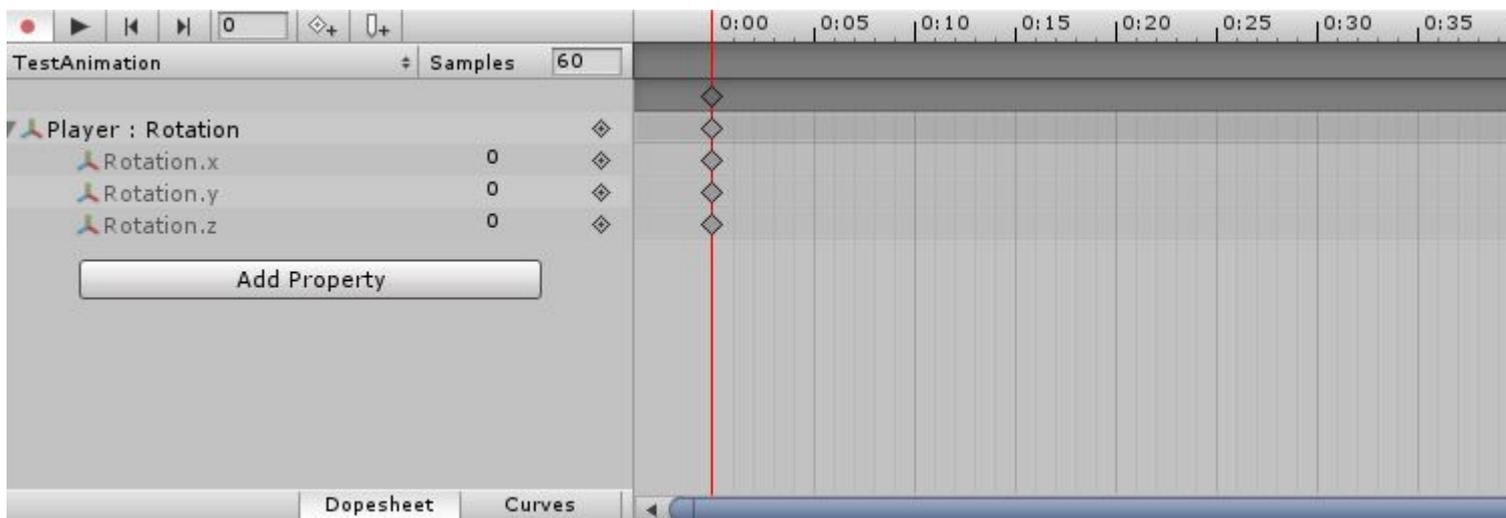
Per creare animazioni, devi prima aprire la finestra Animazione. Puoi aprirlo facendo clic su Finestra e Seleziona animazione o premere Ctrl + 6. Seleziona l'oggetto di gioco al quale desideri applicare la clip di animazione, dalla finestra Gerarchia, quindi fai clic sul pulsante Crea nella finestra Animazione.



Assegna un nome alla tua animazione (come IdlePlayer, SprintPlayer, DyingPlayer ecc.) E salvalo. Ora, dalla finestra di animazione, fare clic sul pulsante Aggiungi proprietà. Questo ti permetterà di cambiare la proprietà dell'oggetto o del giocatore rispetto al tempo. Questo può includere trancie come la rotazione, la posizione e la scala e qualsiasi altra proprietà che è collegata all'oggetto di gioco, ad es. Collider, Mesh Renderer ecc.



Per creare un'animazione in esecuzione per l'oggetto del gioco, è necessario un modello umanoide 3D. È possibile scaricare il modello dal link precedente. Segui i passaggi precedenti per creare una nuova animazione. Aggiungi una proprietà Trasforma e seleziona Rotazione per una gamba del personaggio.



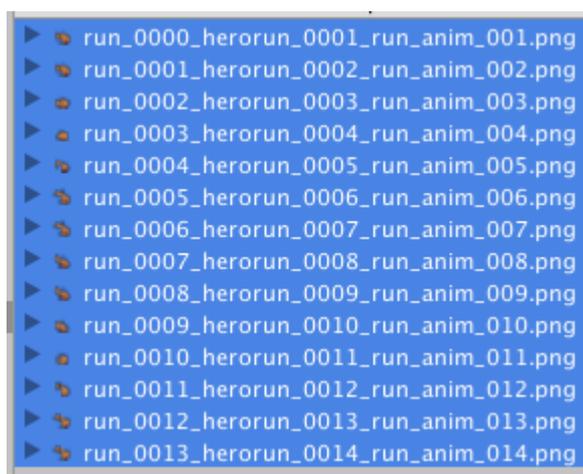
In questo momento, il tuo pulsante Play e i valori di rotazione nella proprietà dell'oggetto del gioco sarebbero diventati rossi. Fare clic sulla freccia a discesa per visualizzare i valori di rotazione X, Y e Z. Il tempo di animazione predefinito è impostato su 1 secondo. Le animazioni utilizzano fotogrammi chiave per interpolare tra valori. Per animare, aggiungi le chiavi in punti diversi nel tempo e modifica i valori di rotazione dalla finestra dell'Inspector. Ad esempio, il valore di rotazione al tempo 0.0s può essere 0.0. Al tempo 0.5s il valore può essere 20.0 per X. Al momento 1.0s il valore può essere 0.0. Possiamo terminare la nostra animazione a 1.0s.

La lunghezza dell'animazione dipende dalle ultime chiavi aggiunte all'animazione. È possibile aggiungere più chiavi per rendere l'interpolazione più fluida.

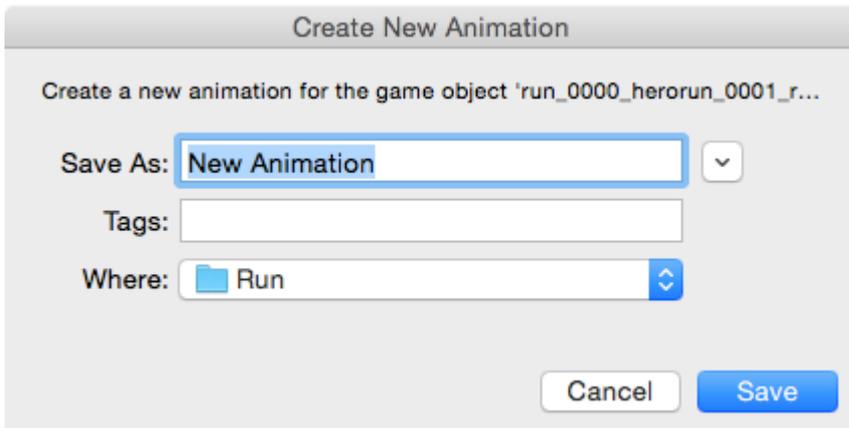
Animazione 2D Sprite

L'animazione Sprite consiste nel mostrare una sequenza esistente di immagini o fotogrammi.

Prima importa una sequenza di immagini nella cartella delle risorse. Crea alcune immagini da zero o scaricali da Asset Store. (Questo esempio utilizza [questa risorsa gratuita](#) .)

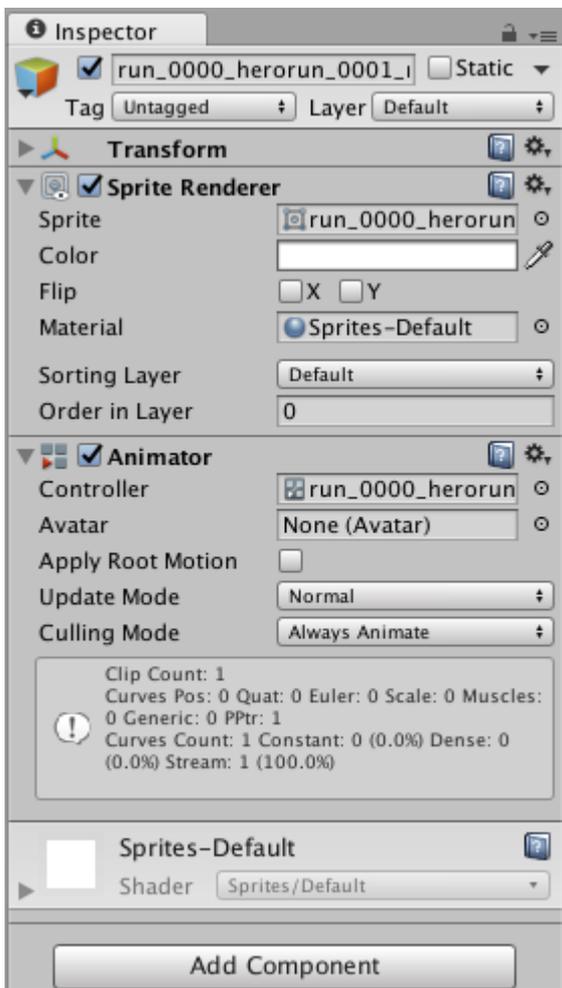


Trascina ogni singola immagine di una singola animazione dalla cartella delle risorse alla vista scena. Unity mostrerà una finestra di dialogo per nominare la nuova clip di animazione.



Questa è una scorciatoia utile per:

- creando nuovi oggetti di gioco
- assegnare due componenti (un Sprite Renderer e un Animator)
- creare controller di animazione (e collegarli a loro il nuovo componente Animator)
- creare clip di animazione con i fotogrammi selezionati



Anteprima della riproduzione nella scheda di animazione facendo clic su Riproduci:



Lo stesso metodo può essere utilizzato per creare nuove animazioni per lo stesso oggetto di gioco, quindi eliminare il nuovo oggetto di gioco e il controller di animazione. Aggiungi la nuova clip di animazione al controller di animazione di quell'oggetto nello stesso modo con l'animazione 3D.

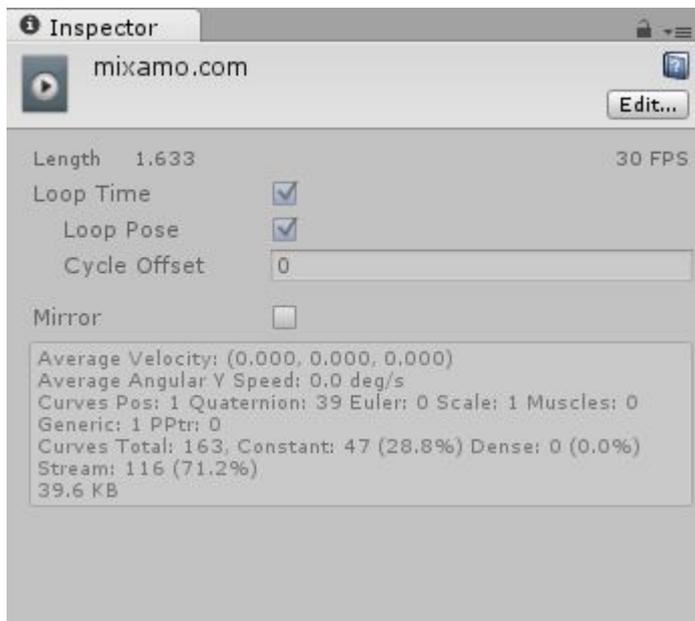
Curve di animazione

Le curve di animazione ti consentono di modificare un parametro float durante la riproduzione dell'animazione. Ad esempio, se è presente un'animazione di lunghezza 60 secondi e si desidera un valore / parametro float, chiamarlo X, per variare l'animazione (come ad animazione = 0.0s; X = 0.0, ad animazione = 30.0s; X = 1.0, a tempo di animazione = 60.0s; X = 0.0).

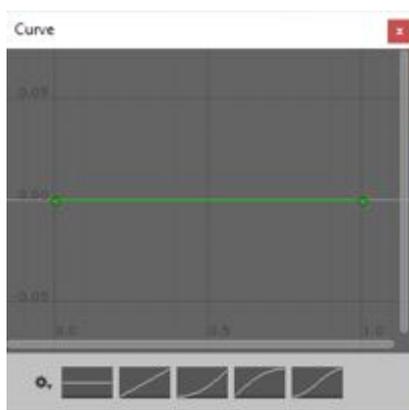
Una volta ottenuto il valore float, puoi usarlo per tradurre, ruotare, ridimensionare o usarlo in qualsiasi altro modo.

Per il mio esempio, mostrerò un oggetto di gioco giocatore in esecuzione. Quando viene riprodotta l'animazione per la corsa, la velocità di traduzione del giocatore dovrebbe aumentare con l'avanzare dell'animazione. Quando l'animazione raggiunge la sua fine, la velocità di traduzione dovrebbe diminuire.

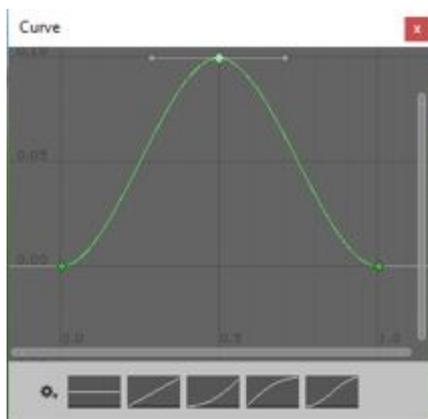
Ho creato una clip di animazione in esecuzione. Selezionare la clip e quindi nella finestra di ispezione, fare clic su Modifica.



Una volta lì, scorri verso il basso fino a Curves. Clicca sul segno + per aggiungere una curva. Assegna un nome alla curva, ad esempio ForwardRunCurve. Clicca sulla curva in miniatura sulla destra. Si aprirà una piccola finestra con una curva di default in esso.



Vogliamo una curva a forma di parabola dove si alza e poi cade. Per impostazione predefinita, ci sono 2 punti sulla linea. Puoi aggiungere più punti facendo doppio clic sulla curva. Trascina i punti per creare una forma simile alla seguente.



Nella finestra degli animatori, aggiungi la clip in esecuzione. Inoltre, aggiungi un parametro float con lo stesso nome della curva, ad esempio ForwardRunCurve.

Quando viene riprodotta l'animazione, il valore float cambia in base alla curva. Il seguente codice mostrerà come utilizzare il valore float:

```
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 variabile `curveValue` mantiene il valore della curva (`ForwardRunCurve`) in qualsiasi momento. Stiamo usando quel valore per cambiare la velocità della traduzione. Puoi allegare questo script all'oggetto del gioco del giocatore.

Leggi Unity Animation online: <https://riptutorial.com/it/unity3d/topic/5448/unity-animation>

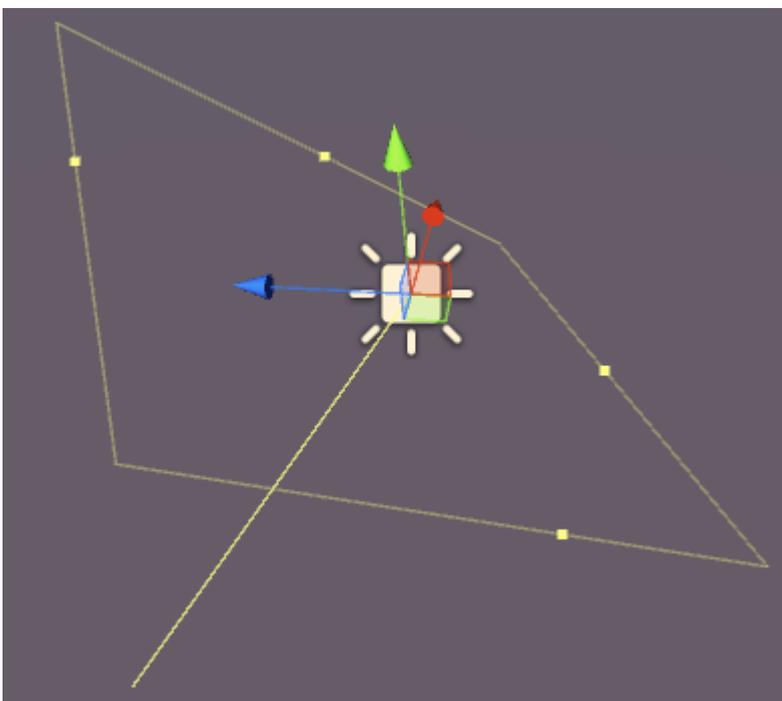
Capitolo 37: Unity Lighting

Examples

Tipi di luce

Area Light

La luce viene emessa attraverso la superficie di un'area rettangolare. Sono solo cotti, il che significa che non sarete in grado di vedere l'effetto fino a quando non si cuoce la scena.

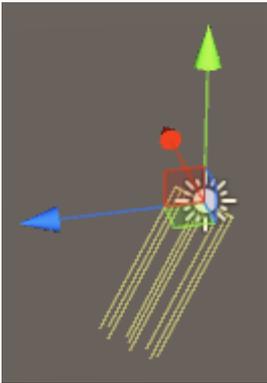


Le luci di area hanno le seguenti proprietà:

- **Larghezza** - Larghezza dell'area chiara.
- **Altezza** - Altezza dell'area chiara.
- **Colore** : assegna il colore della luce.
- **Intensità** - Quanto è potente la luce da 0 a 8.
- **Intensità di rimbalzo** - Quanto è potente la luce *indiretta* da 0 a 8.
- **Disegna Halo** - **Disegna** un alone attorno alla luce.
- **Flare** : consente di assegnare un effetto flare alla luce.
- **Modalità di rendering** : automatica, importante, non importante.
- **Maschera di abbattimento** : consente di illuminare selettivamente parti di una scena.

Luce direzionale

Le luci direzionali emettono luce in un'unica direzione (proprio come il sole). Non importa in quale punto della scena è posizionato il GameObject effettivo in quanto la luce è "ovunque". L'intensità della luce non diminuisce come gli altri tipi di luce.

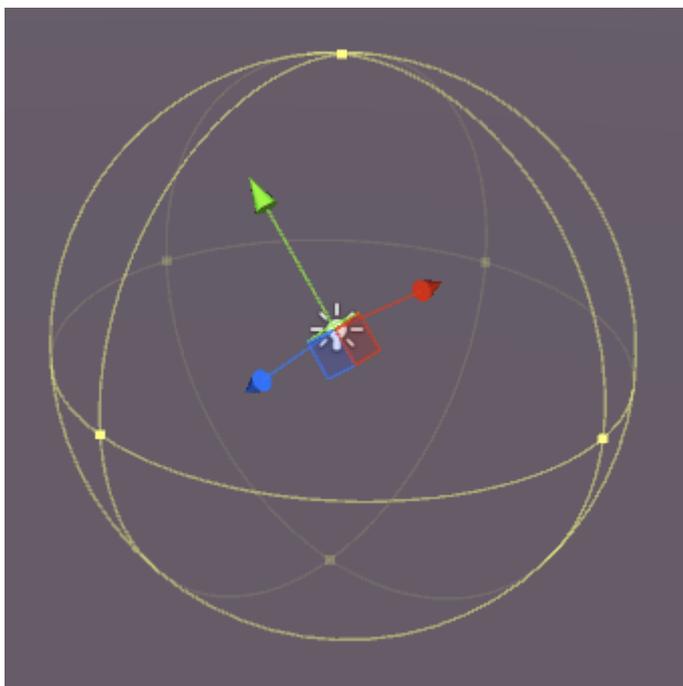


Una luce direzionale ha le seguenti proprietà:

- **Cottura** - Tempo reale, cotto o misto.
- **Colore** : assegna il colore della luce.
- **Intensità** - Quanto è potente la luce da 0 a 8.
- **Intensità di rimbalzo** - Quanto è potente la luce *indiretta* da 0 a 8.
- **Tipo di ombreggiatura** : nessuna ombra, ombre dure o ombre morbide.
- **Cookie** : consente di assegnare un cookie per la luce.
- **Dimensione del cookie** : la dimensione del cookie assegnato.
- **Disegna Halo** - **Disegna** un alone attorno alla luce.
- **Flare** : consente di assegnare un effetto flare alla luce.
- **Modalità di rendering** : automatica, importante, non importante.
- **Maschera di abbattimento** : consente di illuminare selettivamente parti di una scena.

Punto luce

Una luce puntiforme emette luce da un punto nello spazio in tutte le direzioni. Più lontano dal punto di origine, meno intensa è la luce.

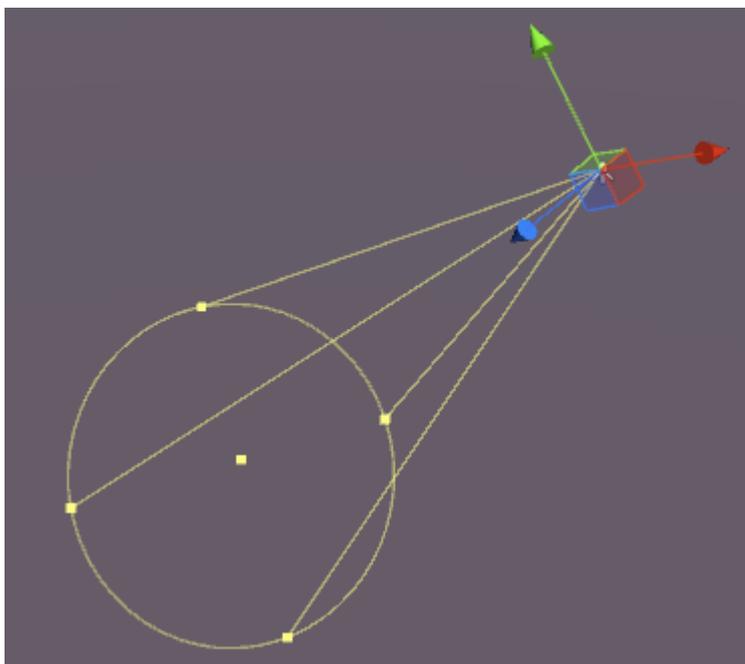


I punti luce hanno le seguenti proprietà:

- **Cottura** - Tempo reale, cotto o misto.
- **Intervallo** : la distanza dal punto in cui la luce non raggiunge più.
- **Colore** : assegna il colore della luce.
- **Intensità** - Quanto è potente la luce da 0 a 8.
- **Intensità di rimbalzo** - Quanto è potente la luce *indiretta* da 0 a 8.
- **Tipo di ombreggiatura** : nessuna ombra, ombre dure o ombre morbide.
- **Cookie** : consente di assegnare un cookie per la luce.
- **Disegna Halo - Disegna** un alone attorno alla luce.
- **Flare** : consente di assegnare un effetto flare alla luce.
- **Modalità di rendering** : automatica, importante, non importante.
- **Maschera di abbattimento** : consente di illuminare selettivamente parti di una scena.

Riflettore

Una luce spot è molto simile a una luce puntiforme, ma l'emissione è limitata ad un angolo. Il risultato è un "cono" di luce, utile per i fari delle automobili o i proiettori.



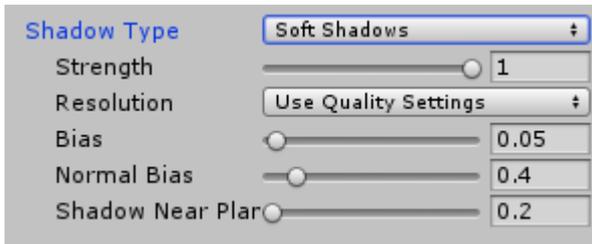
Le luci spot hanno le seguenti proprietà:

- **Cottura** - Tempo reale, cotto o misto.
- **Intervallo** : la distanza dal punto in cui la luce non raggiunge più.
- **Angolo spot** : l'angolo di emissione della luce.
- **Colore** : assegna il colore della luce.
- **Intensità** - Quanto è potente la luce da 0 a 8.
- **Intensità di rimbalzo** - Quanto è potente la luce *indiretta* da 0 a 8.
- **Tipo di ombreggiatura** : nessuna ombra, ombre dure o ombre morbide.
- **Cookie** : consente di assegnare un cookie per la luce.
- **Disegna Halo - Disegna** un alone attorno alla luce.
- **Flare** : consente di assegnare un effetto flare alla luce.
- **Modalità di rendering** : automatica, importante, non importante.
- **Maschera di abbattimento** : consente di illuminare selettivamente parti di una scena.

Nota su Ombre

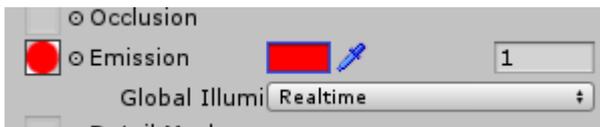
Se selezioni Ombra dura o sfumata, le seguenti opzioni diventano disponibili nella finestra di ispezione:

- **Forza** - Quanto sono scure le ombre da 0 a 1.
- **Risoluzione** : quanto sono dettagliate le ombre.
- **Bias** : grado in cui le superfici di colata dell'ombra vengono allontanate dalla luce.
- **Normal Bias** - Il grado in cui le superfici di lancio dell'ombra vengono spinte verso l'interno lungo le loro normali.
- **Shadow Near Plane** - 0.1 - 10.



emissione

L'emissione è quando una superficie (o meglio un materiale) emette luce. Nel pannello dell'ispettore per un materiale su un oggetto statico che utilizza lo Shader standard, esiste una proprietà di emissione:

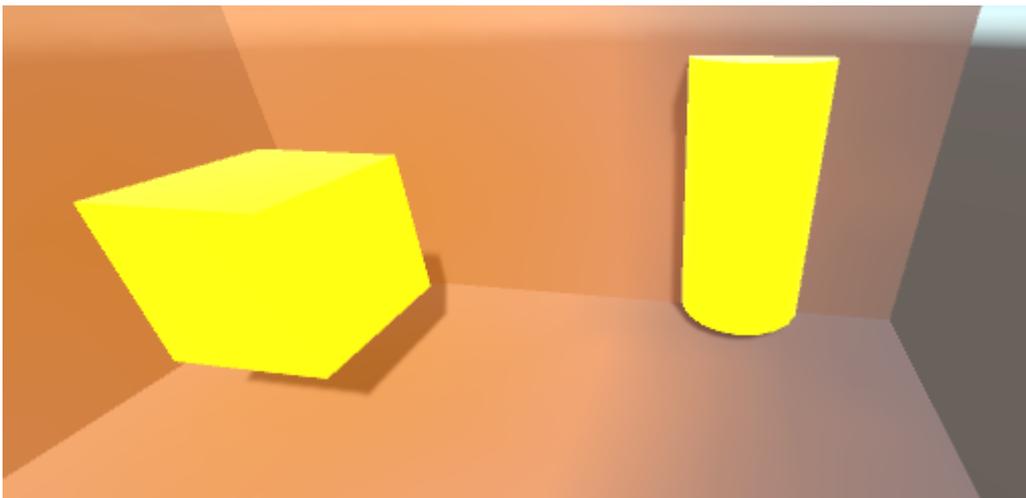


Se si modifica questa proprietà su un valore superiore a quello predefinito di 0, è possibile impostare il colore di emissione o assegnare una **mappa di emissione** al materiale. Qualsiasi texture assegnata a questo slot consentirà all'emissione di utilizzare i propri colori.

C'è anche un'opzione di Illuminazione Globale che ti permette di stabilire se l'emissione è cotta su oggetti statici vicini o no:

- **Cotto** - L'emissione sarà cotta nella scena
- **Realtime** - L'emissione interesserà oggetti dinamici
- **Nessuna** - L'emissione non influirà sugli oggetti vicini

Se l'oggetto *non* è impostato su statico, l'effetto farà apparire "luminoso" l'oggetto ma non viene emessa alcuna luce. Il cubo qui è statico, il cilindro no:



Puoi impostare il colore dell'emissione in codice come questo:

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

La luce emessa cadrà a una velocità quadratica e si mostrerà solo contro i materiali statici nella scena.

Leggi Unity Lighting online: <https://riptutorial.com/it/unity3d/topic/7884/unity-lighting>

Capitolo 38: Unity Profiler

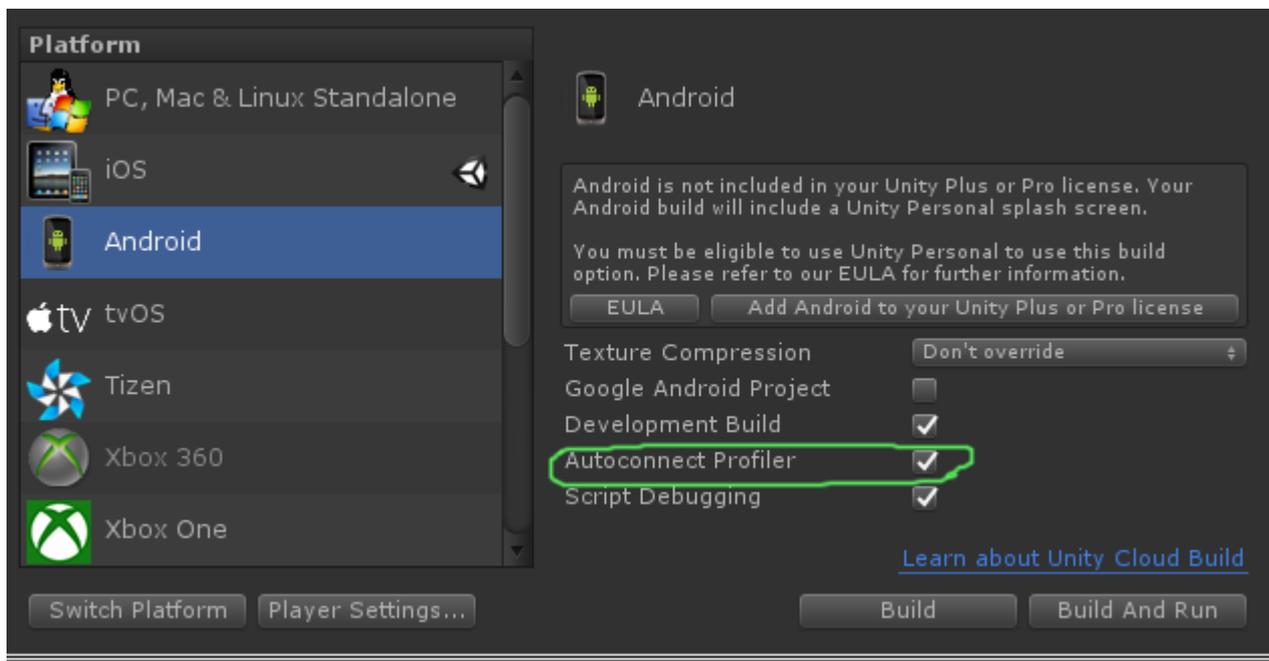
Osservazioni

Utilizzo di Profiler su un altro dispositivo

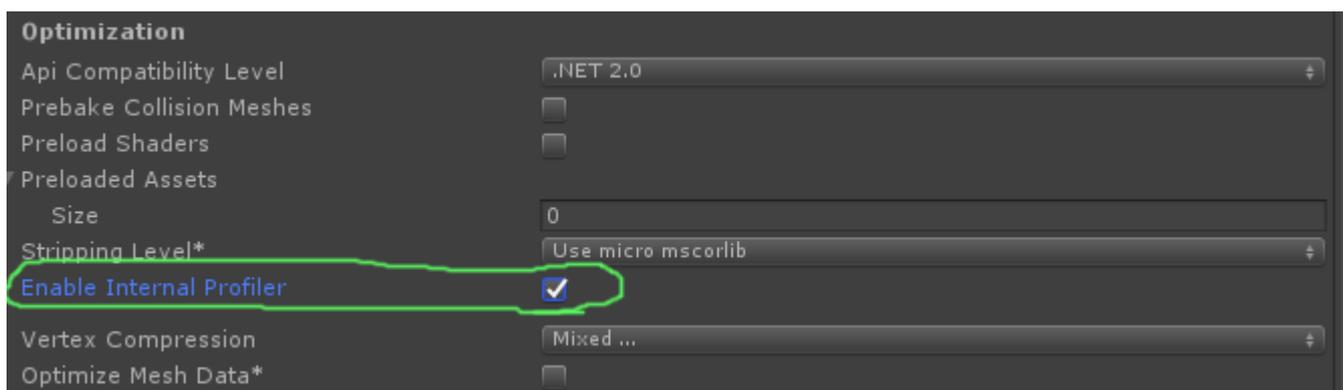
Ci sono alcune cose importanti da sapere per agganciare correttamente il Profiler su piattaforme diverse.

android

Per poter allegare correttamente il profilo, è necessario utilizzare il pulsante "Crea ed esegui" dalla finestra Impostazioni di **compilazione** con l'opzione **Verifica connessione automatica** selezionata.



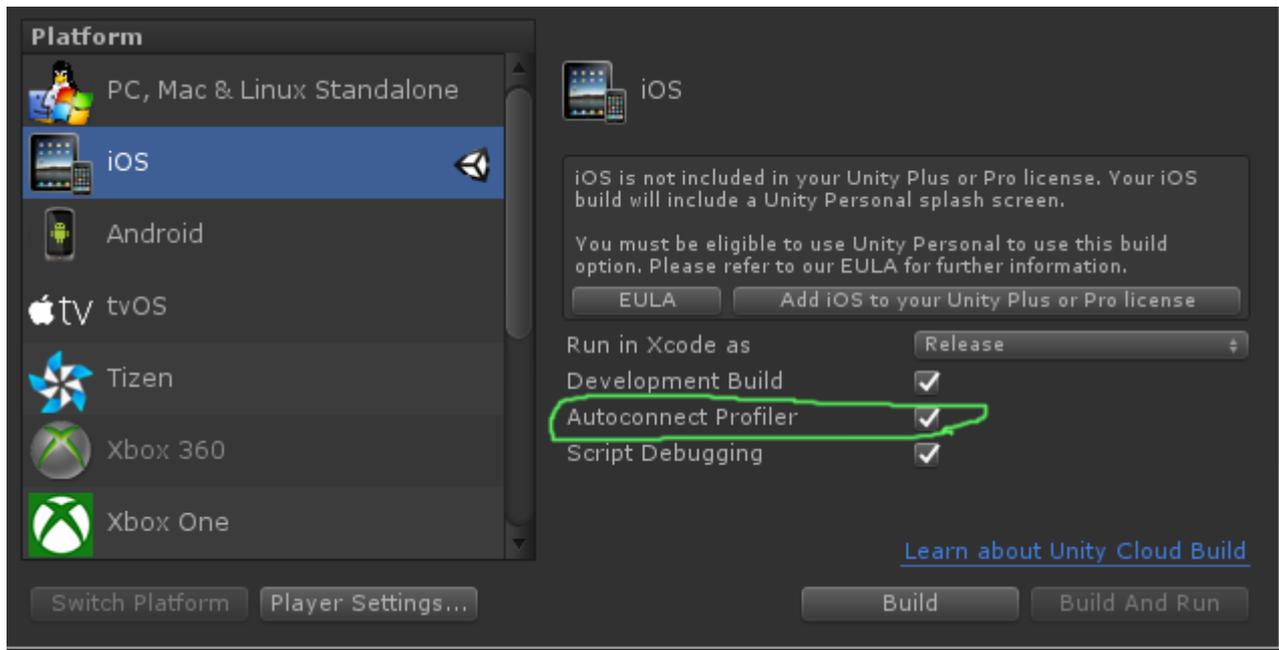
Un'altra opzione obbligatoria, nel [pannello Impostazioni di Android Player](#) nella scheda Altre impostazioni, è la casella di controllo **Abilita il profiler interno** che deve essere controllato in modo che LogCat possa generare informazioni sul profiler.



L'utilizzo solo di "Build" non consente al profiler di connettersi a un dispositivo Android perché "Build and Run" utilizza specifici argomenti della riga di comando per avviarlo con LogCat.

iOS

Per allegare correttamente il profilo, il pulsante "Costruisci ed esegui" dalla finestra delle impostazioni di **compilazione** con l'opzione **Controllo connessione** abilitato deve essere utilizzato alla prima esecuzione.



Su iOS, non ci sono opzioni nelle impostazioni del giocatore che devono essere impostate affinché il Profiler sia abilitato. Dovrebbe funzionare fuori dalla scatola.

Examples

Profiler Markup

Utilizzo della classe Profiler

Una buona pratica è usare Profiler.BeginSample e Profiler.EndSample perché avrà la sua propria voce nella finestra di Profiler.

Inoltre, tali tag verranno eliminati su build non di sviluppo utilizzando l'uso di ConditionalAttribute, quindi non è necessario rimuoverli dal codice.

```
public class SomeClass : MonoBehaviour
{
    void SomeFunction()
    {
        Profiler.BeginSample("SomeClass.SomeFunction");
        // Various call made here
        Profiler.EndSample();
    }
}
```

```
}  
}
```

Ciò creerà una voce "SomeClass.SomeFunction" nella finestra di Profiler che consentirà un debugging e un'identificazione più semplici del collo della bottiglia.

Leggi Unity Profiler online: <https://riptutorial.com/it/unity3d/topic/6974/unity-profiler>

Capitolo 39: Utilizzo del controllo del codice sorgente Git con Unity

Examples

Utilizzo di Git Large File Storage (LFS) con Unity

Prefazione

Git può funzionare con lo sviluppo di videogiochi fuori dagli schemi. Tuttavia, l'avvertenza principale è che i file multimediali con versioni di grandi dimensioni (> 5 MB) possono rappresentare un problema a lungo termine dato che la cronologia dei commit è a dismisura - Git semplicemente non è stato originariamente creato per il versioning di file binari.

La grande notizia è che da metà 2015 GitHub ha rilasciato un plugin per Git chiamato [Git LFS](#) che tratta direttamente questo problema. Ora puoi creare facilmente ed efficientemente file binari di grandi dimensioni!

Infine, questa documentazione si concentra sui requisiti specifici e le informazioni necessarie per garantire che la tua vita Git funzioni bene con lo sviluppo di videogiochi. Questa guida non tratterà come usare Git stesso.

Installazione di Git & Git-LFS

Hai una serie di opzioni a tua disposizione come sviluppatore e la prima scelta è se installare la riga di comando Git di base o lasciare che una delle popolari applicazioni Git della GUI si occupi di esso.

Opzione 1: utilizzare un'applicazione Git GUI

Questa è davvero una preferenza personale in quanto vi sono alcune opzioni in termini di Git GUI o se utilizzare una GUI del tutto. Hai un numero di applicazioni tra cui scegliere, qui ci sono 3 delle più popolari:

- [Sourcetree \(gratuito\)](#)
- [Github Desktop \(gratuito\)](#)
- [SmartGit \(Commerical\)](#)

Una volta installata l'applicazione preferita, fai clic su Google e segui le istruzioni su come assicurarsi che sia configurato per Git-LFS. Salteremo questo passaggio in questa guida poiché è specifica dell'applicazione.

Opzione 2: installa Git & Git-LFS

Questo è piuttosto semplice - [Installa Git](#) . Poi. [Installa Git LFS](#) .

Configurazione di Git Large File Storage sul tuo progetto

Se stai utilizzando il plug-in Git LFS per fornire un supporto migliore per i file binari, dovrai impostare alcuni tipi di file che Git LFS gestirà. Aggiungi il sotto al tuo file `.gitattributes` nella root del tuo repository per supportare i file binari comuni usati nei progetti 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
```

Impostazione di un repository Git per Unity

Quando si inizializza un repository Git per lo sviluppo di Unity, ci sono un paio di cose che devono essere fatte.

Unity Ignora le cartelle

Non tutto dovrebbe essere versionato nel repository. Puoi aggiungere il modello qui sotto al tuo file `.gitignore` nella root del tuo repository. In alternativa, puoi controllare l' [unità](#) open source [.gitignore su GitHub](#) e in alternativa [crearne](#) una utilizzando [gitignore.io per l'unità](#).

```
# 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
```

Per ulteriori informazioni su come impostare un file `.gitignore`, [consulta qui](#) .

Impostazioni del progetto Unity

Per impostazione predefinita, i progetti Unity non sono configurati per supportare il controllo delle versioni correttamente.

1. (Salta questo passaggio nella versione 4.5 e successive) Abilita l'opzione `External in Unity` → `Preferences` → `Packages` → `Repository` .
2. Passa a `Visible Meta Files in Edit` → `Project Settings` → `Editor` → `Version Control Mode` .
3. Passa a `Force Text in Edit` → `Project Settings` → `Editor` → `Asset Serialization Mode` .
4. Salva la scena e proietta dal menu `File` .

Configurazione aggiuntiva

Uno dei pochi maggiori fastidi che si hanno con l'utilizzo dei progetti Git with Unity è che a Git non interessano le directory e lasceranno felicemente le directory vuote dopo aver rimosso i file da esse. Unity `*.meta` file `*.meta` per queste directory e può causare un po 'di battaglia tra i membri del team quando Git si impegna a continuare ad aggiungere e rimuovere questi meta file.

[Aggiungi questo hook post-merge Git](#) alla cartella `/.git/hooks/` per repository con progetti Unity al loro interno. Dopo ogni Git pull / merge, verrà esaminato quali file sono stati rimossi, verificare se

la directory in cui è presente è vuota e, in tal caso, eliminarla.

Fusione di scene e prefabbricati

Un problema comune quando si lavora con Unity è quando 2 o più sviluppatori stanno modificando una scena Unity o prefabbricati (file * .unity). Git non sa come unirli correttamente fuori dalla scatola. Fortunatamente il team Unity ha implementato uno strumento chiamato **SmartMerge** che rende automatica l'unione semplice. La prima cosa da fare è aggiungere le seguenti righe al tuo file `.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"
```

Su **Windows** il percorso di UnityYAMLMerge è:

```
C:\Program Files\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

o

```
C:\Program Files (x86)\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

e su **MacOSX** :

```
/Applications/Unity/Unity.app/Contents/Tools/UnityYAMLMerge
```

Una volta fatto, il mergetool sarà disponibile quando sorgono conflitti durante l'unione / rebase. Non dimenticare di eseguire `git mergetool` manualmente per attivare UnityYAMLMerge.

Leggi Utilizzo del controllo del codice sorgente Git con Unity online:

<https://riptutorial.com/it/unity3d/topic/2195/utilizzo-del-controllo-del-codice-sorgente-git-con-unity>

Capitolo 40: Vector3

introduzione

La struttura `Vector3` rappresenta una coordinata 3D ed è una delle strutture di backbone della libreria `UnityEngine`. La struttura `Vector3` si trova più comunemente nel componente `Transform` della maggior parte degli oggetti di gioco, dove è usata per mantenere la *posizione* e la *scala*. `Vector3` offre buone funzionalità per l'esecuzione di operazioni vettoriali comuni. [Puoi leggere di più sulla struttura di `Vector3` nell'API di Unity.](#)

Sintassi

- `public Vector3 ();`
- pubblico `Vector3 (float x, float y);`
- pubblico `Vector3 (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, distanza di galleggiamento);`

Examples

Valori statici

La struttura `Vector3` contiene alcune variabili statiche che forniscono valori `Vector3` comunemente usati. La maggior parte rappresenta una *direzione*, ma possono comunque essere utilizzati in modo creativo per fornire funzionalità aggiuntive.

`Vector3.zero` **e** `Vector3.one`

`Vector3.zero` e `Vector3.one` sono tipicamente usati in connessione con un `Vector3` *normalizzato*; cioè, un `Vector3` cui i valori `x`, `y` e `z` hanno una magnitudine di 1. Come tale, `Vector3.zero` rappresenta il valore più basso, mentre `Vector3.one` rappresenta il valore più grande.

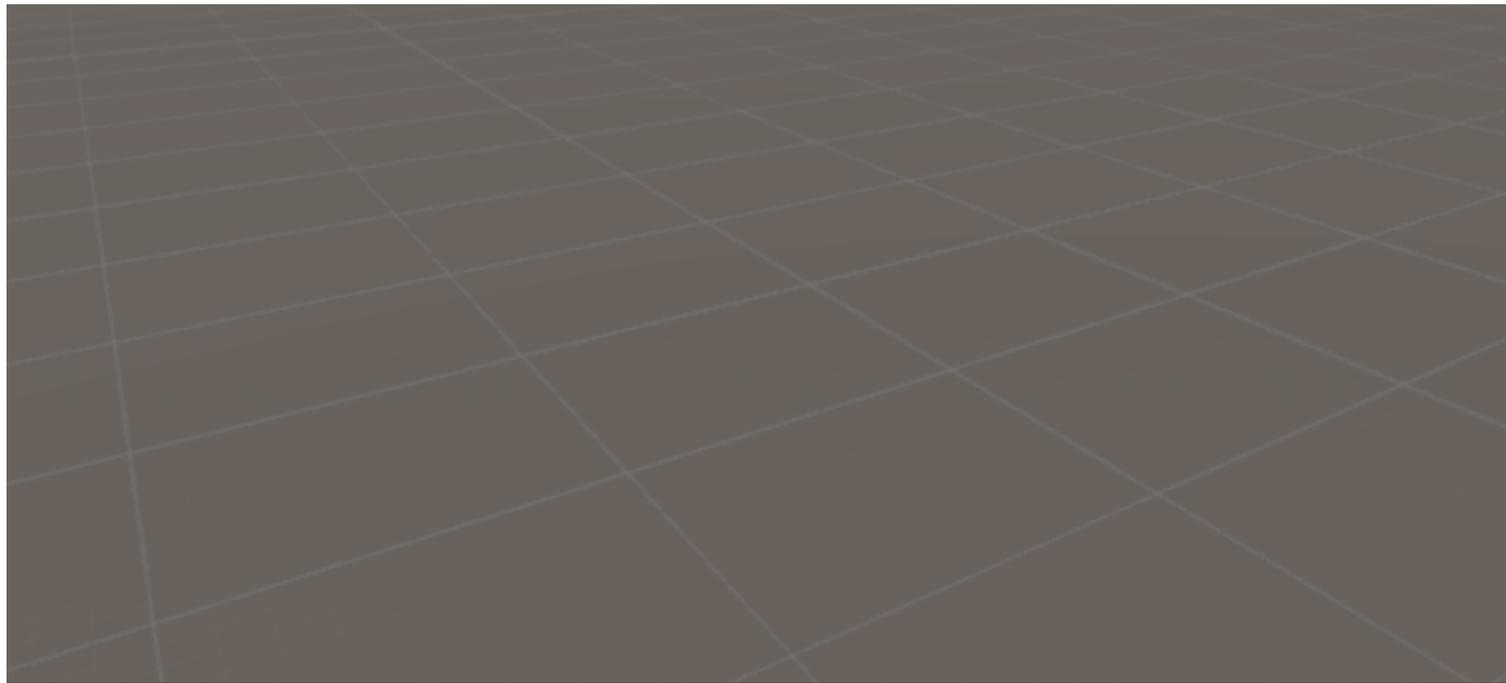
`Vector3.zero` è anche comunemente usato per impostare la posizione di default sulle trasformazioni di oggetti.

La seguente classe usa `Vector3.zero` e `Vector3.one` per gonfiare e sgonfiare una sfera.

```
using UnityEngine;
```

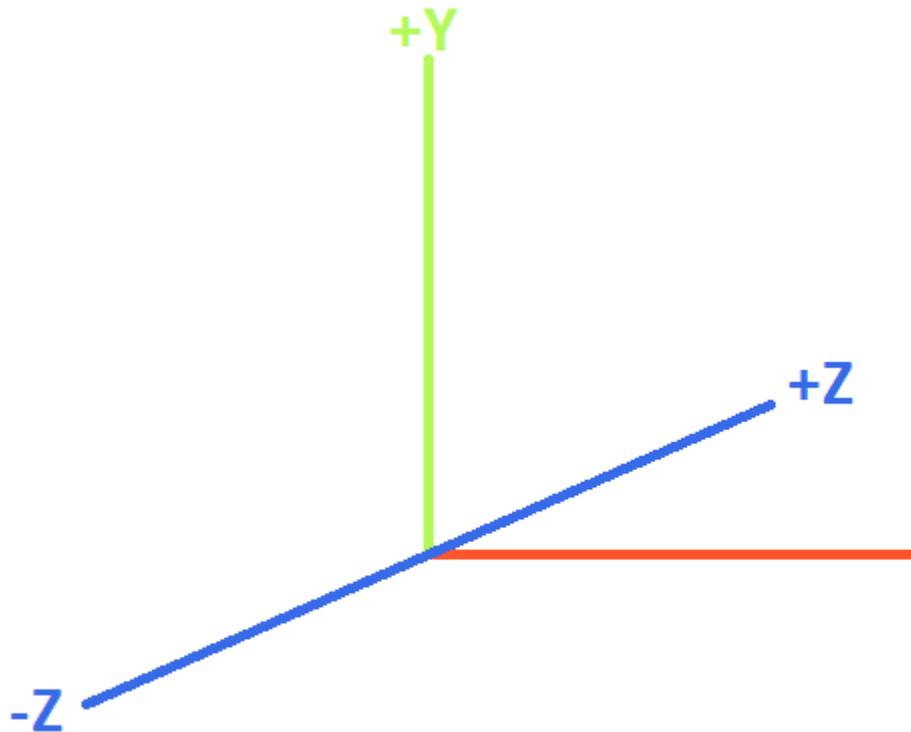
```
public class Inflater : MonoBehaviour
{
    <summary>A sphere set up to inflate and deflate between two values.</summary>
    public ScaleBetween sphere;

    ///<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);
    }
}
```



Indicazioni statiche

Le direzioni statiche possono essere utili in numerose applicazioni, con direzione lungo il positivo e il negativo di tutti e tre gli assi. È importante notare che Unity utilizza un sistema di coordinate mancino, che influisce sulla direzione.



LEFT-HANDED COORDINATE SYSTEM

La seguente classe usa le direzioni di `Vector3` statiche per spostare gli oggetti lungo i tre assi.

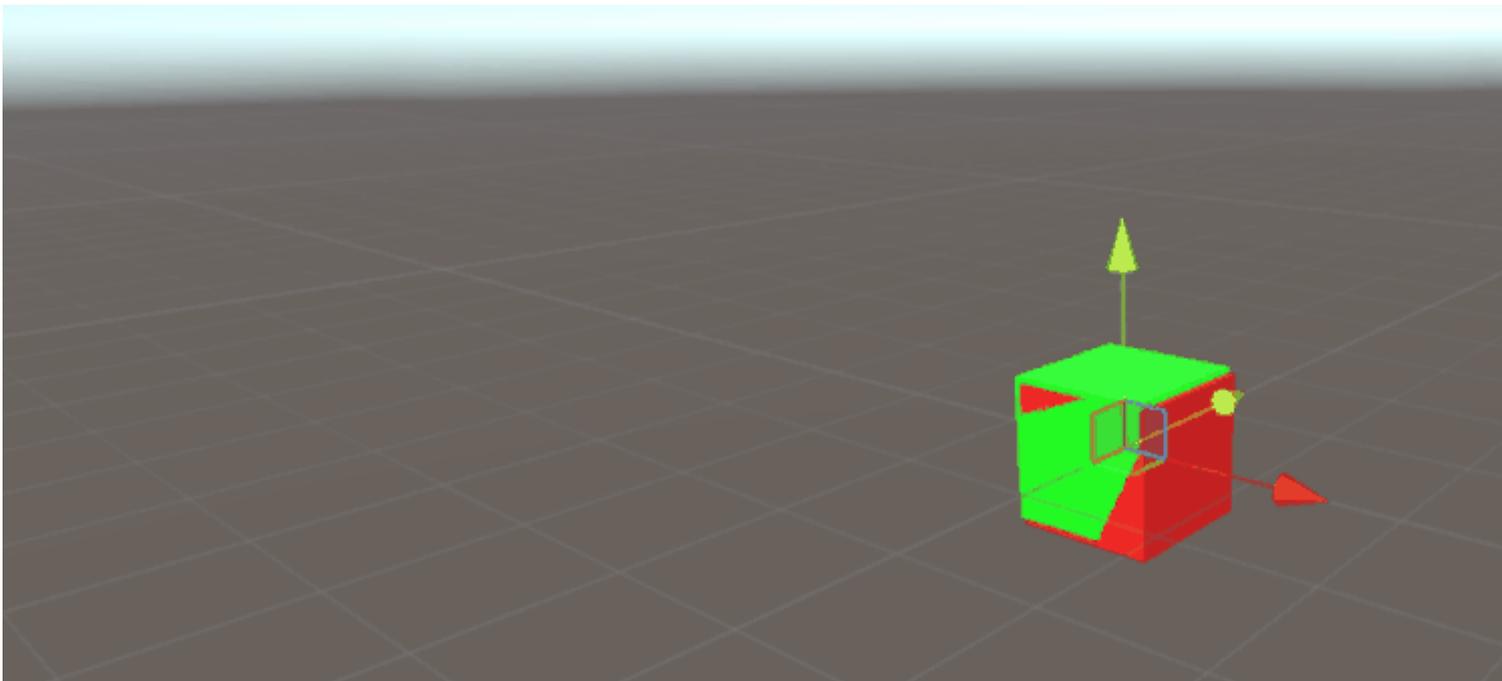
```
using UnityEngine;

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);
    }
}
```



Indice

Valore	X	y	z	<code>new Vector3()</code> metodo	<code>new Vector3()</code> equivalente
<code>Vector3.zero</code>	0	0	0	<code>new Vector3(0, 0, 0)</code>	
<code>Vector3.one</code>	1	1	1	<code>new Vector3(1, 1, 1)</code>	
<code>Vector3.left</code>	-1	0	0	<code>new Vector3(-1, 0, 0)</code>	
<code>Vector3.right</code>	1	0	0	<code>new Vector3(1, 0, 0)</code>	
<code>Vector3.down</code>	0	-1	0	<code>new Vector3(0, -1, 0)</code>	
<code>Vector3.up</code>	0	1	0	<code>new Vector3(0, 1, 0)</code>	
<code>Vector3.back</code>	0	0	-1	<code>new Vector3(0, 0, -1)</code>	
<code>Vector3.forward</code>	0	0	1	<code>new Vector3(0, 0, 1)</code>	

Creare un Vector3

Una struttura `Vector3` può essere creata in diversi modi. `Vector3` è una struttura e, come tale, in genere dovrà essere istanziata prima dell'uso.

Costruttori

Ci sono tre costruttori incorporati per istanziare un `Vector3` .

Costruttore	Risultato
<code>new Vector3()</code>	Crea una struttura <code>Vector3</code> con coordinate di (0, 0, 0).
<code>new Vector3(float x, float y)</code>	Crea una <code>Vector3</code> struttura con il dato <code>x</code> ed <code>y</code> coordinate. <code>z</code> sarà impostato a 0.
<code>new Vector3(float x, float y, float z)</code>	Crea una struttura <code>Vector3</code> con le coordinate <code>x</code> , <code>y</code> <code>z</code> indicate.

Conversione da un `Vector2` o `Vector4`

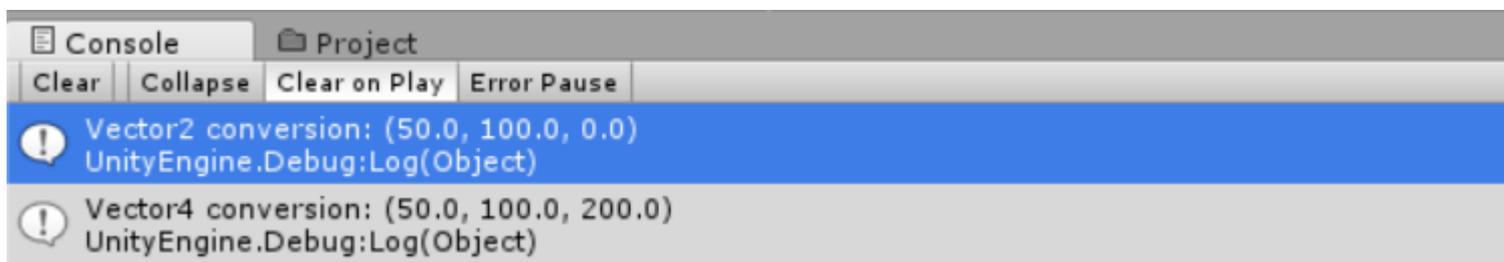
Sebbene rari, è possibile imbattersi in situazioni in cui è necessario trattare le coordinate di una struttura `Vector2` o `Vector4` come `Vector3` . In questi casi, puoi semplicemente passare `Vector2` o `Vector4` direttamente a `Vector3` , senza prima averlo istanziato. Come dovrebbe essere assunto, una struttura `Vector2` passerà solo i valori `x` e `y` , mentre una classe `Vector4` sua `w` .

Possiamo vedere la conversione diretta nello script seguente.

```
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);
}
```



Applicazione del movimento

La struttura `Vector3` contiene alcune funzioni statiche che possono fornire utilità quando si desidera applicare il movimento a `Vector3` .

Le funzioni di lerp forniscono movimento tra due coordinate basate su una frazione fornita. Dove `Lerp` consentirà solo il movimento tra le due coordinate, `LerpUnclamped` consente frazioni che si muovono al di fuori dei confini tra le due coordinate.

Forniamo la frazione di movimento come un `float`. Con un valore di `0.5`, troviamo il punto medio tra le due coordinate `Vector3`. Un valore pari a `0` o `1` restituirà il primo o il secondo `Vector3`, rispettosamente, in quanto questi valori sono correlati a nessun movimento (restituendo così il primo `Vector3`) o il movimento completato (questo restituisce il secondo `Vector3`). È importante notare che nessuna delle due funzioni consentirà cambiamenti nella frazione di movimento. Questo è qualcosa di cui dobbiamo tener conto manualmente.

Con `Lerp`, tutti i valori sono bloccati tra `0` e `1`. Questo è utile quando vogliamo fornire il movimento verso una direzione e non vogliamo superare la destinazione. `LerpUnclamped` può assumere qualsiasi valore e può essere utilizzato per fornire movimento *lontano* dalla destinazione o *oltre* la destinazione.

Il seguente script utilizza `Lerp` e `LerpUnclamped` per spostare un oggetto a un ritmo costante.

```
using UnityEngine;

public class Lerpung : 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);

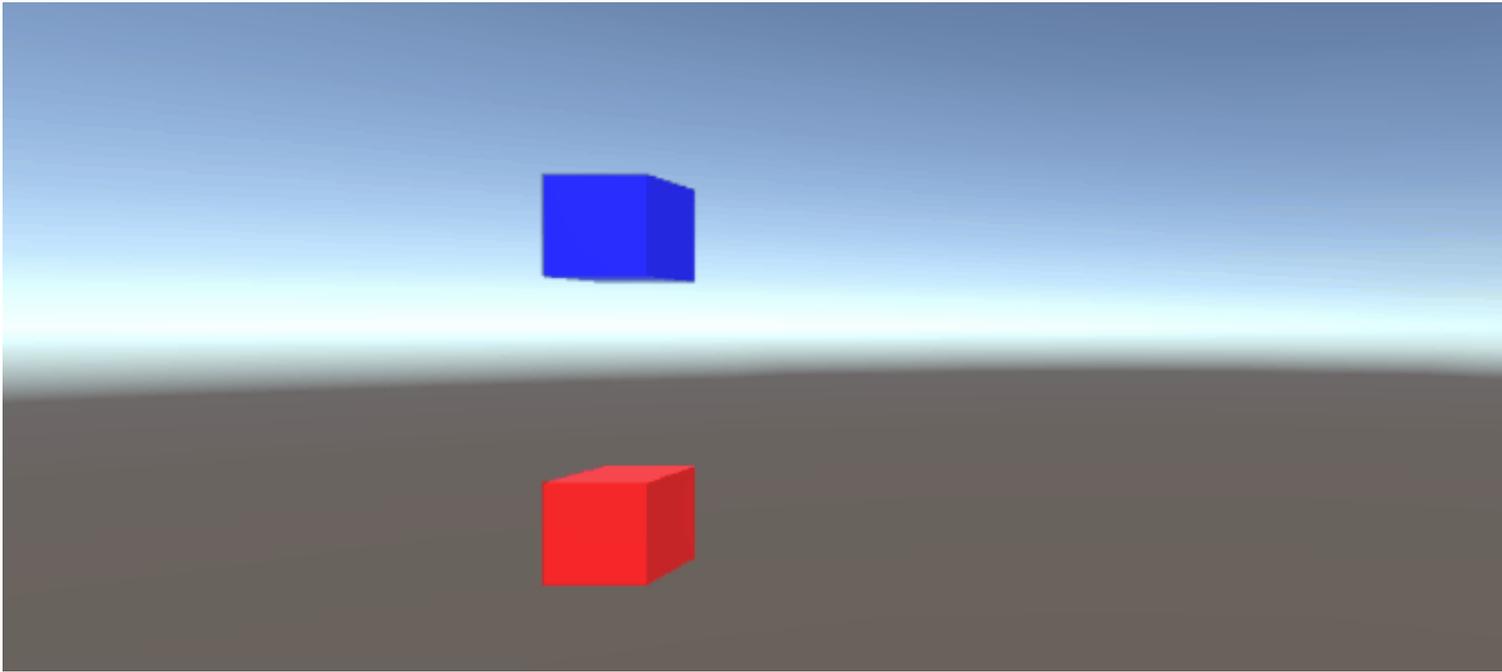
    /// <summary>The blue box will use LerpUnclamped to move. We will
    /// link this object in via the inspector.</summary>
    public GameObject lerpUnclampedObject;
    /// <summary>The starting position for our blue box.</summary>
    public Vector3 lerpUnclampedStart = new Vector3(0, 3, 0);
    /// <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.
        // deltaTime * 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 in modo *molto simile* a `Lerp` ; la differenza principale è che forniamo una *distanza* effettiva da spostare, invece di una *frazione* tra due punti. È importante notare che `MoveTowards` non si estenderà oltre il target `Vector3` .

Molto simile a `LerpUnclamped` , possiamo fornire un valore di distanza *negativo* per *allontanarci* dal `Vector3` destinazione. In questi casi, non `Vector3` mai il target `Vector3` , quindi il movimento è indefinito. In questi casi, possiamo considerare il target `Vector3` come una "direzione opposta"; fino a quando `Vector3` punta nella stessa direzione, rispetto all'inizio `Vector3` , il movimento negativo dovrebbe comportarsi normalmente.

Il seguente script usa `MoveTowards` per spostare un gruppo di oggetti verso un insieme di posizioni usando una distanza livellata.

```
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;

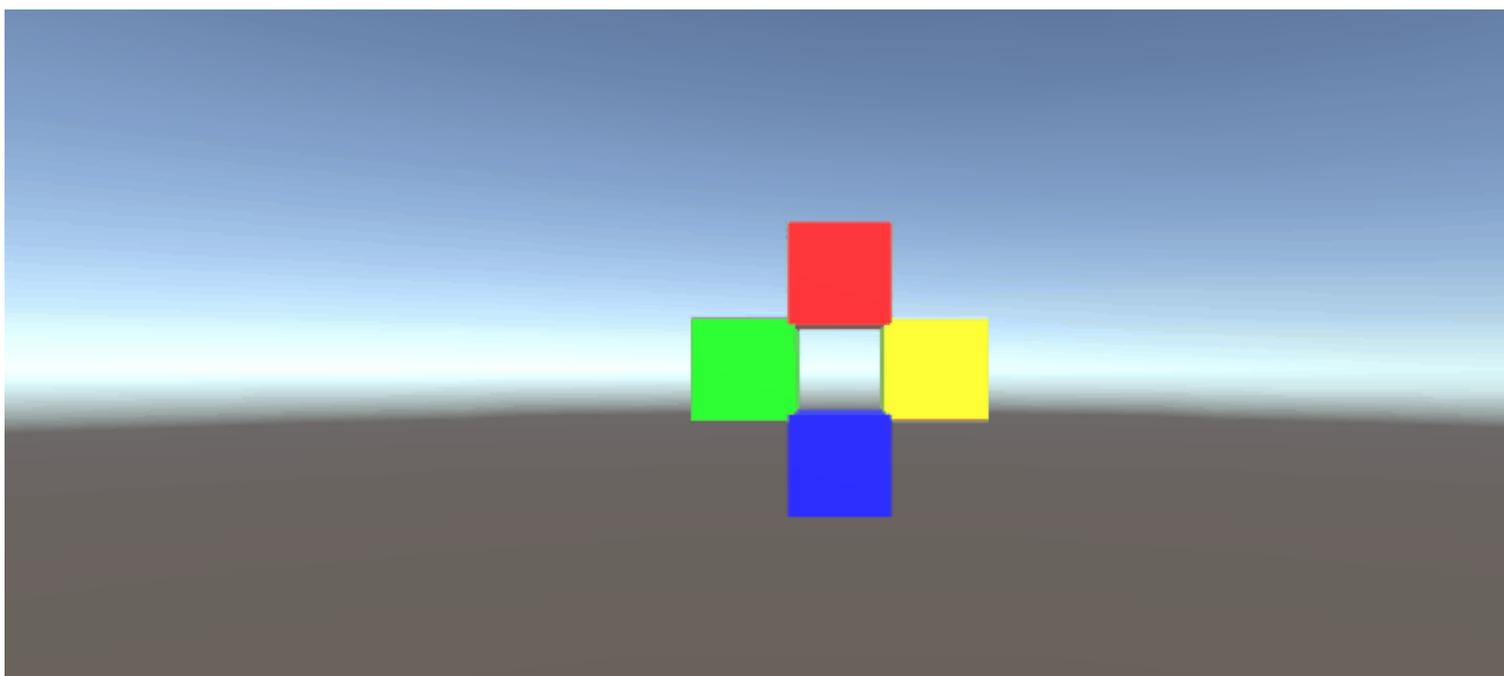
// The up cube will move upwards, until it reaches the
//position of (Vector3.up * 2), or (0, 2, 0).
upCube.transform.position
    = Vector3.MoveTowards(upCube.transform.position, (Vector3.up * 2f), distance);

// 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, indefinitely, 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

Pensa a `SmoothDamp` come a una variante di `MoveTowards` con smoothing incorporato. In base alla documentazione ufficiale, questa funzione è più comunemente utilizzata per eseguire il follow-up della fotocamera.

Insieme alle coordinate `Vector3` partenza e di destinazione, dobbiamo anche fornire un `Vector3` per rappresentare la velocità e un `float` rappresenta il tempo *approssimativo* che dovrebbe impiegare per completare il movimento. A differenza degli esempi precedenti, forniamo la velocità come *riferimento*, da incrementare, internamente. È importante prendere nota di ciò, poiché la modifica

della velocità al di fuori della funzione mentre stiamo ancora eseguendo la funzione può avere risultati indesiderati.

Oltre alle variabili *richieste*, possiamo anche fornire un `float` per rappresentare la velocità massima del nostro oggetto, e un `float` per rappresentare il gap temporale dal precedente richiamo di `SmoothDamp` all'oggetto. Non abbiamo *bisogno* di fornire questi valori; per impostazione predefinita, non ci sarà alcuna velocità massima, e il gap temporale sarà interpretato come `Time.deltaTime`. Ancora più importante, se si chiama la funzione uno per oggetto all'interno di una funzione `MonoBehaviour.Update()`, *non* è necessario dichiarare un intervallo di tempo.

```
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;

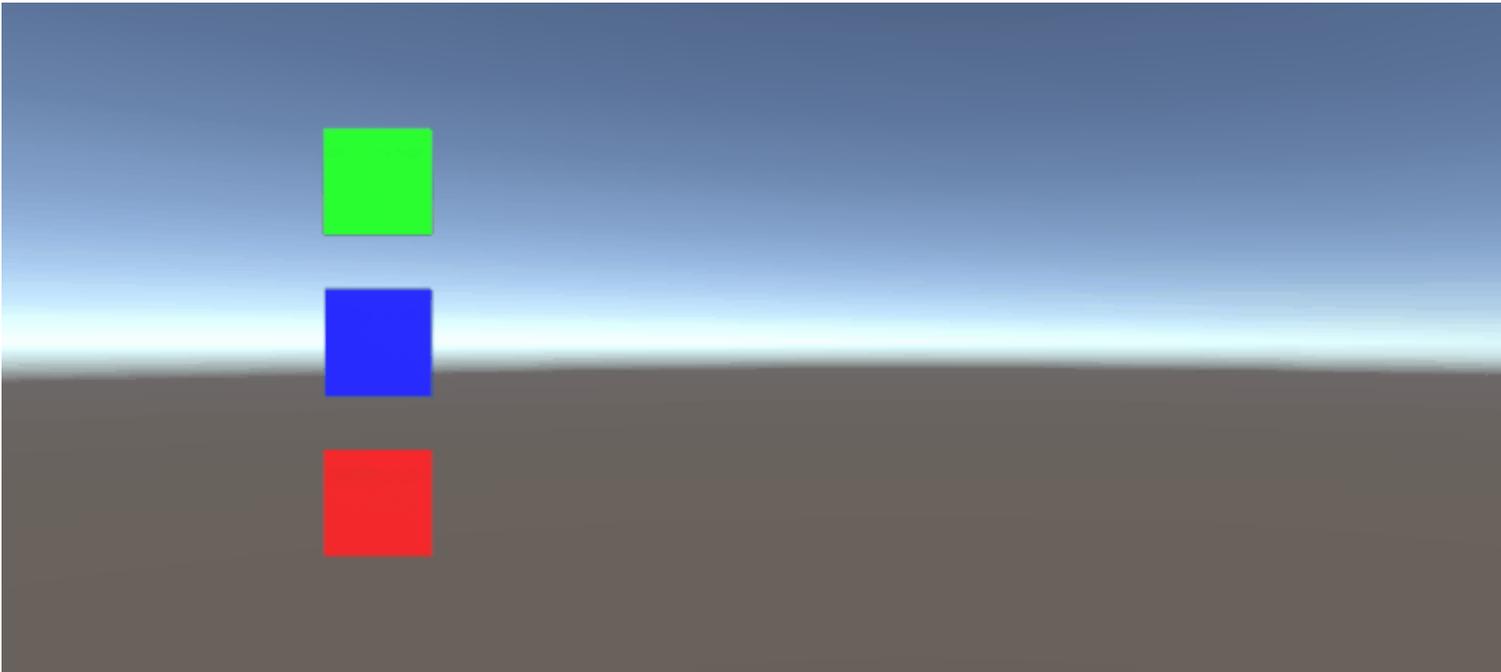
    void Update()
    {
        // The default SmoothDamp function will give us a general smooth movement.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime);

        // 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);
}
}
```



Leggi Vector3 online: <https://riptutorial.com/it/unity3d/topic/7827/vector3>

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con unity3d	Alexey Shimansky , Chris McFarland , Community , Desutoroiya , driconmax , F̃lámínġ óm̃bíé , James Radvan , josephsw , Linus Juhlin , Luís Fonseca , Maarten Bicknese , martinhodler , matiaslauriti , Mike B , Minzkraut , PlanetVaster , R.K123 , S. Tarik Çetin , Skyblade , SourabhV , SP. , tenpn , tim , user3071284
2	API CullingGroup	volvis
3	attributi	4444 , Thundernerd
4	Collisione	F̃lámínġ óm̃bíé , jjhavokk , Xander Luciano
5	Come utilizzare i pacchetti di asset	F̃lámínġ óm̃bíé
6	Comunicazione con il server	David Martinez , devon t , F̃lámínġ óm̃bíé , Maxim Kamalov , tim
7	coroutine	agiro , Fattie , Fehr , Giuseppe De Francesco , Problematic , Skyblade , Thulani Chivandikwa , Thundernerd , l̃olæz əɥl̃ qoq , volvis
8	Estendere l'editor	Pierrick Bignet , Skyblade , Thundernerd , l̃olæz əɥl̃ qoq , volvis
9	Fisica	eunoia , F̃lámínġ óm̃bíé , jack jay
10	Implementazione della classe MonoBehaviour	matiaslauriti , Skyblade , Thundernerd , user3797758
11	Importatori e (Post) Processori	gman , Skyblade , volvis
12	Integrazione annunci	l̃olæz əɥl̃ qoq
13	Livelli	Arijoon , dreadnought , Light Drake , RamenChef , Skyblade
14	Modelli di progettazione	Ian Newland
15	Negozi di beni	JakeD , Trent , zwcloud

16	Networking	David Martinez , driconmax , Rafiwui , RamenChef
17	Ottimizzazione	Ed Marty , EvilTak , Fjǫlmíng ómbíe , Grigory , JohnTube , Skyblade , Thulani Chivandikwa , volvis
18	Piattaforme mobili	Airwarfare , Skyblade
19	Plugin Android 101 - Un'introduzione	Venkat at Axiom Studios
20	Pooling di oggetti	Chris McFarland , Ed Marty , lase , matiaslauriti , S. Tarik Çetin , Thulani Chivandikwa , Thundernerd , Iolæz əɥɫ qoq , volvis
21	prefabbricati	Brandon Mintern , Dávid Florek , Fjǫlmíng ómbíe , gman , Gnemlock , Guglie , James Radvan , Jean Vitor , josephsw , Lich , matiaslauriti , Skyblade , Thulani Chivandikwa , Iolæz əɥɫ qoq , Woltus , yummypasta
22	quaternions	matiaslauriti , Tiziano Coroneo , Xander Luciano , yummypasta
23	raycast	driconmax , Meinkraft , Skyblade , user3570542 , volvis , wouterrobot
24	Realtà virtuale (VR)	4444 , Airwarfare , Guglie , pew. , Pratham Sehgal , tim
25	risorse	glaubergft , MadJlzz , Skyblade , Venkat at Axiom Studios
26	ScriptableObject	volvis
27	Singletons in Unity	David Darias , Fehr , James Radvan , JohnTube , matiaslauriti , Maxim Kamalov , Simon Heinen , SP. , Tiziano Coroneo , Umair M , volvis , Zze ,
28	Sistema audio	R4mbi , Iolæz əɥɫ qoq
29	Sistema di input	Programmer , Skyblade , Iolæz əɥɫ qoq
30	Sistema di interfaccia utente (UI)	Helium , matiaslauriti , Maxim Kamalov , Programmer , RamenChef , Skyblade , Umair M
31	Sistema di interfaccia utente grafica in modalità immediata (IMGUI)	Skyblade , Soaring Code
32	Sviluppo multiplatforma	user3797758 , volvis
33	tag	Arijoon , Augure , glaubergft , Gnemlock , MadJlzz , Skyblade ,

		Trent
34	Trasformazioni	ADB , Jean Vitor , matiaslauriti , S. Tarık Çetin , Skyblade , Thundernerd , Xander Luciano
35	Trovare e collezionare GameObjects	Pierrick Bignet , S. Tarık Çetin , volvis
36	Unity Animation	4444 , Fiery Raccoon , Guglie
37	Unity Lighting	F̄l̄ámínġ óm̄bíé
38	Unity Profiler	Amitayu Chakraborty , ForceMagic , RamenChef , Skyblade
39	Utilizzo del controllo del codice sorgente 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 , F̄l̄ámínġ óm̄bíé , Gnemlock