

Premessa

“UNITY 3D tutorials – Volume 1” è un pdf contenente le versioni testuali dei primi 30 **videotutorial su UNITY (versioni 3.x e 4.x)** pubblicati gratuitamente da Francesco Milanese sul suo canale Youtube e sul sito www.francescomilanese.com .

L'ebook, che consta di 323 pagine in formato A4, ha un prezzo volutamente basso perché si rivolge a chi ha seguito i videotutorial gratuiti (in particolare, tutti quelli pubblicati prima del 2015), li ha trovati interessanti e desidera avere un testo in pdf (con screenshots) da poter consultare facilmente, mediante le funzioni di ricerca o stampandolo... oppure a chi desidera ringraziare in maniera concreta l'autore dei video, ottenendo comunque una piccola raccolta di tutorial su alcuni principi di base e operazioni effettuabili con UNITY 3D.

SOMMARIO

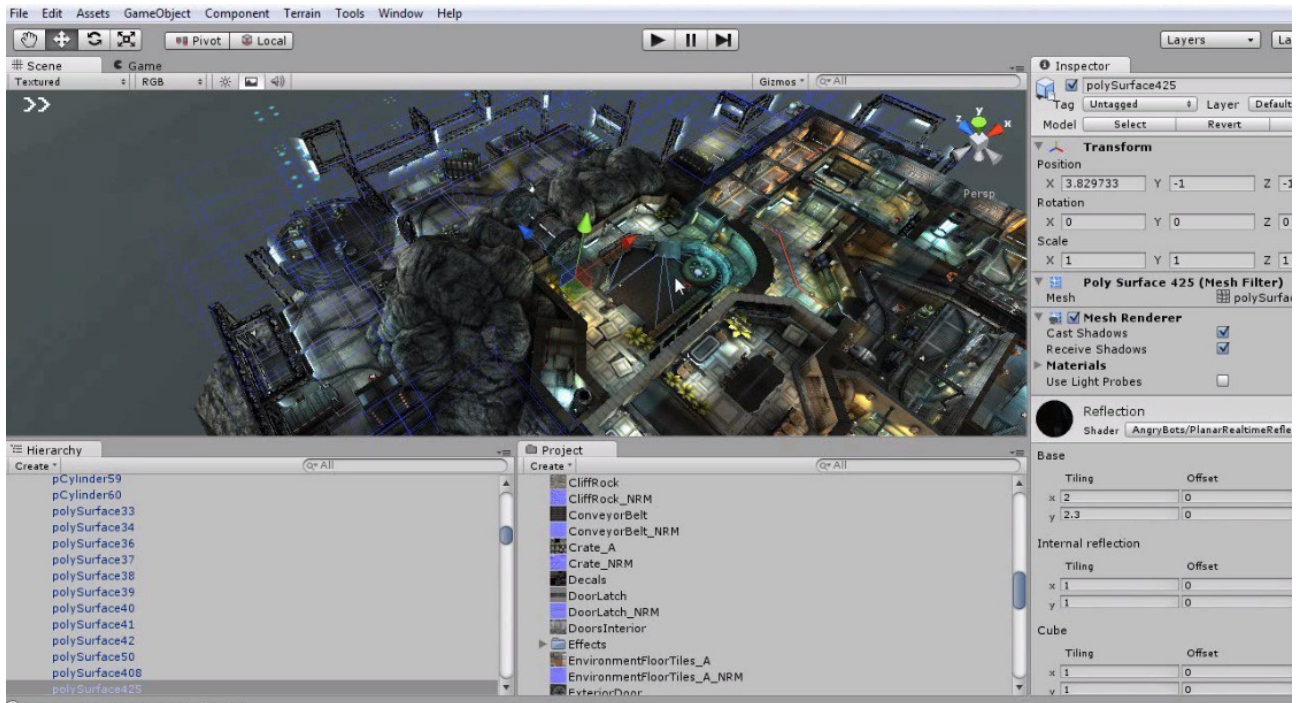
1.	Le basi di Unity 3D 3.x e 4.x – 1: Introduzione, interfaccia	1
2.	Le basi di Unity 3D 3.x e 4.x – 2: Prefabs, trasformazioni, global vs local	8
3.	Le basi di Unity 3D 3.x e 4.x – 3: importare Assets e Packages; Terrain; illuminazione	23
4.	Le basi di Unity 3D 3.x e 4.x – 4: un semplice script Javascript in Unity: countdown	40
5.	Le basi di Unity 3D 3.x e 4.x – 5: Physics Materials; pressione dei tasti; movimenti	55
6.	Le basi di Unity 3D 3.x e 4.x – 5A: extra sulla programmazione (IF-ELSE)	75
7.	Le basi di Unity 3D 3.x e 4.x – 6: istanziamento Game Objects mediante script	83
8.	Le basi di Unity 3D 3.x e 4.x – 7: Salvataggio e caricamento dati su disco (pc, tablet, ...)	97
9.	Le basi di Unity 3D 3.x e 4.x – 8: tocco delle dita su touch; click del mouse su pc	110
10.	Tips & Tricks: Coroutine (script Javascript)	121
11.	Script JS: eliminare Game Objects dopo n secondi	126
12.	Le basi di Unity 3D 3.x e 4.x – 9: funzioni mouse buttons; Materials, Shaders, Textures	129
13.	Le basi di Unity 3D 3.x e 4.x – 10: FPS Character Controller; rotazioni; cambiare scena	141
14.	Il Car Tutorial – Parte 1	154
15.	Il Car Tutorial – Parte 2	175
16.	Il Car Tutorial – Parte 3	187
17.	Unity + Blender: rotazione con parenting e script Javascript	215
18.	L'operatore modulo, Time, renderer e "l'effetto intermittenza"	227
19.	Script JS: semplice Constraint di posizione per seguire un oggetto	231
20.	Script JS: Time.deltaTime: a cosa serve, come si usa	236
21.	Script JS: Quaternion, Slerp e le rotazioni animate	240
22.	Algoritmo (e script JS) per mescolare le carte – I parte	247
23.	Algoritmo (e script JS) per mescolare le carte – II parte	255
24.	Muri a spinta – I parte	269
25.	Muri a spinta – II parte	283

26.	Riprodurre dei file audio in seguito a delle collisioni (triggering)	297
27.	Come capire se un oggetto è inquadrato dalla telecamera	305
28.	Script JS: accedere ad un Array definito in un altro script	309
29.	Tips & Tricks: dove si trovano le impostazioni dell'Anti-Aliasing (AA)	313
30.	Unity + GIMP: creare delle Height Maps / Parallax Maps	315

Le basi di Unity 3D 3.x e 4.x – 1: Introduzione, interfaccia

Questo è il primo tutorial su Unity 3D, realizzato utilizzando la versione 3.5 ma la maggior parte dei concetti espressi in questo mini-corso introduttivo sono validi anche per le versioni 4.x e per certi moduli della versione 5; in generale, come introduzione alla “logica” del programma, vanno più che bene. Per prima cosa, ovviamente, **avviamo Unity 3D**.

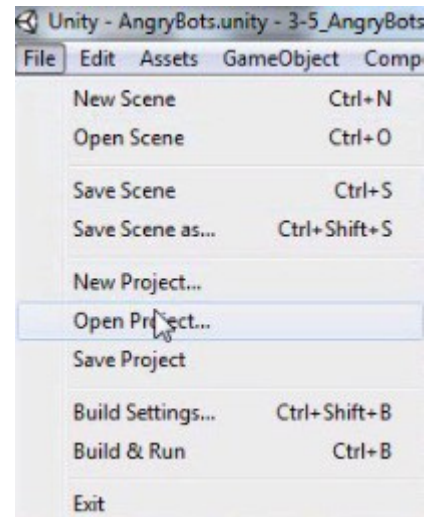
Di default, **Unity** dovrebbe avviarsi con il **progetto AngryBot**, fornito di serie con l’installazione;



qualora questo non dovesse succedere, possiamo **caricare tale progetto con Open Project** e scegliendolo nella **cartella UnityProjects**, all'interno della cartella del programma su disco.

Diamo un'occhiata al **layout della schermata**: notiamo, principalmente, una **finestra “Scene”**, che però ha anche il tab **“Game”**; una finestra **Hierarchy** (gerarchia), una finestra **Project** e una finestra **Inspector**.

Ci sono poi, in alto, **pulsanti e selettori**, come i quattro strumenti di **navigazione** in alto a sinistra, i **selettori del pivot** e della **modalità**, gli strumenti di **esecuzione in tempo reale** e i selettori **Layers** e **Layout**.



Su questi strumenti torneremo in maniera approfondita nei prossimi capitoli; per il momento, **occupiamoci della scena** che ci si presenta con AngryBots.

In **Scene**, abbiamo intuitivamente la **rappresentazione visiva della scena corrente**.

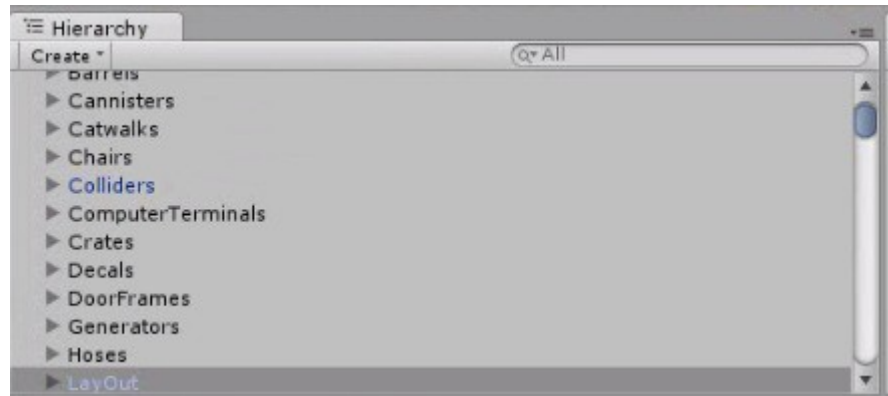


Una **“scena”**, in **Unity**, è un ambiente o, se preferite, **un livello del gioco**.

Se in un **progetto Unity** mettiamo **una sola scena**, il gioco inizierà da quella scena e non ce ne saranno altre, il che in realtà non succede mai perché si prevede sempre **una schermata di avvio**, magari con un **menù**, più una schermata di **Game Over**, una per i **punteggi migliori**, eccetera... ecco, tutte queste **schermate** vanno gestite come **“Scenes”**, **scene**, all'interno del **“Project”**, il **progetto**.

In questo momento, quindi, abbiamo una scena del progetto *AngryBots* aperta.

L'elenco degli **oggetti** che sono presenti all'interno della scena corrente si trova in *Hierarchy*, il pannello **“Gerarchia”**.

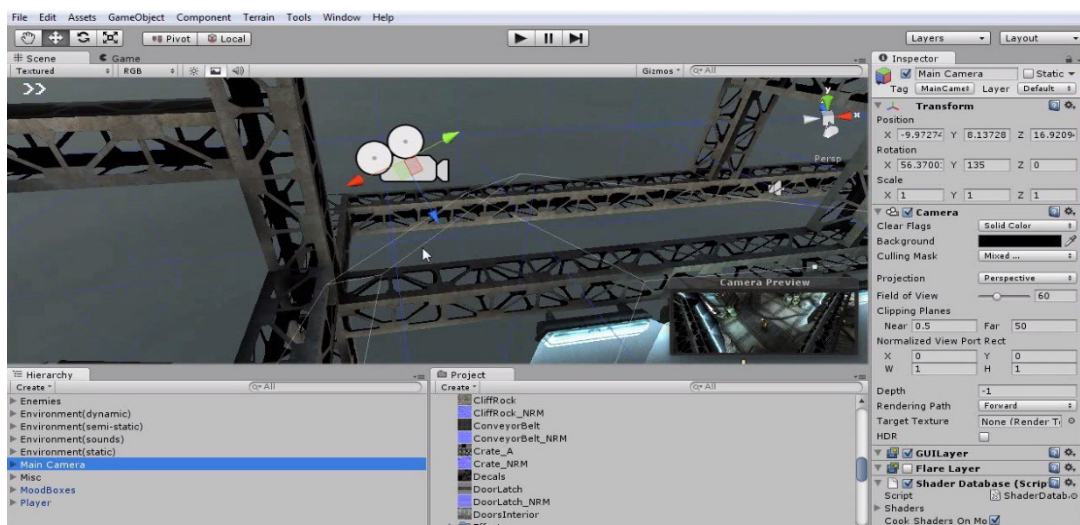


Faccio notare che **“oggetti”** non significa per forza qualcosa di fisico, visibile durante il gioco e col quale si può interagire: un oggetto ausiliario della telecamera può essere ad esempio

l'Audio Listener, un **oggetto virtuale**, che non si vede nel gioco ma che serve a **“sentire i suoni”** della scena.

In *Hierarchy*, gli oggetti possono essere resi **“figli” di un oggetto genitore**, il che ha **due vantaggi**: trasformando il genitore, ad esempio spostandolo, **si trasformeranno anche i figli**, mantenendo le distanze relative (comunque i figli manterranno una certa **autonomia**, per così dire: potranno essere spostati in maniera indipendente, senza spostare il genitore... la direzione della relazione è quindi dal genitore al figlio); il secondo vantaggio è una **maggiore chiarezza nel leggere la composizione della scena**, infatti gli elementi vengono raggruppati in base allo scopo o funzione.

Diamo subito un'occhiata alla **pratica**: nel pannello **Gerarchia**, facciamo **doppio click su Main Camera**.

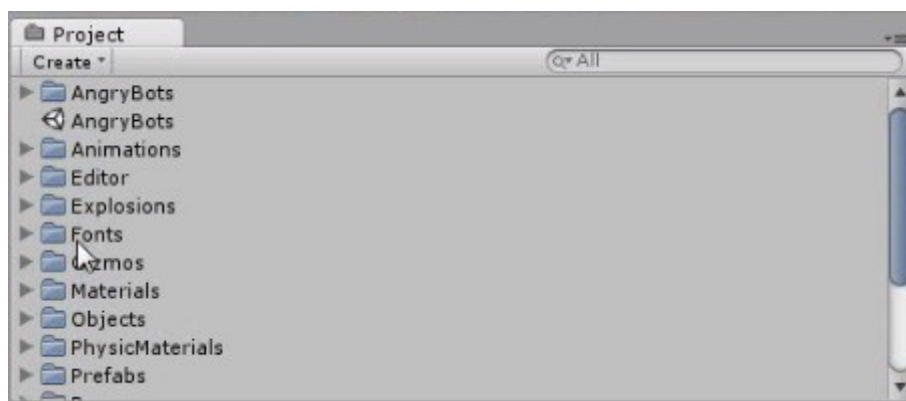


Nella vista **Scene**, Unity si sposterà automaticamente per mostrare, nella **scena**, l'**oggetto selezionato**, cioè proprio la telecamera. Nel pannello **Gerarchia**, vediamo inoltre che ci sono **due oggetti figli**, accessori della telecamera.

Nel pannello **Inspector**, notiamo che **il contenuto è cambiato** e che, in generale, cambia se si seleziona un oggetto diverso.

L'**Inspector** è, in un certo senso, la **carta d'identità dell'oggetto selezionato**: ci dice dove si trova, com'è orientato, quali sono le sue dimensioni, qual è il suo aspetto, quali script ha associati... tutto! Neanche a dirlo, lo utilizzeremo parecchio, in futuro.

Prima di passare alla **navigazione nella 3D View** e soprattutto all'**anteprima real time**, parliamo dell'ultimo tassello che ci manca: **la finestra Project**. Che cosa c'è qui dentro? A che serve?



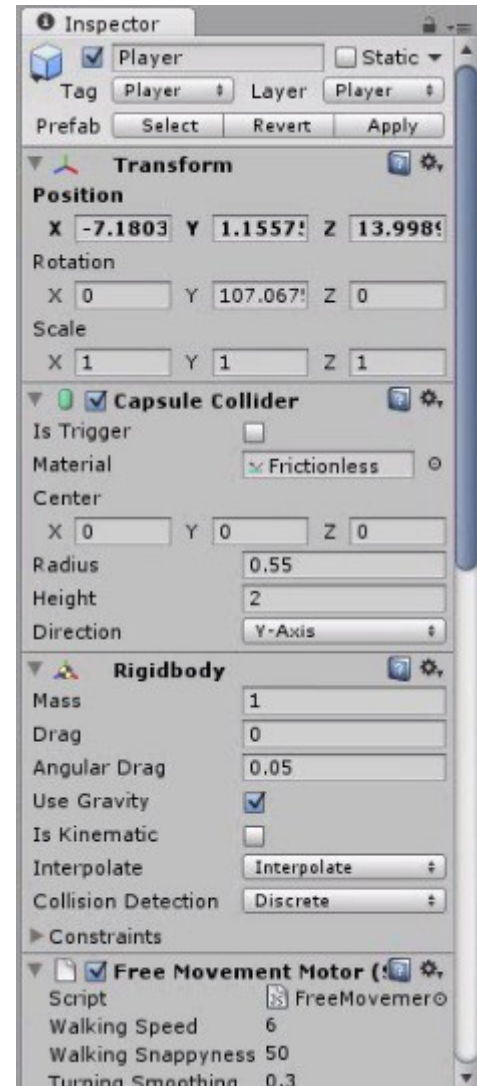
Ecco, qui ci vanno **tutti gli elementi**, fisici e non, del progetto, che **POTREBBERO finire in una scena o livello del gioco**; non solo: gli elementi qui presenti non devono essere per forza **Objects**, oggetti, ma anche loro **caratteristiche**, oppure **caratteristiche del progetto stesso**.

Questi oggetti sono raggruppati in **cartelle** e queste cartelle e i loro elementi hanno degli elementi **corrispondenti nel disco fisso**, all'interno della **cartella del progetto**, quindi se volete inserire, ad esempio, **un'immagine texture o un modello 3D nel progetto**, trascinate il file qui, in **Project**: non trasferite i file direttamente nella cartella su disco fisso.

Chiariamo un po' il concetto di **attributi** e **caratteristiche**: ad esempio, un personaggio avrà una **forma geometrica (il modello)**, un **Material** (che ne definisce l'ombreggiatura), una **Texture** (per l'aspetto) e qualche **script** per i comandi o l'**intelligenza artificiale**; ebbene, nelle **cartelle Materials** troviamo prototipi di **materiali**, in **Scripts** troviamo gli **script di programmazione** disponibili nel progetto, poi ad esempio in **Scenes** troviamo le **scene**, i “**livelli**”, presenti nel nostro progetto, e così via.

Ricapitolando:

- Quando si crea un gioco si crea un **Progetto**.
- Nella finestra **Projects** vanno tutti gli elementi, detti **Assets**, che serviranno per creare gli oggetti del gioco, inclusi i livelli, gli script di programmazione ed altri elementi “intangibili”.
- Un **Progetto** è fatto di più **Scene**, i livelli o ambienti del gioco.
- Nel pannello **Gerarchia**, troviamo gli oggetti, fisici e non, che si trovano all'interno della **Scena** corrente.
- Per ciascun **oggetto**, le **caratteristiche**, i parametri e altre informazioni utili si trovano nell'**Inspector**.



A tutto questo va legato il discorso dei **Prefabs**, che però tratteremo nel prossimo capitolo, per evitare di mettere troppa carne al fuoco qui... prima di chiudere questo capitolo, invece, diamo un'occhiata alla **navigazione nella finestra di gioco**.

Per **spostarci nella scena**, senza muovere la telecamera (il nostro **punto di vista in modalità editing** è una cosa diversa dalla **vista telecamera in fase di gioco**, visibile con la schermata **Game**), selezioniamo lo strumento con l'**icona della mano**, in alto a sinistra.



Con il **mouse nella 3D View**, ruotiamo la **rotellina** del mouse: questo è lo **zoom**, che si può ottenere anche (sempre con l'icona della mano selezionata) con **ALT e click del tasto destro del mouse**, tenendoli entrambi premuti e muovendo il mouse.

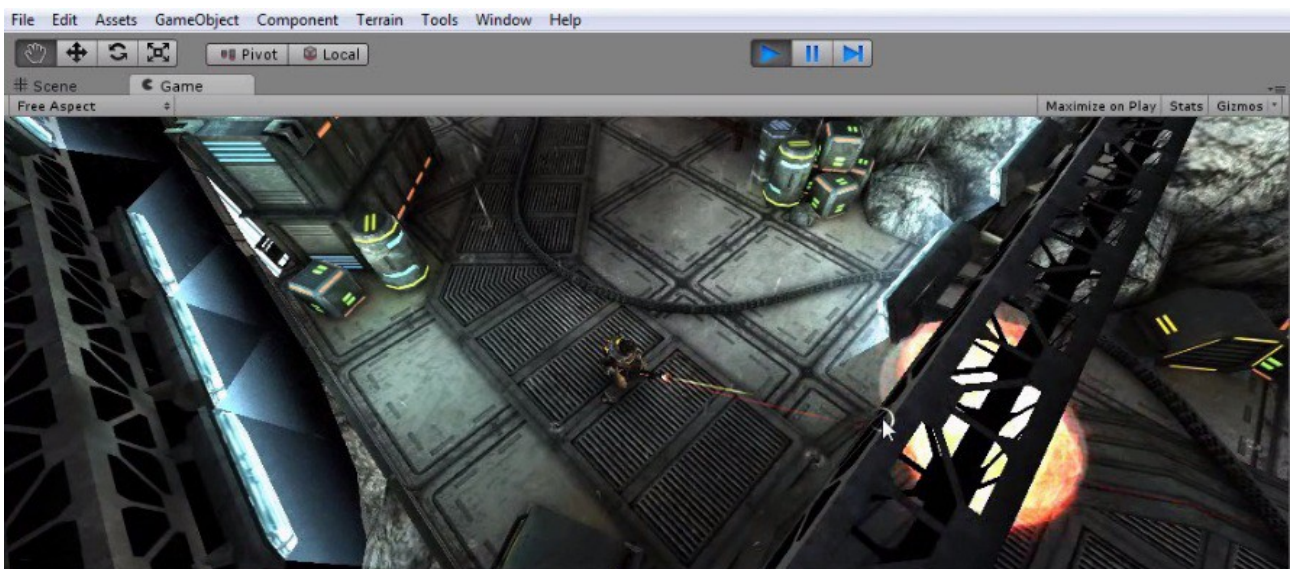
Cliccando il **tasto sinistro**, oppure la **rotellina o il tasto centrale** del mouse, e muovendo il mouse, abbiamo invece il **panning o spostamento** nella scena.

Premendo, infine, **ALT e il tasto sinistro del mouse**, e muovendo il mouse tenendoli premuti, si ha la **rotazione della vista**.

Come vi dicevo, questi strumenti **riguardano la vista libera**, nell'editor del gioco, ma **non la vista telecamera**, che sarà invece il nostro punto di osservazione **durante l'esecuzione...** già, ma come si vede **ciò che vede la telecamera**?

È sufficiente scegliere il **tab GAME** anziché **SCENE**. Qui, i comandi di **panning, zoom e rotazione** non valgono più, perché la vista non è più libera, ma è quella dell'oggetto.

È giunto finalmente il momento di **provare un po' di interattività**: con la vista **Game**, facciamo click sul pulsante **Play** al centro in alto nella finestra, e... **attenzione al robot che si avvicina subito a noi!**



Potete **orientare il giocatore muovendo il mouse e sparare con il tasto sinistro**, inoltre potete muovere il giocatore con i tasti **WASD**, quelli tipici dei **first person shooter**.

Per **terminare l'esecuzione a runtime**, cliccate di nuovo sul pulsante **Play**; *non cliccate su Pausa*: c'è una differenza di fondo tra l'utilizzo di questi due pulsanti, ma ne parleremo un'altra volta.

Il giocatore è quindi provvisto di un **controller che registra l'attività di tasti e mouse e lo sposta, lo ruota e gli fa fare fuoco**; inoltre, **la telecamera segue il giocatore** e i **nemici si attivano** e gli vanno incontro se il giocatore si avvicina troppo.

Tutte queste operazioni si basano sullo **schema evento-azione**, noto ad esempio nel **Blender Game Engine** come **schema Sensore-Controller-Attuatore**, e in questa serie di tutorial esploreremo gli strumenti e le configurazioni principali per realizzare i nostri giochi.

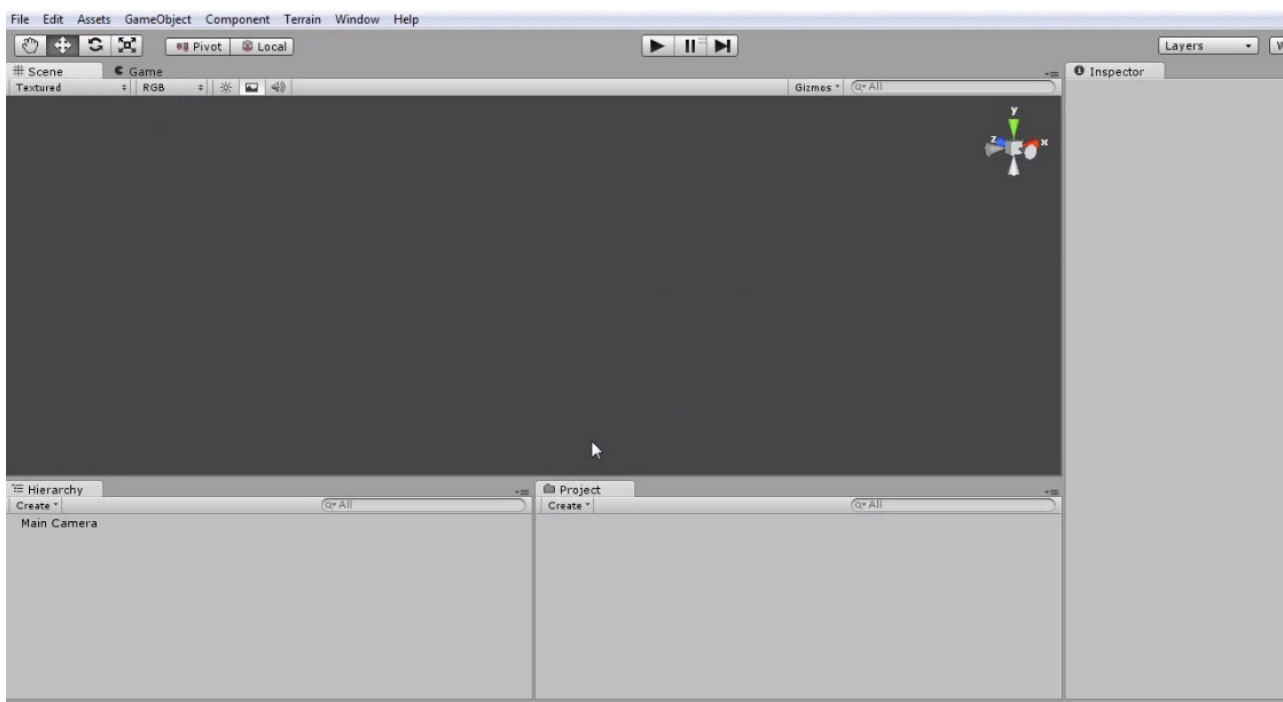
Per questo capitolo, comunque, è meglio fermarsi qui... nel prossimo, inizieremo da un progetto vuoto per parlare di **Prefabs**, tasti **Play-Pausa** e **inserimento di oggetti nella scena 3D**.

* * *

Le basi di Unity 3D 3.x e 4.x – 2: Prefabs, trasformazioni, global vs local

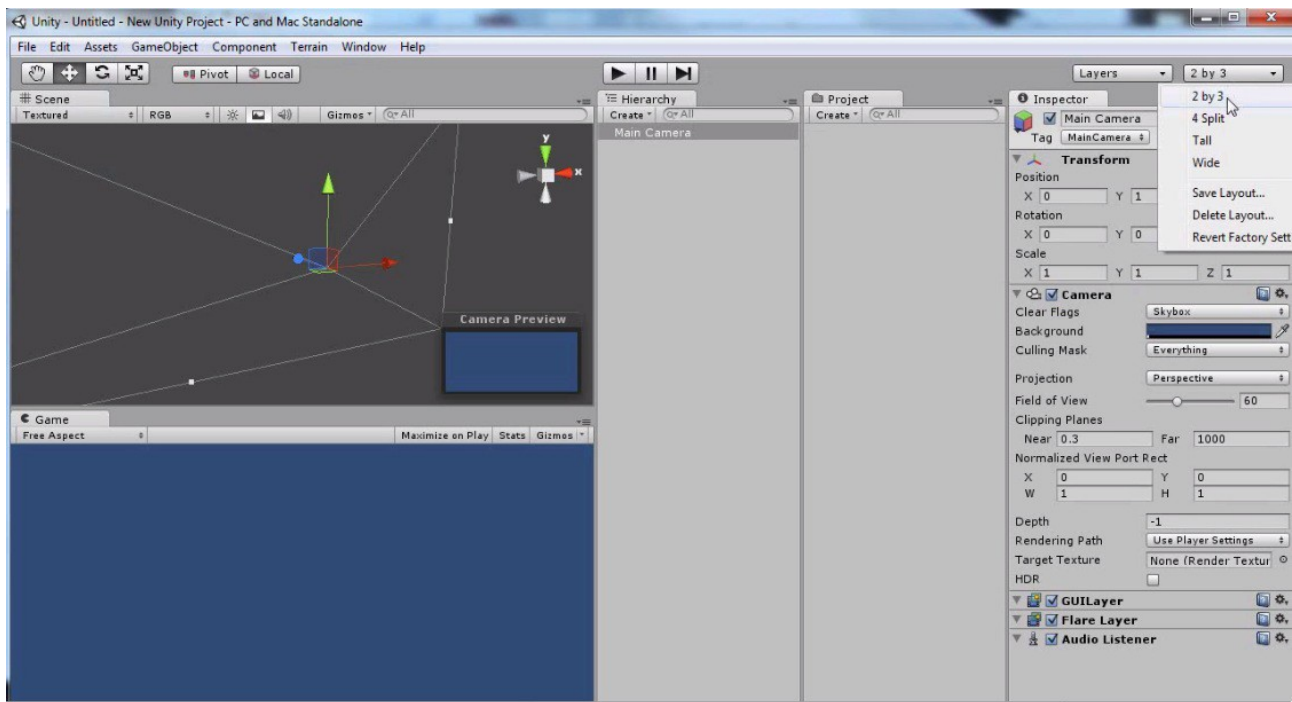
In questo secondo capitolo sulle **basi di Unity** parleremo dei **Prefabs**, degli oggetti di scena e del loro spostamento. Prima di iniziare, però, una considerazione sul **Layout** della schermata.

La disposizione delle finestre nel mio progetto prevede, come visto nel capitolo precedente, una finestra con due tab **Scene / Game** al centro, l'**Inspector** a destra e, sotto a sinistra e al centro, la **Gerarchia** e il **Progetto**.



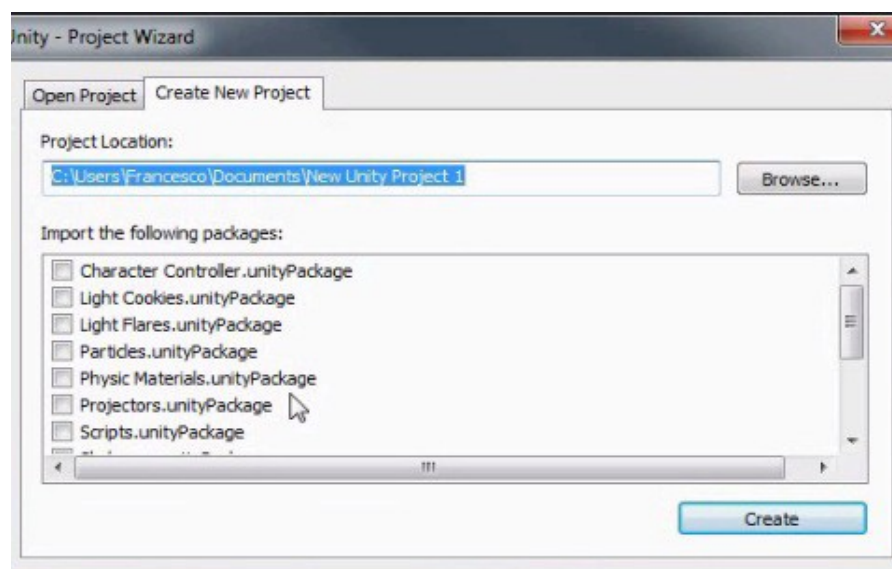
Ho scelto questa impostazione perché mi consente di avere **una vista in formato Wide della scena**, inoltre l'**Inspector** ha i parametri disposti in sezioni in **colonna**, quindi così offre una buona visualizzazione (chi usa **Blender 3D dalla versione 2.5** può pensare, ad esempio, alle posizioni di **3D View, Timeline e Properties Window**).

Questo tipo di **Layout** è detto **Wide** e può essere selezionato dal menù **Layout**, in alto a destra; qui, potete scegliere tra gli altri **Layout** standard forniti da **Unity**, come **2by3**, **4 split** e **Tall**, oppure definire **un vostro layout** cambiando posizione e disposizione delle finestre (fate click sull'icona in alto a destra in una finestra e scegliete **“Close Tab”** o **“Add Tab”** per aggiungere altri tabs), salvando poi il **Layout** personalizzato con **Save Layout**, sempre nel menù **Layout**.



Ok, fine del discorso sui **Layout**, passiamo agli argomenti principali di questo **tutorial**.

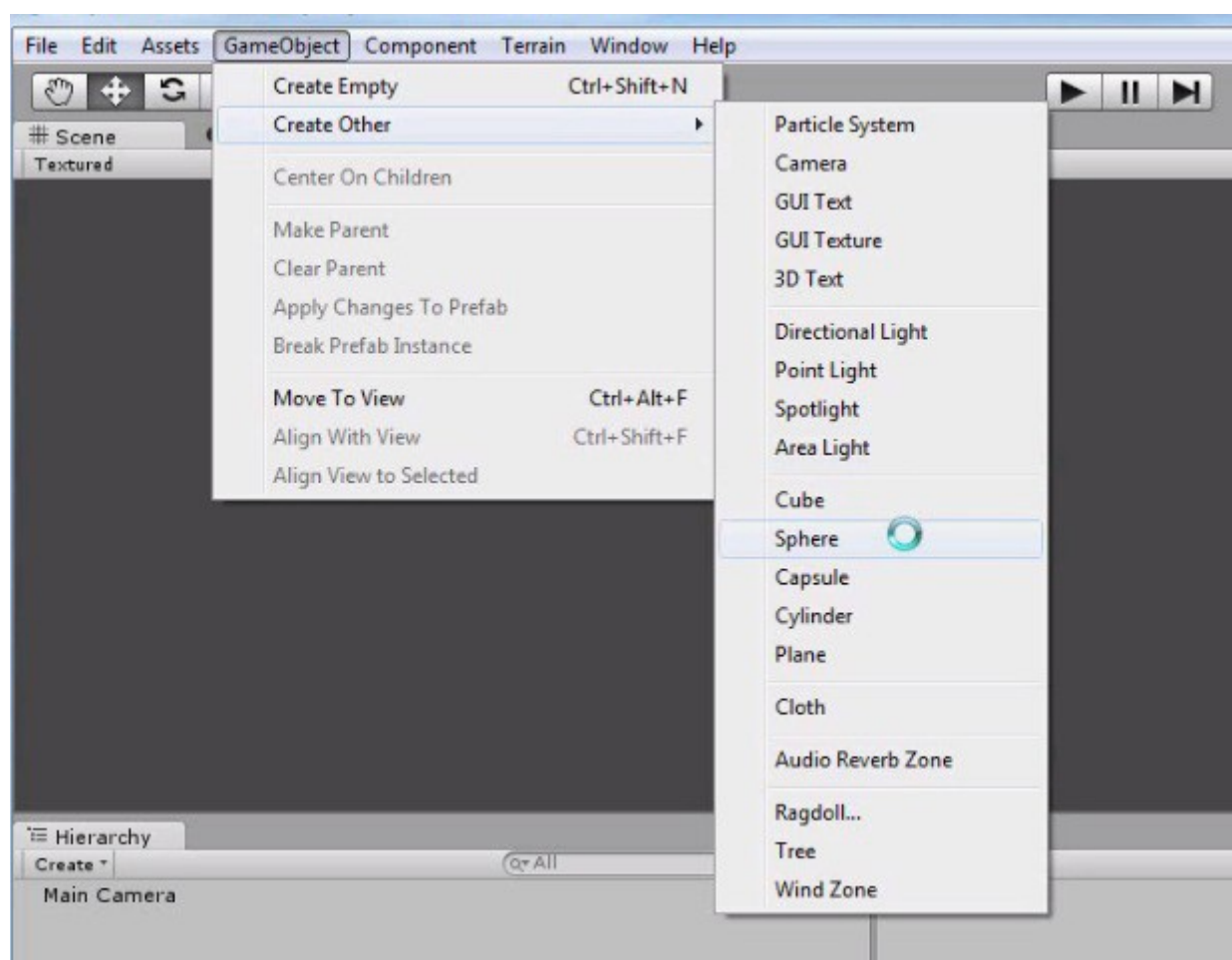
Iniziamo con un **progetto vuoto**, quindi per prima cosa scegliamo **New Project** dal menù **File**, impostiamo il percorso e il nome del progetto lasciando deselezionate le caselle presenti nella schermata di creazione del progetto e confermiamo.



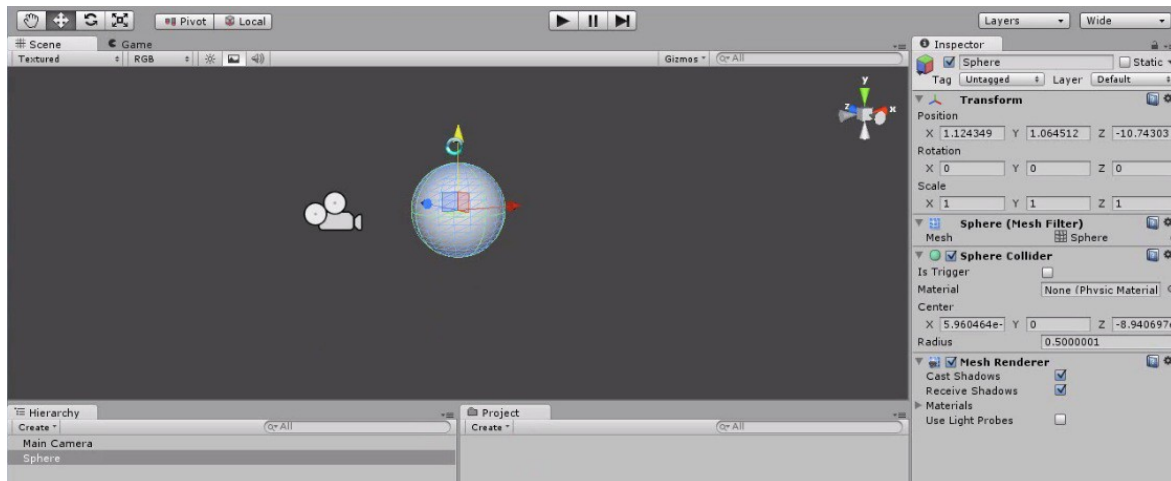
Seguirà un **riavvio del programma**: Unity ripartirà con il **nuovo progetto** pronto per essere utilizzato.

Per prima cosa, apriamo il menù **GameObject** e scegliamo **Create Other**: si aprirà un sottomenù, dove potremo scegliere se aggiungere, ad esempio, una **telecamera**, una **fonte di luce Point**, una **direzionale**, una **sfera**, un **cubo**, eccetera... notate che le voci sono **suddivise in gruppi a seconda della loro natura**: un oggetto geometrico, o mesh, oppure una fonte di luce, un simulatore (c'è ad esempio **cloth**, quello dei tessuti), e così via.

Scegliamo **Sphere**, la sfera: al click, **una sfera verrà inserita nella scena**, in una posizione che possiamo conoscere dando un'occhiata all'**Inspector**... tra poco, per chi non ha mai usato un programma di **grafica o editing 3D**, spiegherò cosa vogliono dire **X, Y e Z**.

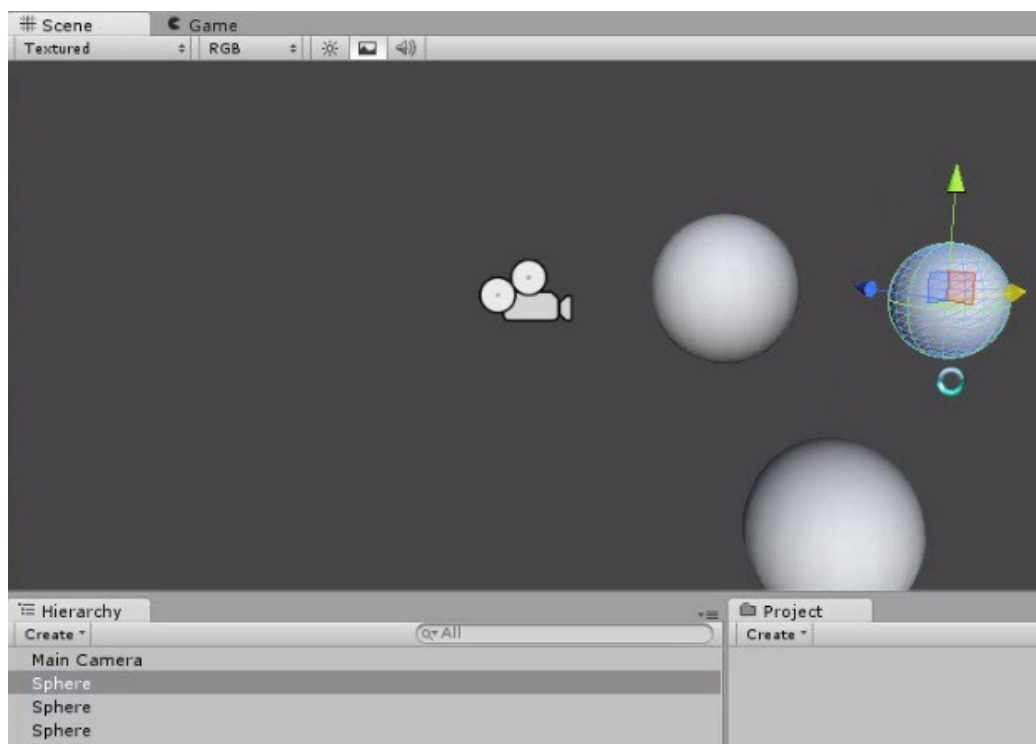


Notiamo che nel pannello **Gerarchia** c'è la nostra sfera, ma nel pannello **Progetto** non c'è nulla.



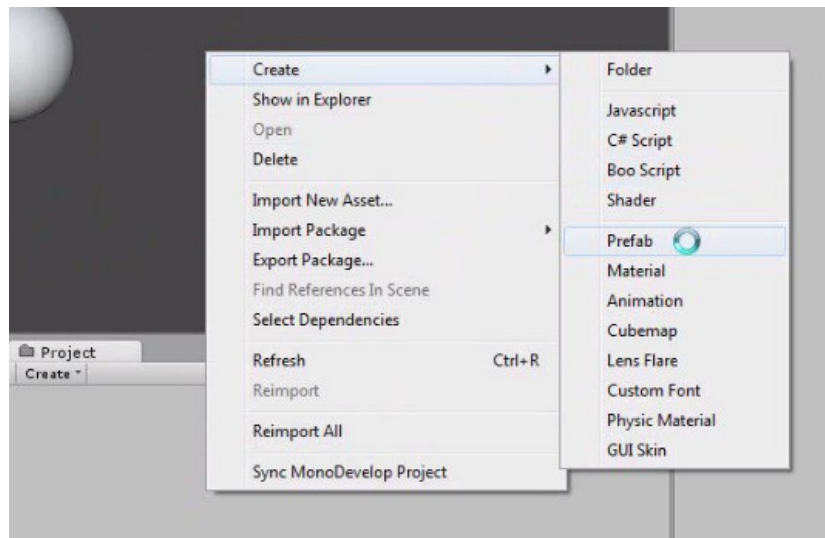
La **sfera** è stata aggiunta alla scena ed è interamente modificabile a parte, ma non c'è un suo **prototipo** nel progetto, ossia un **modello** dal quale creare altre sfere che **condividano** gli stessi materiali, gli stessi script, eccetera.

Significa che, ora come ora, se volessimo inserire **altre due sfere** da rendere identiche alla prima (per quanto riguarda, ad esempio, l'aspetto visivo o le proprietà di rimbalzo), l'inserimento di **due Game Object Sphere** produrrebbe **due oggetti distinti**, per un totale di tre sfere, completamente diverse tra loro. Per assegnare a tutte, ad esempio, lo stesso script, **dovremmo linkarlo tre volte**, una volta per ogni sfera... e, se in seguito volessimo aggiungerne un altro o modificarlo, la stessa cosa... e qui sono solo tre sfere, ma immaginate 100 porte o 100 finestre in uno scenario virtuale!

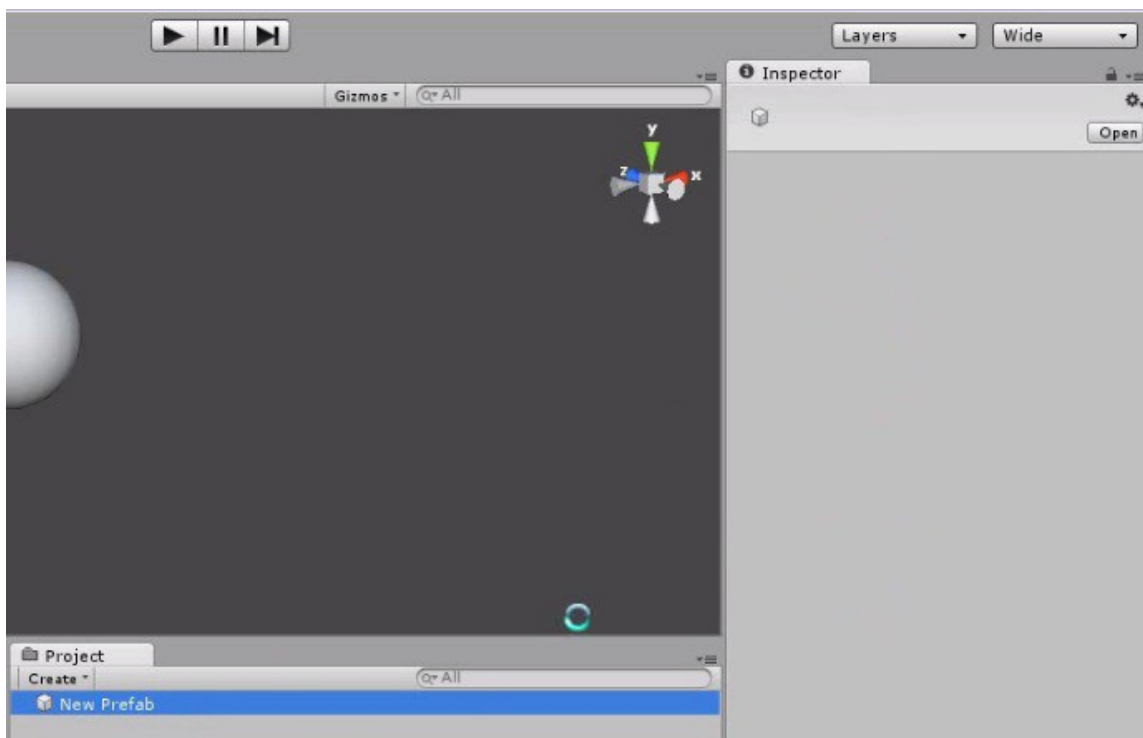


Facciamo ora **click destro sulla finestra Project** e scegliamo di creare un **Prefab**.

Apparentemente non è successo nulla, a parte l'inserimento di una voce **“Prefab”** nel pannello **Progetto**, comunque a questa voce non è associato niente, come si può vedere dando un'occhiata all'**Inspector** dopo averla selezionata (immagine sotto).



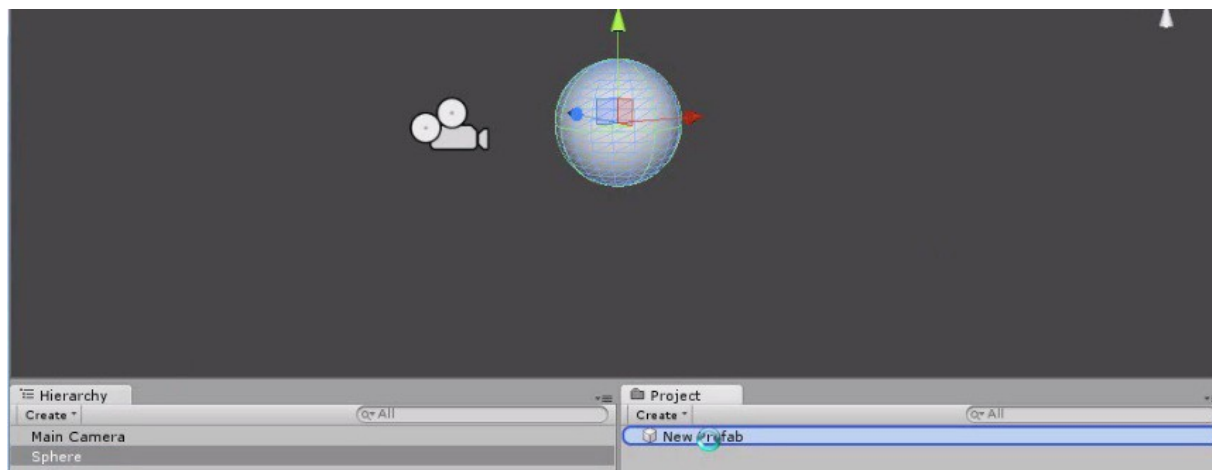
Un **Prefab** è un **prototipo...** di qualsiasi cosa: può essere un prototipo di un oggetto, di una fonte di luce, di una fonte sonora... davvero **qualsiasi cosa**.



Gli **oggetti di gioco** creati a partire dal prototipo saranno **ISTANZE** di tale **prototipo**, cioè resteranno legate, dal punto di vista dei **parametri** e delle **caratteristiche**, alle caratteristiche del **prototipo**.

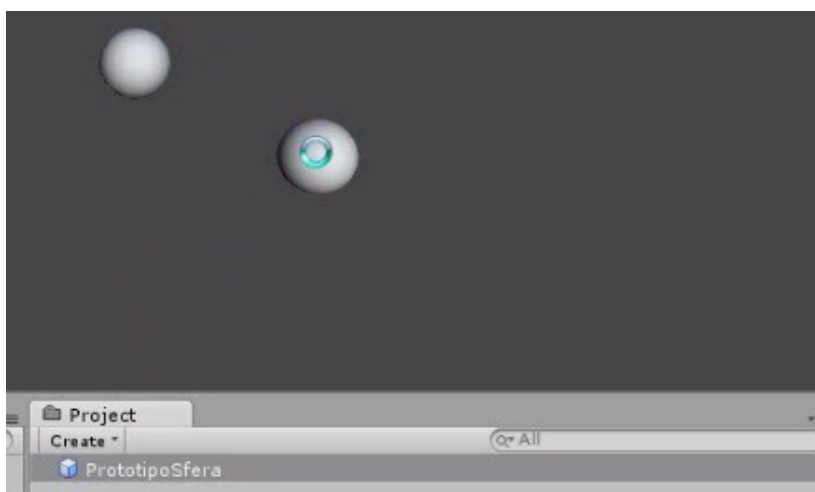
Agganciando quindi, per esempio, un **Material**, una **Texture** o uno **script** ad un **prototipo**, ritroveremo questi elementi automaticamente **in tutti gli oggetti** di gioco creati a partire da quel prototipo.

Per realizzare un **prototipo** della nostra **sfera**, selezioniamola con un click dalla finestra **Gerarchia** e, tenendo premuto il tasto sinistro, **trasciniamo** sulla voce **Prefab** in **Project**, voce che ora possiamo rinominare ad esempio **“PrototipoSfera”**.



L'utilizzo dei **Prefabs** rende il progetto più leggibile, **più facile da modificare**, più ordinato ma anche **più leggero** a livello di memoria (sia su disco che RAM, durante l'esecuzione) occupata, perché le **istanze** sono, appunto, istanze di un **prototipo**, quindi non c'è un'inutile **replicazione di dati** pressoché uguali.

Tornando al nostro progetto, l'inserimento delle copie della sfera nella **scena 3D** va fatto cliccando su **PrototipoSfera** e **trascinando** nella finestra **Scene**, una volta per ogni copia.



Gli elementi **appariranno** anche nella **finestra Gerarchia**

ma questa volta i loro nomi sono scritti in **blu**, visto che si tratta, come detto, di **istanze di un Prefab**, mentre ad esempio l'oggetto **telecamera** è scritto in nero.

Ok, gli **oggetti** sono nella scena... ma esattamente, **dove** sono?

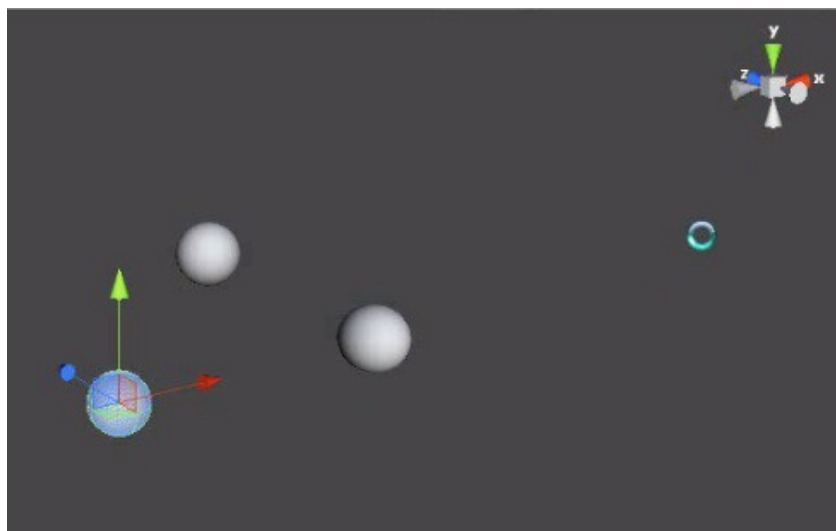
L'**universo virtuale 3D** è organizzato come un **diagramma cartesiano a 3 dimensioni**; nei piani cartesiani studiati a scuola, ci sono l'asse delle X e quello delle Y, quindi possiamo dare delle coordinate di posizione agli oggetti specificando questi due valori e calcolare lunghezze e distanze con le apposite formule.

Nel **3D**, al piano si aggiunge una **terza direzione**, perpendicolare, per ottenere la **terza dimensione** (che qui è la profondità).

In **Unity**, il piano che fa da **pavimento** è detto **piano XZ**, dal nome dei due assi, tra loro perpendicolari, che lo compongono, mentre l'**asse verticale è quello delle Y**.

Come vi dicevo, il terzo asse, Z, è la profondità. Date un'occhiata all'**icona in alto a destra**, che rappresenta appunto tale sistema: X è l'asse orizzontale, Y quello verticale, quindi abbiamo il piano XY come nel grafico cartesiano, in più c'è l'asse della profondità, Z.

Questi assi hanno origine da un punto, detto appunto **Origine dell'universo virtuale**, che ha **coordinate X, Y e Z pari a 0.0, 0.0 e 0.0**; inoltre, gli assi hanno un **verso**: X verso destra è positivo (notate la freccia rossa), Z verso avanti è positivo (freccia blu) e Y verso l'alto è positivo (freccia verde).

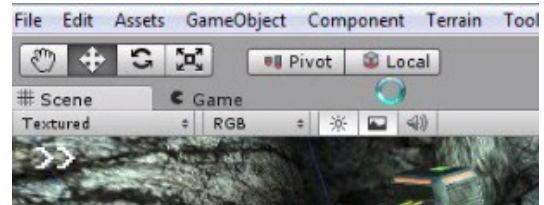


Da notare che i coni e il cubo nell'icona in alto a destra in **Scene** sono cliccabili: cliccando ad esempio su **Y superiore**, la **vista libera** nella scena 3D si sposterà automaticamente per **inquadrare la scena 3D dall'alto** (Y superiore) verso il basso (il cono di Y superiore punta verso il cubo) in **modalità ORTOGRAFICA**, senza effetto **prospettiva**; allo stesso modo, cliccando ad esempio su

X rossa, a destra, avremo una vista da destra **verso sinistra ORTOGRAFICA** (fate riferimento sempre al verso del cono per capire qual è il verso di osservazione), mentre con **un click sul cubo** torneremo alla **vista libera in PROSPETTIVA**.

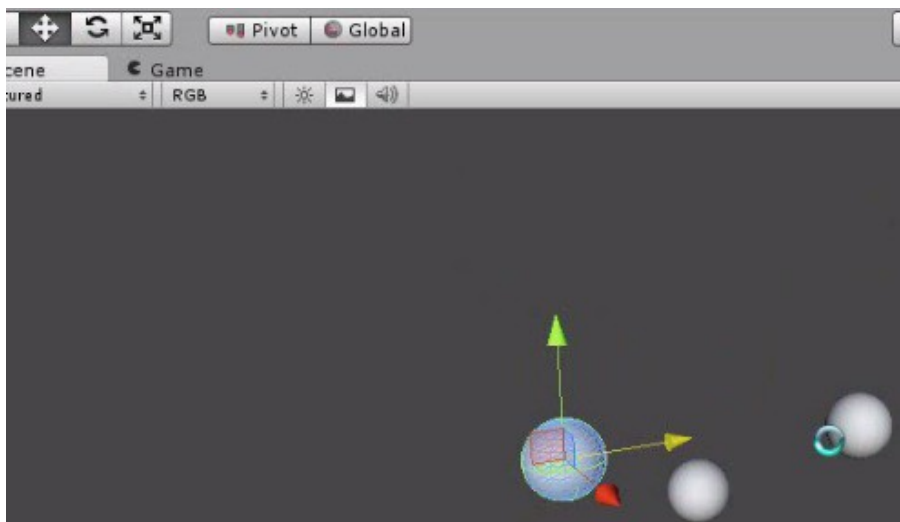
Le **viste ORTOGRAFICHE** servono per avere delle **viste parallele e senza effetti di prospettiva** (come quelle delle **proiezioni ortogonali** nel disegno tecnico e nel CAD) degli oggetti nella scena, mentre la **vista libera** è quella in **prospettiva**, la vista del gioco (in futuro vedremo come impostare la **modalità della telecamera di gioco tra PERSP e ORTHO**, quest'ultima molto utile per i **giochi Platform o 2D**).

Queste informazioni fanno riferimento al **sistema di coordinate GLOBALE**; nella seconda parte del capitolo, parlerò delle **differenze tra sistema GLOBALE e LOCALE**, per cui per ora vi chiedo solo, in fiducia, di scegliere la **modalità GLOBAL** nel **selettore GLOBAL/LOCAL** in alto a sinistra nella schermata, con un click del tasto sinistro.



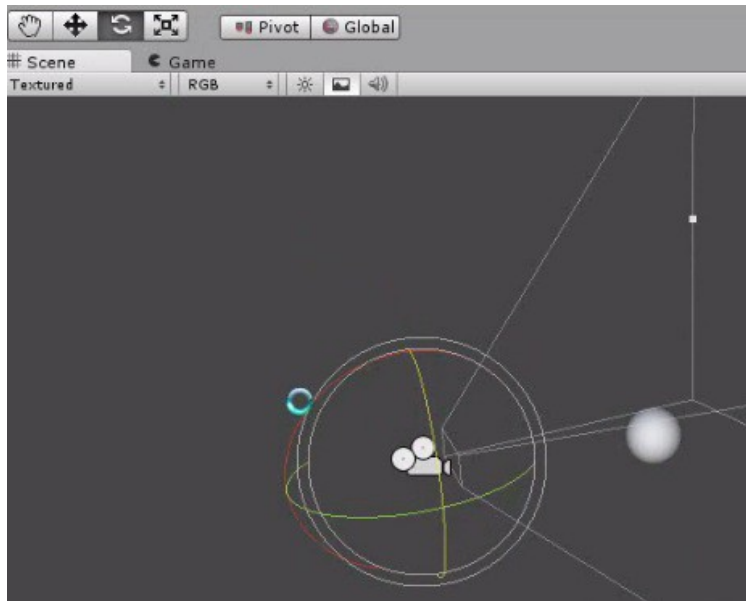
Ecco quindi che **Position XYZ** nella finestra **Inspector** ha un senso... e, come avrete intuito, scrivendo lì un **valore** possiamo posizionare con **estrema precisione** gli oggetti, in maniera detta **“parametrica”**.

Per **spostarli liberamente nella scena 3D**, invece, selezioniamoli in **Scene**, quindi selezioniamo l'icona **“Move the Selected Object”** in alto a sinistra, facciamo click sull'asse che ci interessa e, tenendo premuto, trasciniamo. Ecco la **traslazione**.

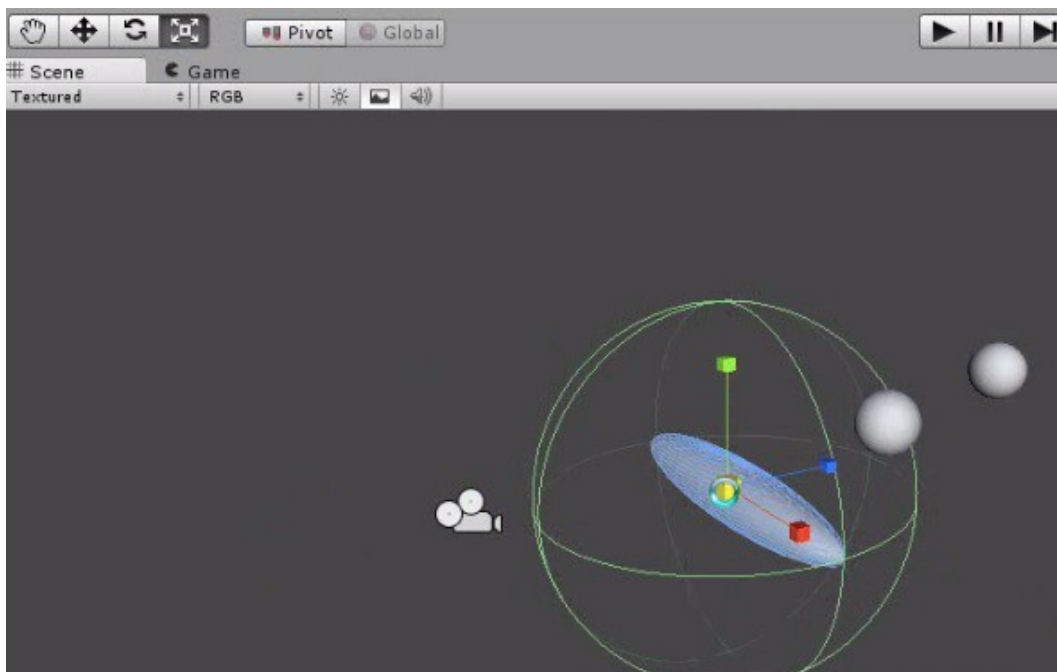


Le **frecche colorate** sono dette **manipolatori di trasformazione**, quindi in questo caso abbiamo il manipolatore di traslazione con le 3 frecce relative ai vari assi, qui in **modalità GLOBAL**.

Intuitivamente, **clickando sull'icona Rotate** in alto a sinistra entreremo in modalità **rotazione**, che tratteremo meglio tra pochissimo, visto che bisogna stare attenti agli assi e alla **modalità GLOBAL/LOCAL...**



... mentre con **Scale** avremo il **ridimensionamento**, che può essere fatto o **su un solo asse**, cliccando su uno dei quadrati del manipolatore di scaling, oppure **su tutti e tre gli assi in maniera uniforme** cliccando sul cubo grigio al centro.



Si tratta di **operazioni intuitive**: fate delle prove voi stessi per prendere **confidenza** con questi strumenti.

Ricordate che potete **cambiare le viste**, cosa molto utile per **posizionare correttamente gli oggetti** rispetto agli altri, perché la **prospettiva** gioca brutti scherzi se non ci sono altri **elementi di riferimento**, quindi passate ad esempio alla vista superiore, destra, eccetera.

Vi ricordo inoltre che **posizioni, orientamenti e fattori di scala** per le dimensioni possono essere specificati in maniera **parametrica** mediante l'**Inspector**.

Chiaramente, il **Prefab** definisce gli aspetti “generalisti” **comuni a tutti gli oggetti**, come ad esempio Materials o Script, ma **certi aspetti sono liberi**, come ad esempio posizioni, orientamenti e dimensioni, infatti stiamo **trasformando le istanze a piacere**.

Quello del **Prefab** non è quindi un **vincolo rigido** ma una buona base di partenza per l'ottimizzazione di certe operazioni.

* * *

Vediamo adesso un **esempio** decisamente più interessante, almeno dal punto di vista visivo, anche se incompleto per via di alcuni aspetti tecnici che descriverò a breve.

Riprendiamo **Angry Bots**, il **gioco** fornito di serie con **Unity**.

Già nel capitolo precedente abbiamo visto che la **telecamera**, qui, segue il protagonista **dall'alto**; una visuale interessante, ma personalmente mi trovo meglio con la **visuale in prima persona**.

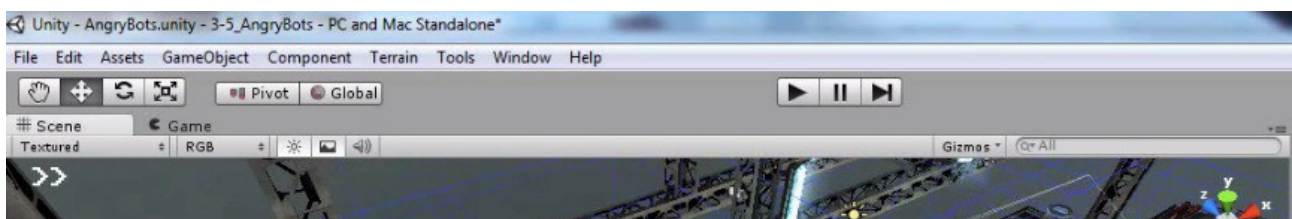
Abbiamo visto anche che la **telecamera** è agganciata al personaggio principale, infatti **lo segue sempre** mantenendo comunque la **distanza relativa**.

Spostando quindi la telecamera in fase di editing della scena, le daremo **una nuova posizione relativa rispetto al personaggio**, ma per il resto si sposterà nella misura data dal protagonista (ricordate il discorso della relazione genitore-figlio, con il figlio che si può muovere liberamente senza influenzare il genitore ma che deve seguire il genitore se questo si sposta?)... ne approfitteremo, tra l'altro, per mostrare la **differenza tra sistema di riferimento globale e locale** nelle trasformazioni.

Selezioniamo l'oggetto *Main Camera* dal pannello **Gerarchia**, con un doppio click sul nome dell'oggetto.

Il primo spostamento che vogliamo fare è in **verticale**, per portare la **telecamera** alla quota del protagonista.

Abbiamo a disposizione gli **assi del manipolatore di traslazione**, ma qui bisogna stare attenti ad utilizzare **il sistema di riferimento GLOBALE**, da selezionare nell'apposito menù in alto.



Il **sistema di riferimento globale** è quello dove l'asse **Y** è **verticale**, l'asse **Z** è **orizzontale** e l'asse **X** è **il verso avanti/dietro**, e tali assi restano sempre così, indipendentemente dall'oggetto selezionato e dal suo orientamento (sono, appunto, **assi GLOBALI, della scena**).

Quando un oggetto viene inserito nella scena, ha **un suo sistema di riferimento LOCALE**, che ci consente di individuare un avanti, un dietro, un sopra, eccetera... di default, all'inserimento nella scena, gli assi locali coincidono con quelli globali, ma non è così per tutti gli oggetti, e lo vedremo a breve con la telecamera; inoltre, **se ruotiamo l'oggetto**, ad esempio di 30° sul SUO asse orizzontale (immaginate ad esempio di ruotare la testa verso l'alto) e facciamo uno spostamento “in avanti”, questo “in avanti” avrà due effetti diversi a seconda del sistema di riferimento.



Facendo uno **spostamento in avanti sull'asse locale**, l'oggetto si staccherà dal terreno e andrà **in avanti e in alto**, perché ora punta verso l'alto con un angolo di 30° e, spostandosi secondo i suoi assi locali, adesso orientati in questo modo, **seguirà quella nuova direzione**.

Con uno **spostamento in avanti sull'asse globale**, invece, l'oggetto **resterà attaccato a terra**, anche se si manterrà **inclinato** (sarebbe camminare guardando sempre in alto, per intenderci... l'esempio reale con gli assi locali, invece, non ve lo posso fare perché, ahinoi, non possiamo volare).

Ok, una bella descrizione teorica, ma vediamo un **esempio pratico**.

Con la **Main Camera** selezionata, passiamo alla modalità **LOCAL**.

Ecco che succede: **l'asse Z locale è quello del verso dell'inquadratura**, quindi facendo click su tale asse e muovendo il mouse, faremo **avanzare la telecamera lungo la SUA direzione**, quella dell'inquadratura.

Da notare che non c'è una freccia verticale: **l'asse Y locale è perpendicolare all'inquadratura**, che è rivolta verso il basso con un certo angolo, quindi anche tale asse è inclinato.

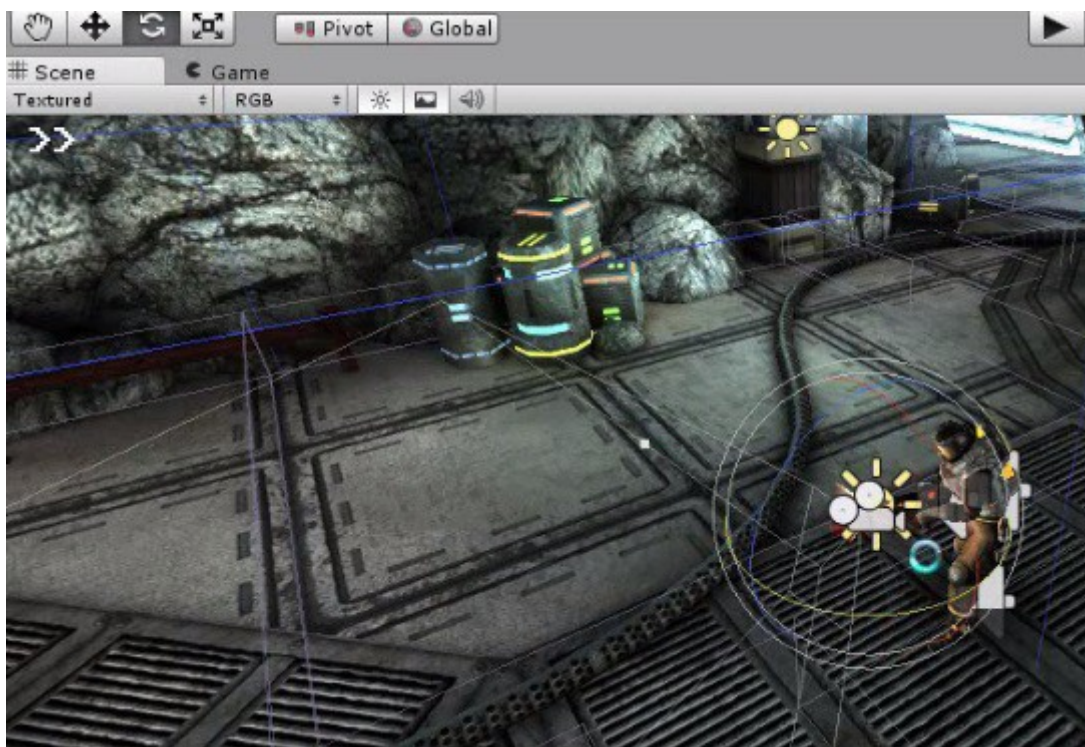
Tornando alla **modalità GLOBAL**, l'asse Z è quello della scena: uno spostamento laterale; **l'asse Y, invece, è verticale**: d'altronde, è quello globale della scena.

Per il primo movimento, lo **spostamento** in basso, ci serve il sistema **GLOBALE**: scelto tale sistema, facciamo click sull'asse e spostiamo la telecamera in basso.

Nella vista **Scene**, in basso a destra c'è la **finestra Camera Preview**, quindi lì possiamo vedere il risultato in tempo reale.

Terminato questo lavoro, sempre col **sistema globale** selezionato, facciamo degli **spostamenti** sugli assi del piano **orizzontale**, ossia **XZ**, per mettere la telecamera **leggermente avanti rispetto al protagonista**.

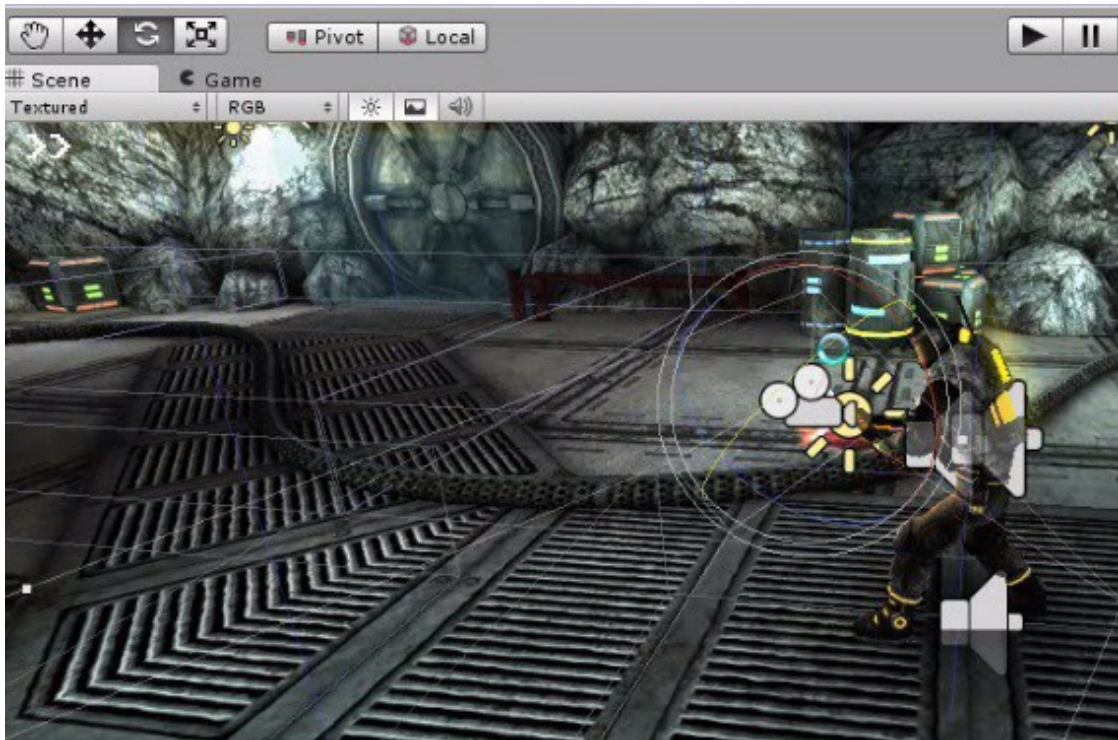
Per la **prima rotazione**, per orientare meglio la **telecamera**, usiamo sempre il **sistema globale**: passiamo allo strumento **rotazione** e ruotiamo un po' la telecamera **intorno all'asse verticale**, cliccando quindi sul **cerchio Y** (che fa riferimento appunto all'asse verticale, intorno al quale ruota).



Per **alzare l'inquadratura**, invece, ecco che ci serve il **sistema locale**: la telecamera, infatti, partiva “inclinata” perché inquadrava il personaggio dall'alto, ma una rotazione intorno all'asse X o a quello Z globali non farebbe altro che **peggiorare la situazione**, perché renderebbe ancora più angolata l'inquadratura (mantenendo, tra l'altro, la direzione), mentre noi vogliamo **ruotare intorno all'asse orizzontale PROPRIO della telecamera** (e questo “proprio” deve farvi pensare, appunto, a **LOCALE**), un movimento noto come *Tilting*.

Passiamo quindi alla **modalità LOCALE** e **ruotiamo** un po' intorno all'**asse Y locale**, quindi di nuovo quello verde... lo stesso asse di prima, quello Y, ma prima era globale e puntava verso l'alto nella scena, ora è quello locale, che è **quello orizzontale proprio della telecamera!**

Notate che differenza nel risultato!



Ok, fatto questo possiamo andare nella **schermata Game** e avviare il gioco col tasto **Play...** e qui ci rendiamo conto del limite “tecnico” di questa inquadratura: nella vista dall'alto, la **rotazione** del personaggio era data dal **mouse**, che era libero di ruotare **intorno al personaggio** proprio perché lo guardava dall'alto... se, ora, la telecamera vede quello che vede il personaggio, **lo spostamento in orizzontale del mouse è limitato** dalla schermata, quindi il nostro personaggio non riesce a girarsi... e noi non riusciamo a giocare.



Il tutto si può **sistemare con gli script e i controlli**, comunque lo scopo di questo esempio era mostrare come certe cose, ad esempio la rotazione del personaggio seguendo il mouse, vadano **pianificate con attenzione**, perché magari all'inizio impostiamo tutto per un'inquadratura dall'alto e un certo tipo di controllo ci va bene, poi però vogliamo passare alla visuale in prima persona e ci tocca modificare il sistema di controllo... meglio **conoscere certi problemi in anticipo** e tenerne conto, prima di iniziare un nuovo progetto!

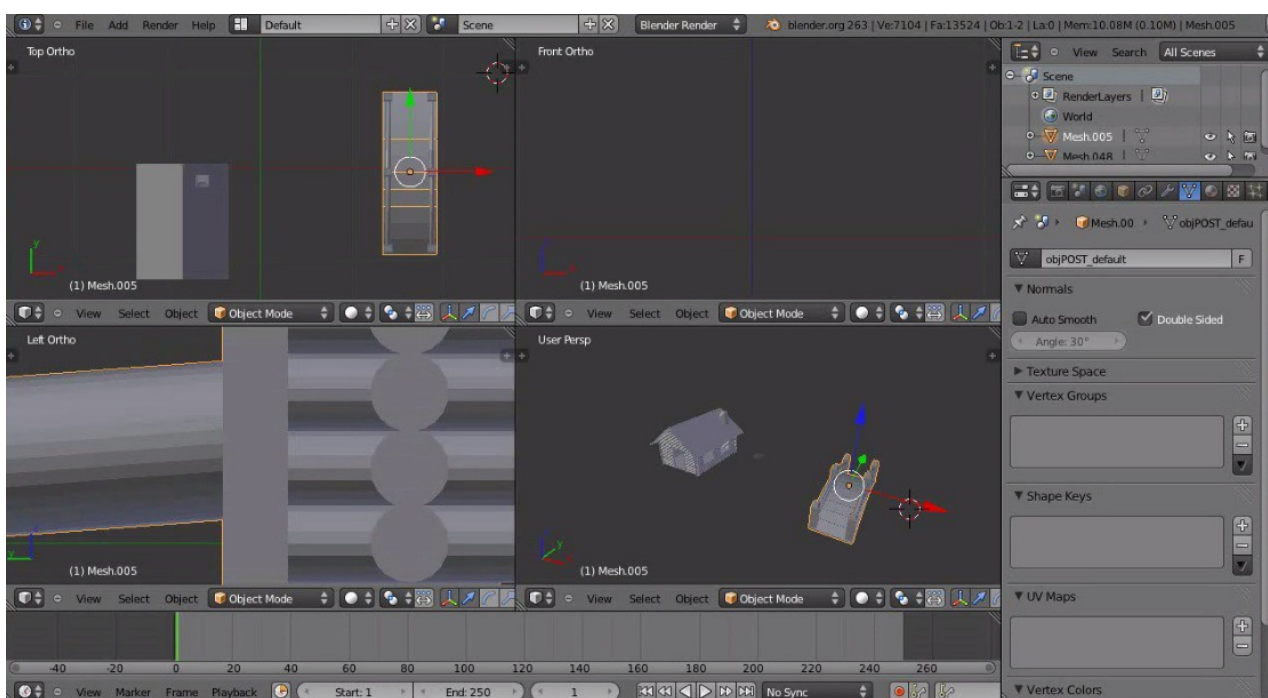
* * *

Le basi di Unity 3D 3.x e 4.x – 3: importare Assets e Packages; Terrain; illuminazione

Eccoci al terzo capitolo sulle basi di **Unity 3D**; in questo tutorial, parleremo **dell'importazione di “assets”**, ovvero – in generale – file utili per il nostro progetto.

Non si tratta solo di **modelli 3D** ma anche di **textures**, **shaders**, **script** o intere **scene** sviluppate con altri programmi; vedremo, inoltre, come **importare “pacchetti” di assets** forniti di serie da **Unity** e **scene** da altri programmi, in particolare **Blender 3D** (ma quello che vedremo qui può essere esteso facilmente a **3D Studio MAX** o **altri software**, i concetti e le procedure non cambiano).

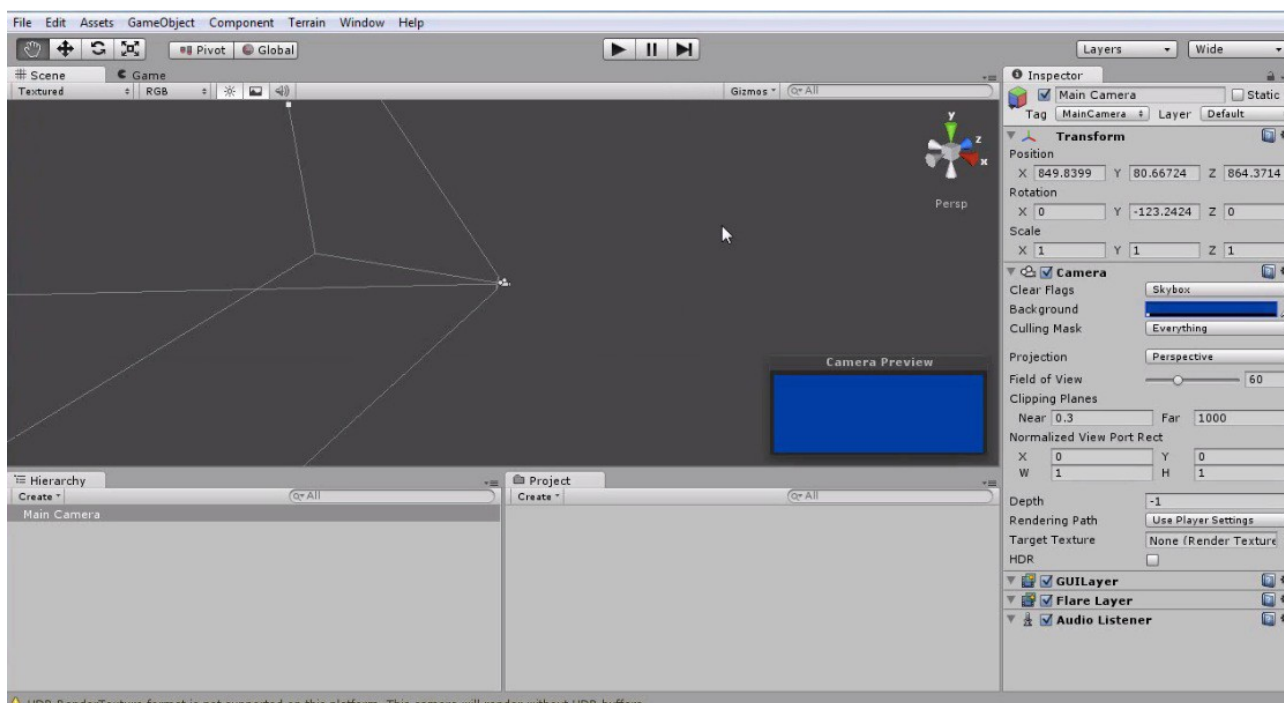
Questa puntata inizia, in realtà, con **un file di scena di Blender 3D**, all'interno del quale abbiamo **due oggetti**: una casetta e un ponticello.



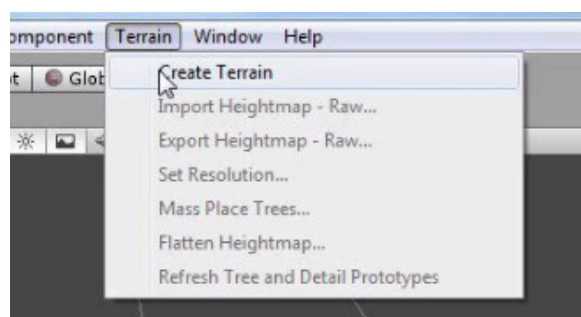
Una piccola nota per chi non conosce o non sa usare **Blender 3D**: non vi preoccupate, qui il concetto non è **“da Blender a Unity”**, il punto è mostrare come sia possibile **importare Assets creati e modificati con programmi esterni in Unity**; inoltre, non è detto che ciascuno DEBBA per forza saper usare tutti i programmi: se ad esempio lavorate in **team**, ci saranno i **modellatori** che nemmeno si occuperanno delle **Textures**, che spetteranno invece ai **disegnatori**, mentre voi metterete i prodotti finali nella scena, quindi non è necessario **saper modellare mesh o creare delle Textures**, vediamo solo come importare gli assets in **Unity**.

Prendiamo nota del nome del file della scena, che si chiama **“ScenaBlender.blend”**.

Chiudiamo **Blender** e passiamo a **Unity**, con un progetto nuovo.

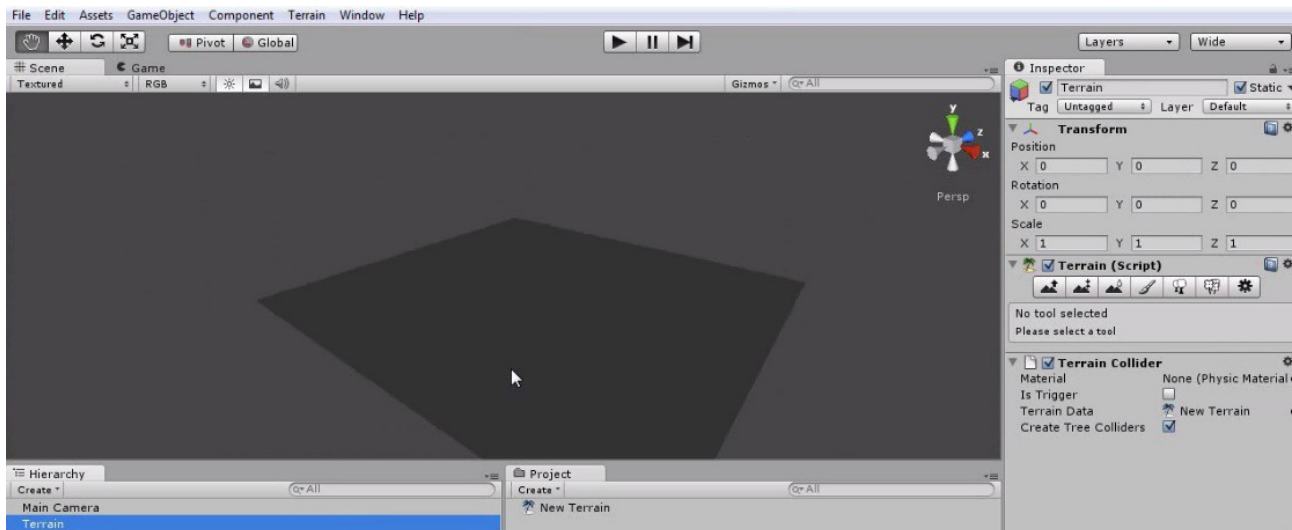


All'inizio di un nuovo progetto, lo **scenario virtuale** è particolarmente triste: **sfondo blu uniforme** (in fase di gioco, nella schermata **Game**), inoltre l'unico oggetto presente, peraltro puramente virtuale, è la **telecamera**.



Approfittiamone per aggiungere un **“pavimento” alla scena**; in particolare, facciamo click su **Terrain – Add Terrain**, nel menù in alto.

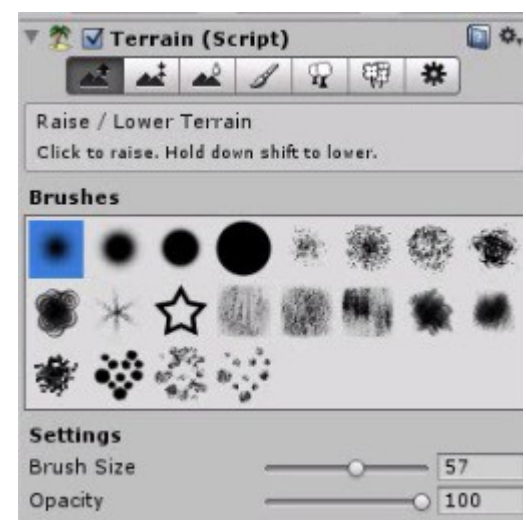
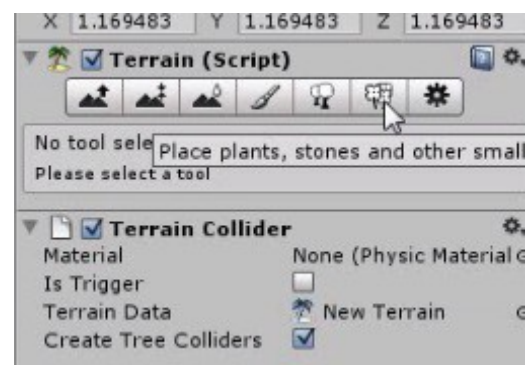
L'oggetto, inserito sia nel **Project** che nella **Gerarchia**, sembra essere un semplicissimo piano, ma non fatevi trarre in inganno: un **Terrain** è un oggetto particolare di **Unity**, messo a disposizione in maniera nativa per accelerare (mediante script che modificano aspetto visivo e geometrie dell'oggetto) il processo di creazione di un **ambiente** davvero ricco di elementi.



In particolare, con tale oggetto selezionato diamo un'occhiata alla finestra **Inspector**: c'è uno **Script**, **“Terrain Script”**, che nella maschera contiene delle icone che sembrano suggerirci... qualcosa alla quale vogliamo credere ma stentiamo a farlo perché ci sembra troppo bella per essere vera: **la possibilità di inserire al volo, con dei click, montagne, colline, valli, textures, alberi ed altri elementi!**

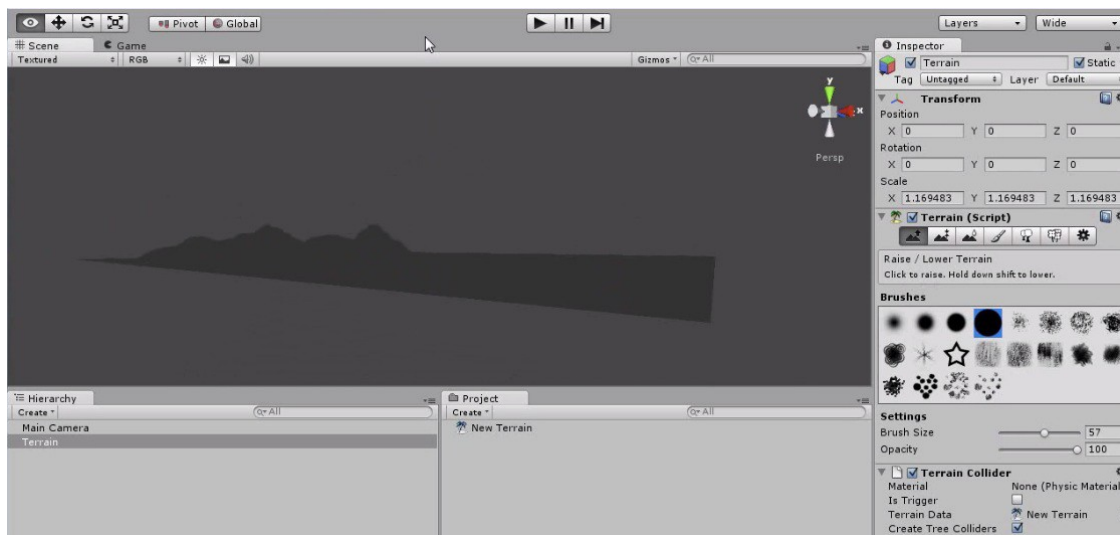
Stentiamo a crederci, ma è la verità: clicchiamo sulla prima icona, **“Raise / Lower”** (alza/abbassa).

La finestra ora ci informa che un click alzerà il terreno, mentre un click ma tenendo premuto SHIFT lo abbasserà.



La forma dei **pennelli (Brushes)** serve a definire la forma delle modifiche, come un pennello in **Photoshop** o **GIMP**, mentre in **Settings – Brush size** possiamo definire la **grandezza del pennello** (e, quindi, la dimensione delle modifiche, a ogni click) e in **Opacity**, in un certo senso, **l'intensità**.

Portiamo **Opacity** a 100 e iniziamo a creare dei **rilievi** cambiando a piacere forma e dimensioni del

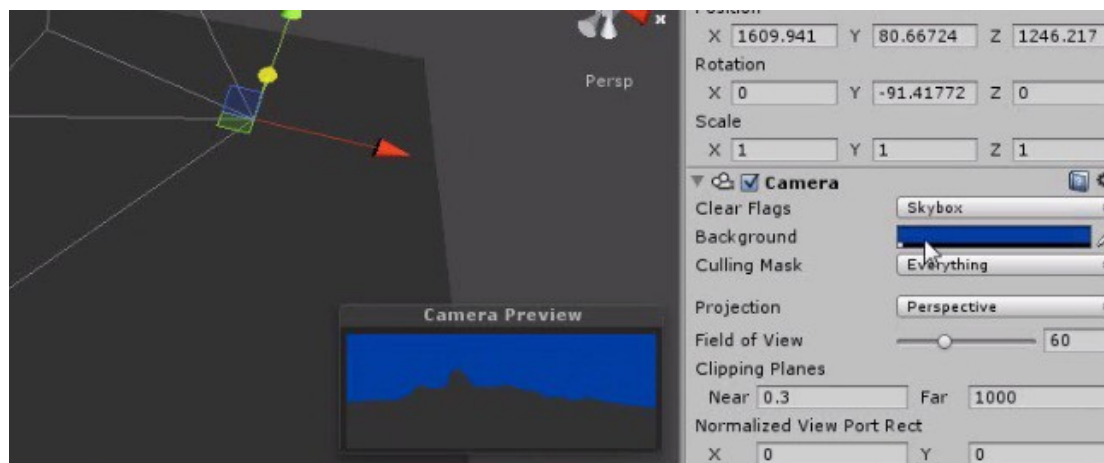


pennello, per prendere confidenza con lo strumento.

Sto andando a caso, quindi non stupitevi se il risultato sarà particolarmente... brutto!

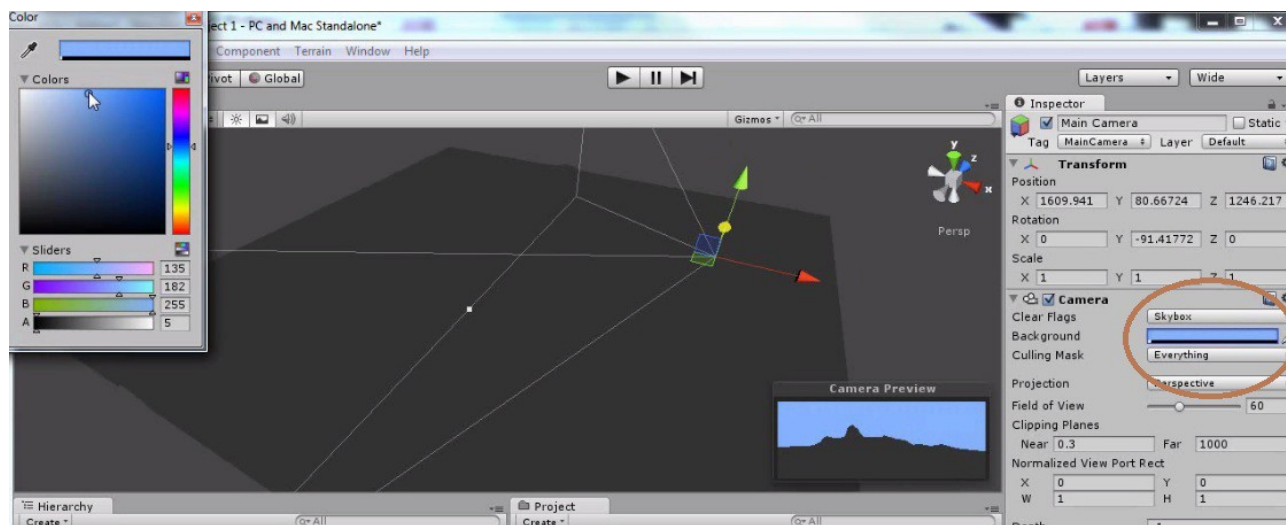
Ok, il **Terrain** quindi è un ottimo punto di partenza per definire un **ambiente virtuale** in poco tempo, tuttavia in questo capitolo ci fermiamo a **rilievi e valli** (e, tra poco, **Textures**).

Molto probabilmente l'oggetto **Main Camera** sarà ora in un punto poco interessante, quindi **spostiamolo** per definire meglio l'inquadratura (visibile nell'anteprima in basso a destra nella finestra **“Camera Preview”** della scheda Scene).



Notiamo, comunque, che il **cielo** ha sempre quel colore che non ci piace, mentre **lo scenario è grigio scuro**.

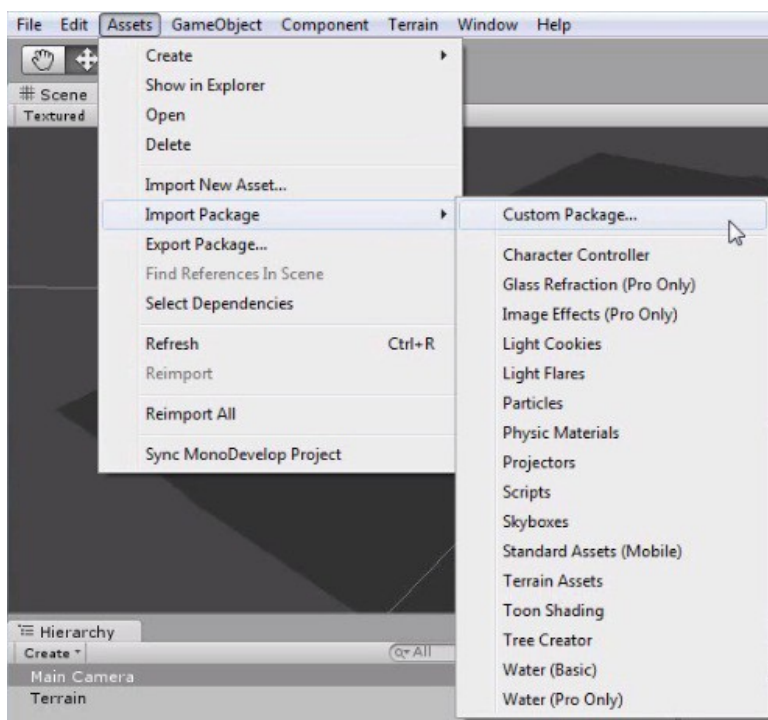
Per risolvere il primo problema, con la **telecamera** selezionata, nell'Inspector cambiamo – a piacere – il colore nel campo **Background**... beh, senza **Textures** il cielo resta brutto, ma comunque meglio di prima, inoltre ora sappiamo dove si trova **il colore dello sfondo**: nella scheda della **Telecamera virtuale**.



Per quanto riguarda la **mesh**, importiamo il primo **Asset** esterno nella nostra scena... anzi, per la verità importiamo **un intero pacchetto di assets**, come una cartella di file ma contenente, nel nostro caso, elementi di vario tipo.

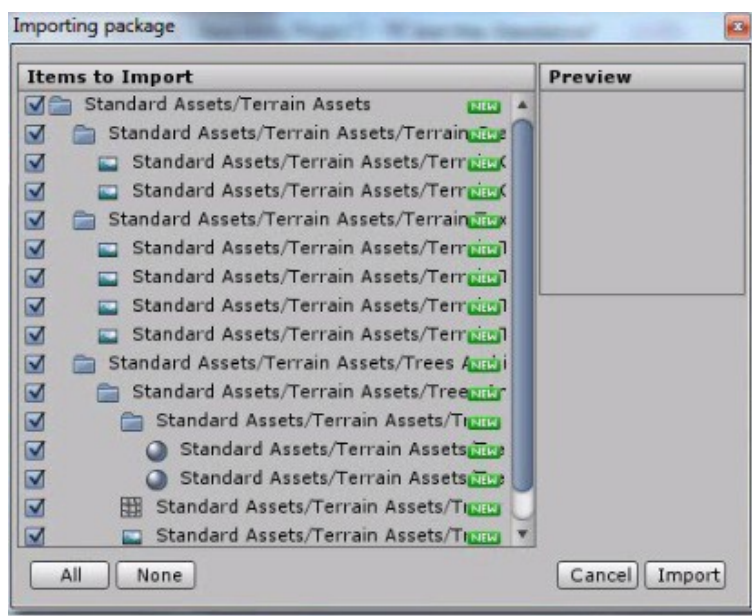
Unity è un programma meraviglioso che fornisce, di serie anche nella versione gratuita, un'incredibile quantità di oggetti.

Clicchiamo sulla voce di **menù Assets**, in alto, scegliendo “**Import Package**”.



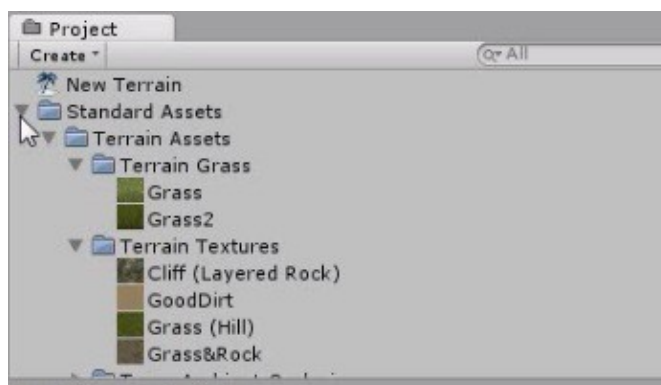
La prima voce, in questo sottomenù, è **Custom Package**, cioè un pacchetto personalizzato (infatti è possibile creare e salvare pacchetti personalizzati da rendere disponibili per altri utenti e altri progetti), comunque tutte le altre voci sono **pacchetti di elementi e strumenti** forniti dal programma (alcuni di questi sono solo per la **versione PRO**, come indicano le voci **“Pro Only”**, ad esempio per **Glass Refraction** o **Image Effects**).

Scegliamo il pacchetto **Terrain Assets** e confermiamo l'importazione di tutti gli elementi nelle schermate che appariranno.



Al termine dell'import, in **Project** vedremo l'icona di una cartella e la voce **“Standard Assets”**, dentro la quale (con un doppio click) troveremo **“Terrain Assets”** e, dentro, altri elementi.

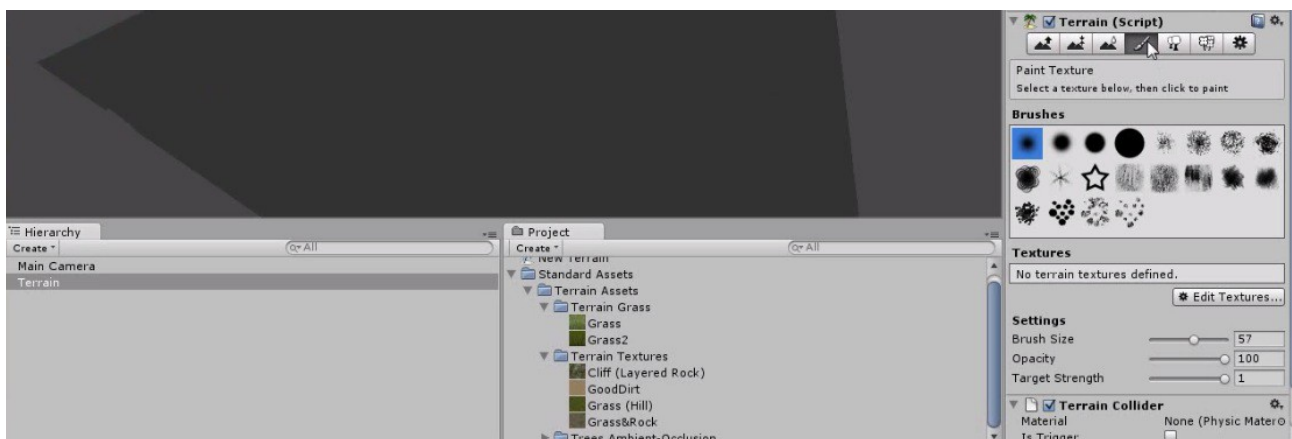
Scegliamo, in particolare, **“Terrain Textures”**: il pacchetto ci mette a disposizione **4 Textures** per il nostro **terreno**.



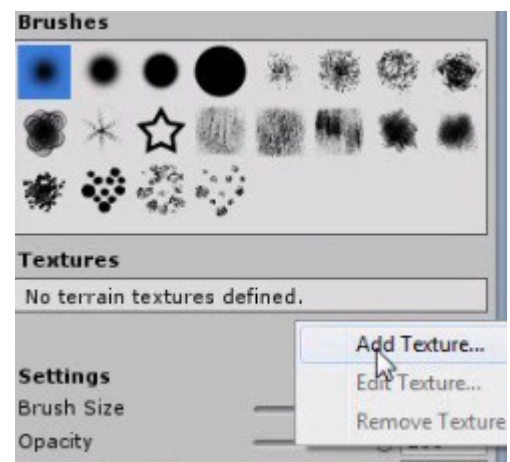
A questo punto la domanda è: come facciamo a dare, ad esempio, **“Grass (Hill)”** al nostro ambiente?

Selezioniamo l'oggetto **Terrain**, ma attenzione a selezionarlo questa volta in **Gerarchia**, non in **Project**, visto che vogliamo modificare l'elemento presente nella **scena corrente**.

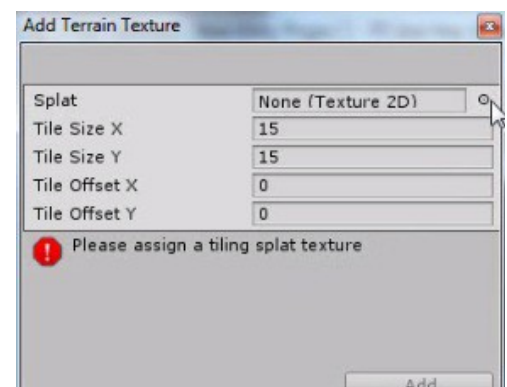
Nella sezione **Terrain (Script)**, selezioniamo l'icona col pennello, **“Paint the Terrain Texture”**, quindi clicchiamo sul pulsante **“Edit Textures”** in basso.



Apparirà una nuova finestra dove potremo selezionare alcune **Textures** tra quelle presenti nel progetto.

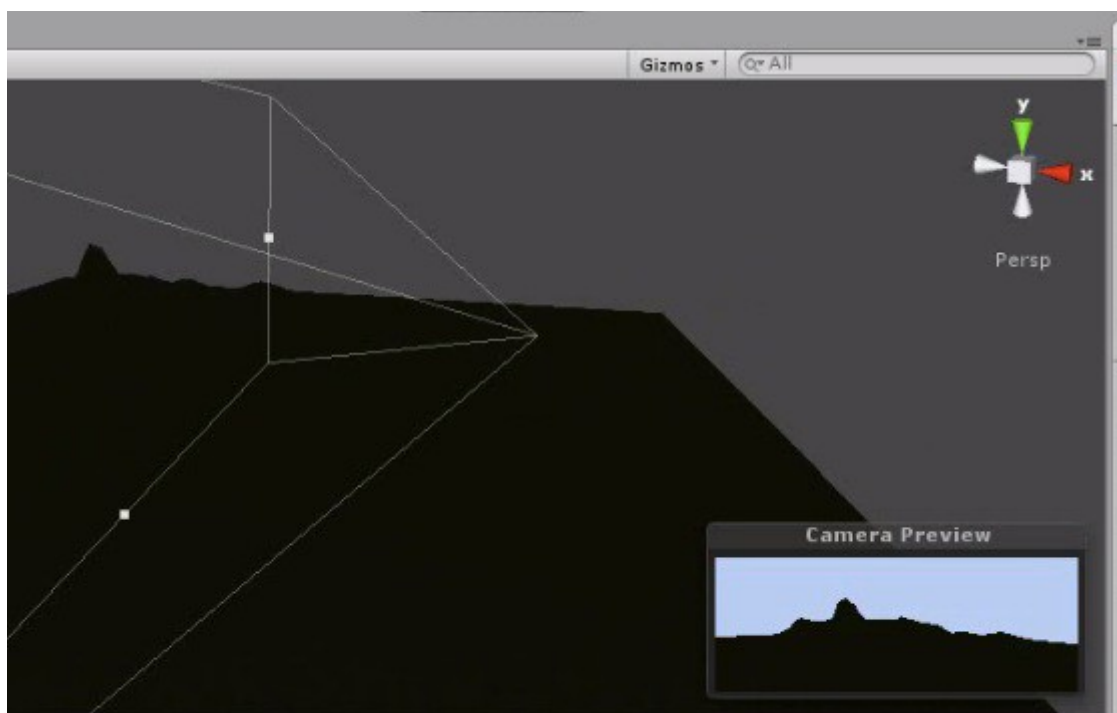
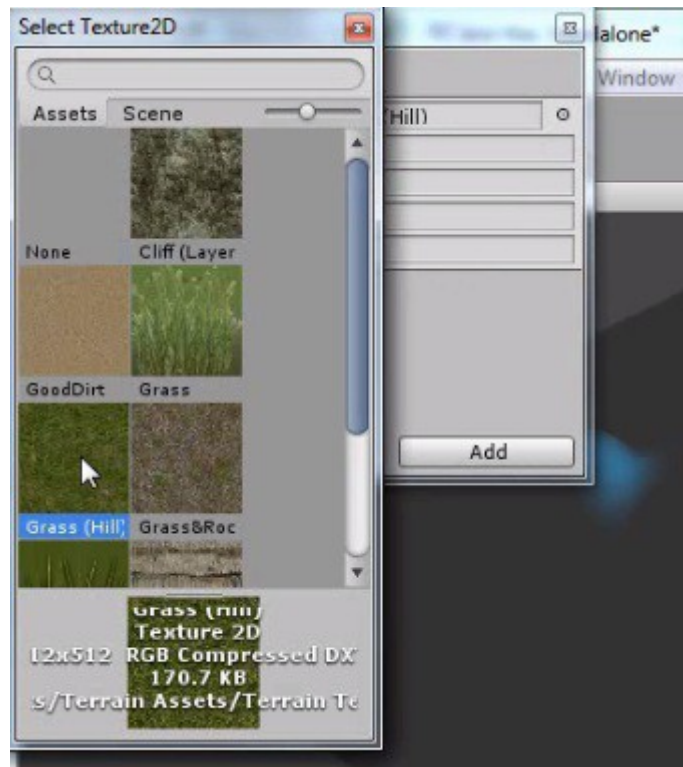


All'avvio di un **nuovo progetto**, questa finestra in genere è vuota perché **non ci sono Assets di tipo Textures**, ma ora abbiamo le **Textures importate** nel progetto mediante il **Package Terrain Assets!**



Scegliamo quindi **Grass (Hill)**, lasciando le altre impostazioni come di default (potranno essere comunque modificate in seguito) e confermiamo.

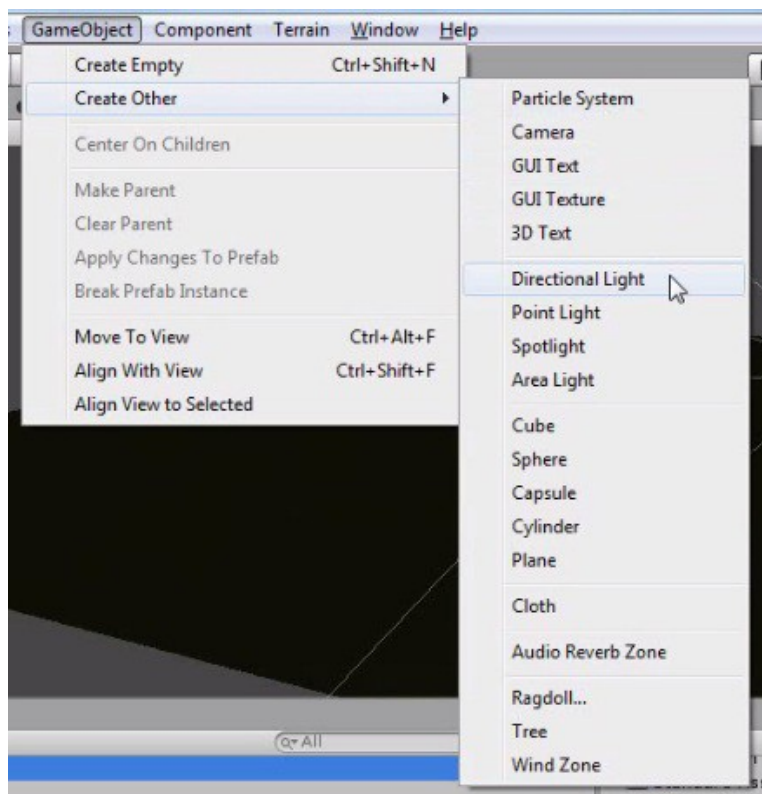
La **Texture** verrà applicata, replicandola, **su tutto il terreno**, tuttavia guardando l'anteprima della vista **Telecamera**, o comunque nella scheda **Game**, notiamo che **il Terrain è nero!** Non è cambiato molto, rispetto al grigio di prima!



Il motivo sta nell'illuminazione della scena: attualmente, non ci sono fonti di luce nella scena, né **Textures HDR** (particolari textures “illuminanti”, per così dire) come **sfondo dell'universo virtuale**.

Aggiungere una fonte di luce è semplicissimo: dal **menù Game Object**, scegliamo **Create Other** e, da qui, **Directional Light**.

Le **fonti di luce** presenti sono 4 (notate i separatori nel sottomenù **Create Other**) e i loro utilizzi e le loro caratteristiche sono facilmente intuibili: **Point** è una sorgente **puntiforme omnidirezionale**, con ombre e attenuazione, come una **lampadina a bulbo**; **Spot** è il classico **faretto**, quindi con direzione, cono d'apertura, ombre e attenuazione; **Area** è una **superficie emettitrice**, come un pannello luminoso o un monitor; **Directional**

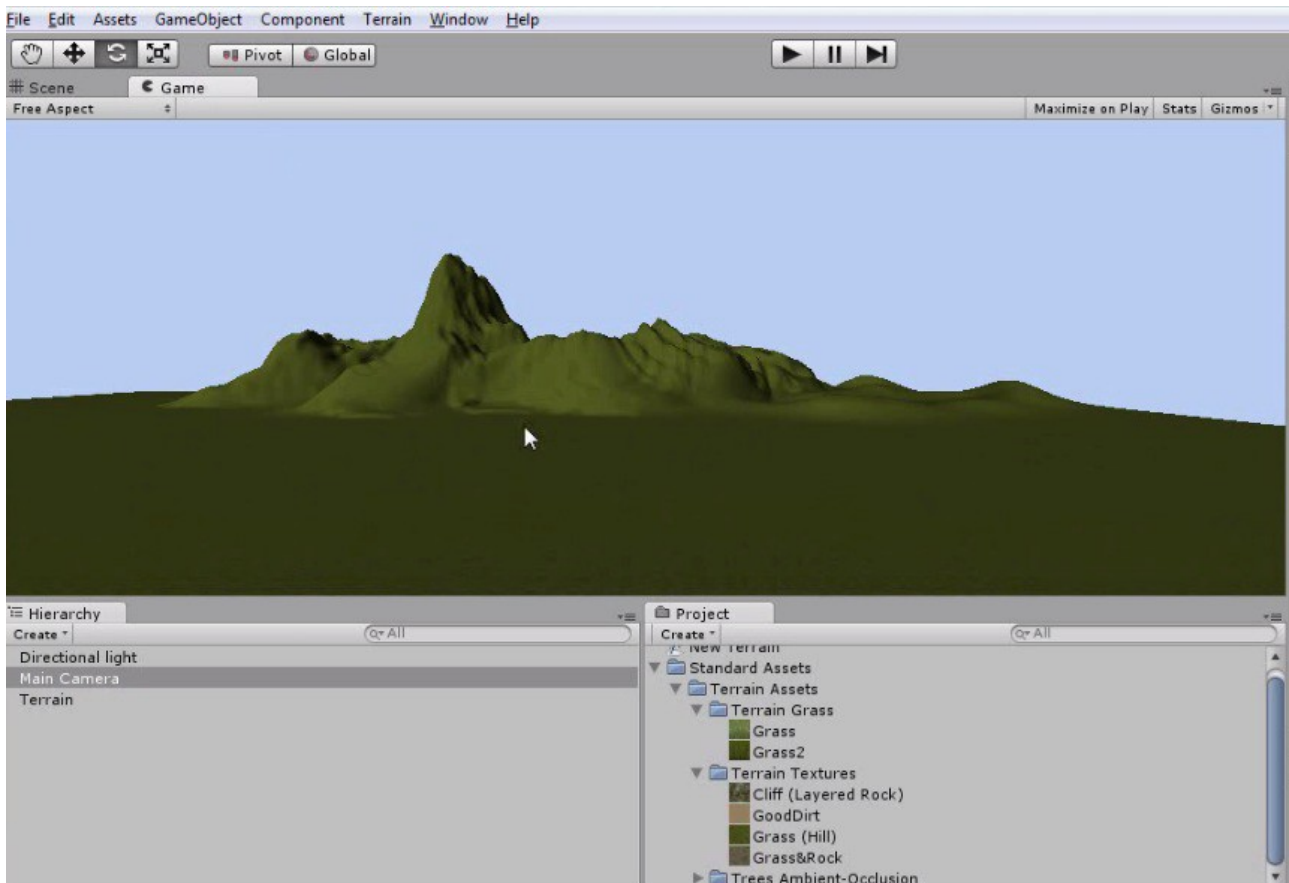


Light, quella che abbiamo scelto, è una luce che ha origine da distanza infinita e con superficie emettitrice illimitata, senza attenuazione, quindi qui – ed è l'unico caso tra le 4 luci – **non ha importanza la POSIZIONE** dell'icona della luce nella scena, **ma solo il suo ORIENTAMENTO**, perché sarà l'orientamento dei raggi luminosi.

Per tutte queste caratteristiche, **Directional Light** è ottima per **l'illuminazione ambientale di uno scenario aperto**, come il nostro.

Fate un po' di pratica con tale oggetto nella scena spostandolo e, soprattutto, ruotandolo, utilizzando sia le coordinate **Global** che **Local**... ovviamente, potete fare anche delle prove con **altre fonti di luce**, inserendo oggetti **Point Light**, **Spot Light** o **Area** e spostandoli o ruotandoli per valutarne caratteristiche ed effetti.

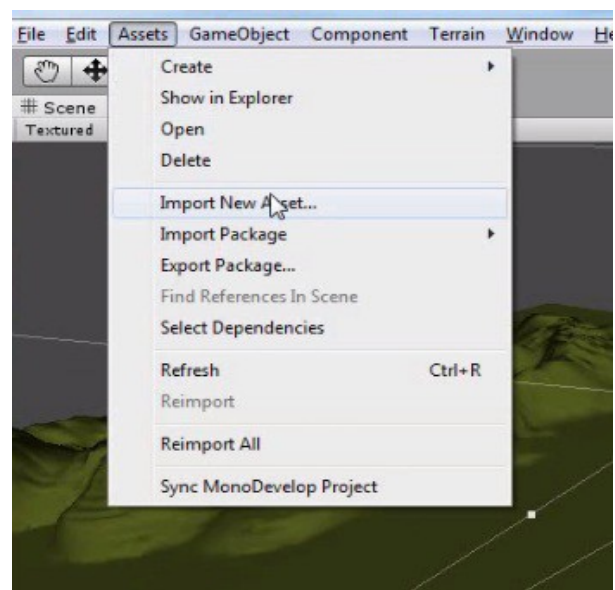
Come si può notare nella **Camera Preview** o nella finestra **Game**, lo **scenario è ora illuminato** (immagine nella prossima pagina).



* * *

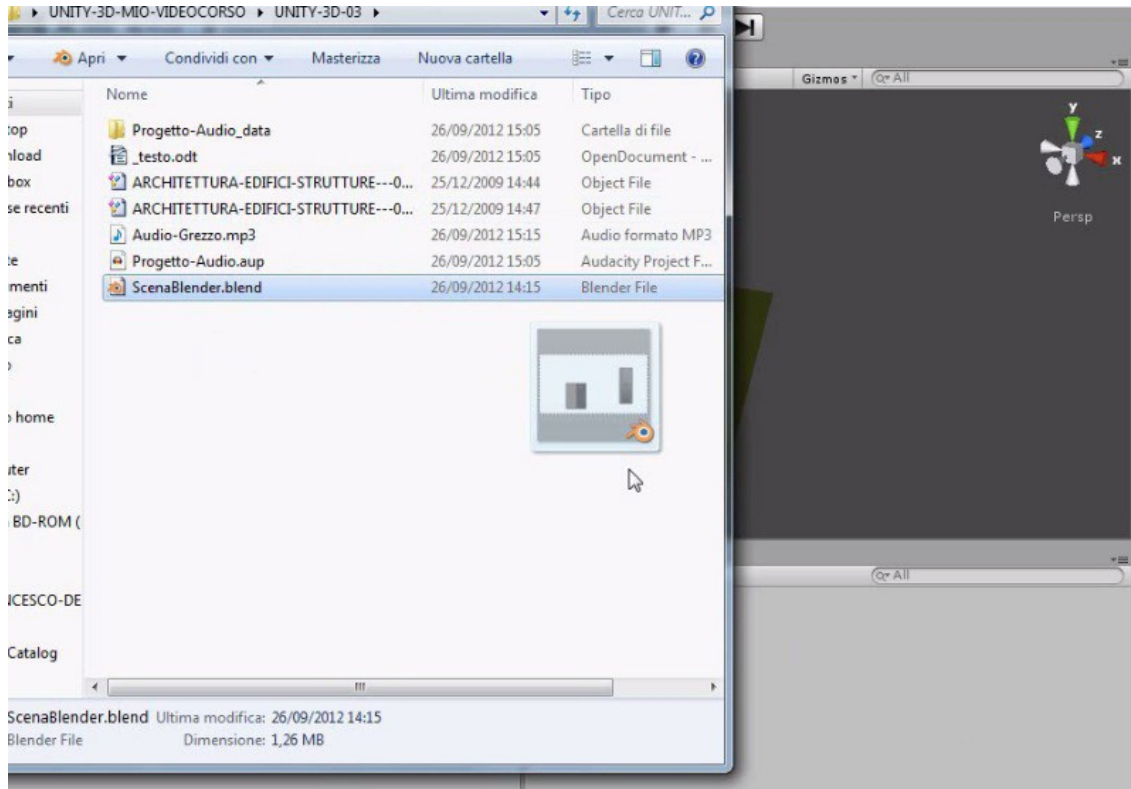
Trattiamo adesso il secondo argomento di questo tutorial: **l'inserimento di assets in arrivo da altri software.**

In teoria, in base a quanto visto finora, dovrebbe essere sufficiente selezionare **Assets – Import new asset** e scegliere (riprendendo il nome del file .blend citato all'inizio del capitolo) **“ScenaBlender.blend”**... beh, sì: è proprio quello che bisogna fare!

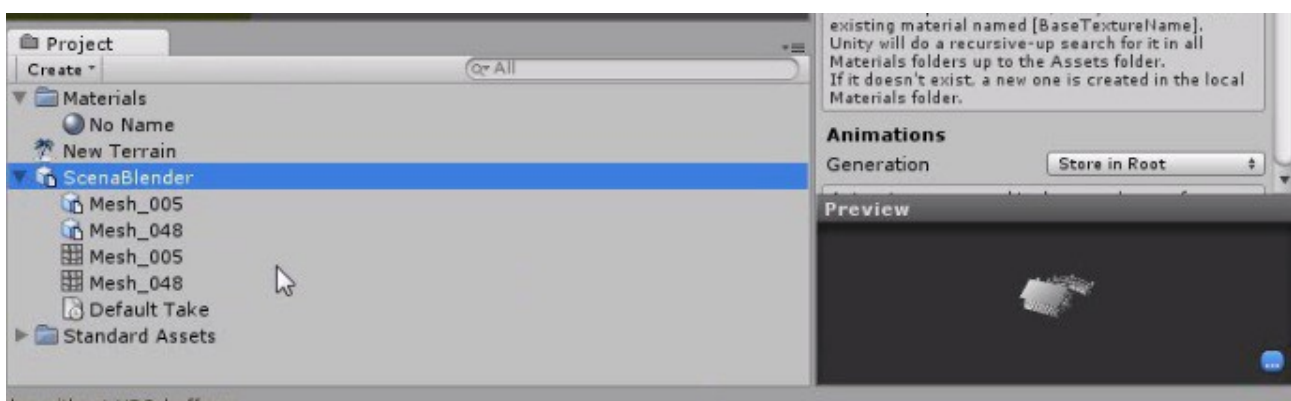


Raccolta di tutorial **“UNITY 3D tutorials – Volume 1”** dal sito *francescomilanese.com* --- © 2015

Oppure... possiamo procedere in questo modo (immagine qui sotto): con la finestra **“Esplora risorse”** con la cartella con la **scena .blend** che voglio importare aperta, **trascino il file in Project**, nella finestra di Unity: ecco **creato un Prefab della scena di Blender!**

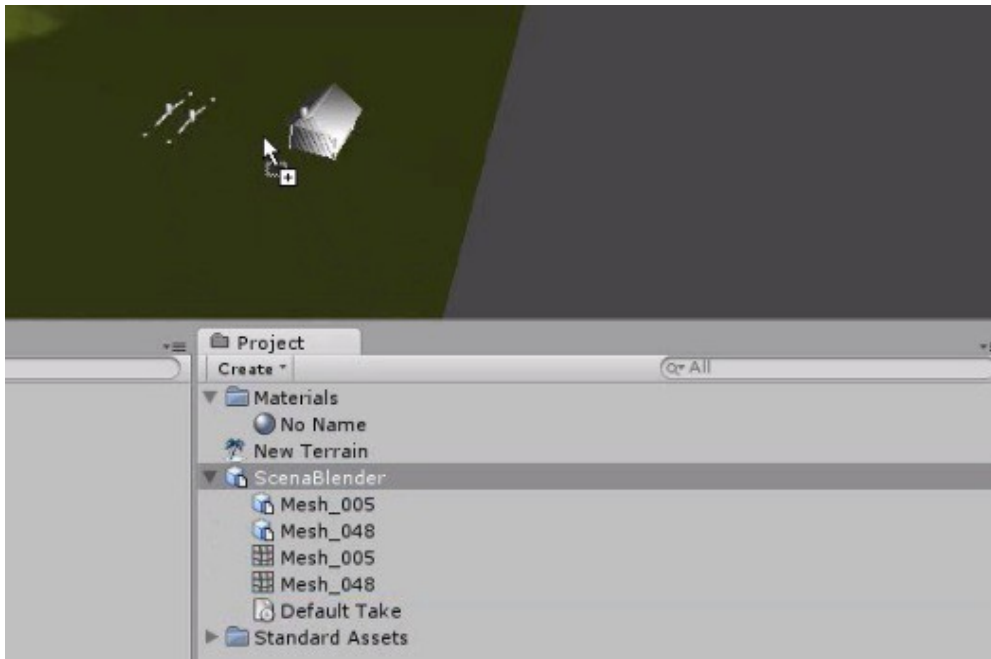


Facendo click sul **Prefab “ScenaBlender”**, notiamo la presenza di **due “Oggetti”** (combinazione di **geometria mesh e materiali**, eventualmente con **Textures**) e delle sole **mesh**, che sono le sole informazioni sulla **geometria del modello**, senza informazioni sull'aspetto, i materiali, eccetera.

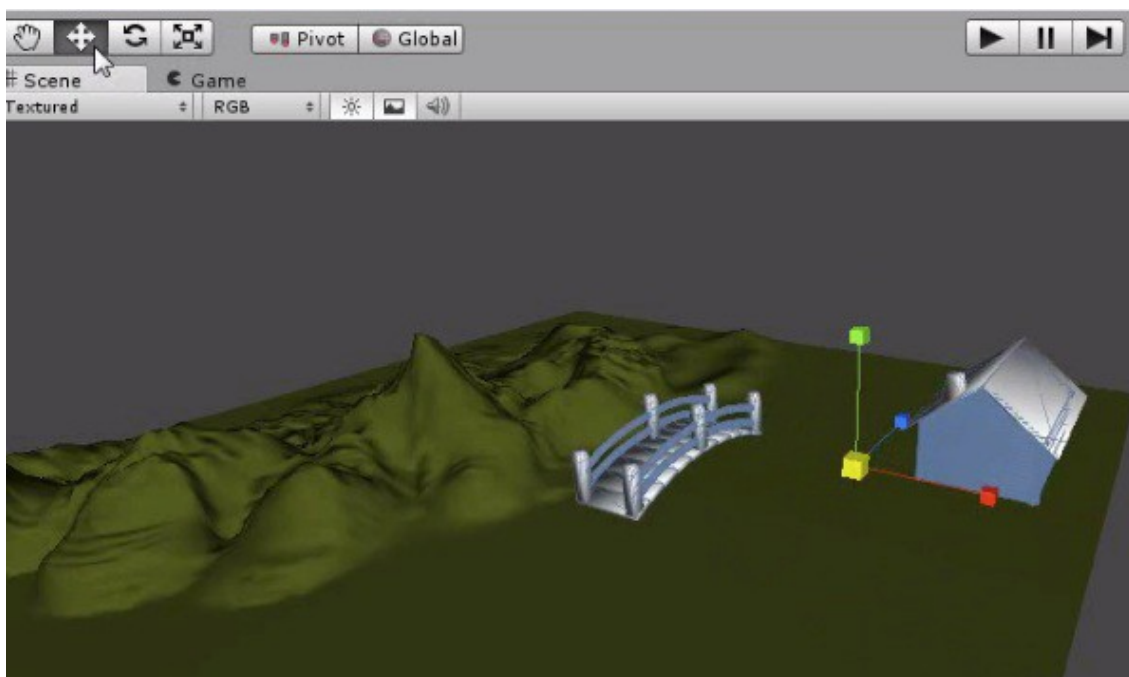


Gli **Oggetti generici** hanno come icona il **cubo con il foglio bianco**, mentre le **mesh** hanno una specie di **griglia 3D**, a suggerire appunto che si tratta di **informazioni sulla geometria** (vertici, spigoli e facce).

Trasciniamo *“ScenaBlender”* in un punto nella finestra **Scene**: ecco importati i **due modelli** presenti nel file *“ScenaBlender.blend”* all'interno del nostro **scenario Unity**!



Qui, possiamo **traslare, ruotare e scalare** a piacere questi oggetti.

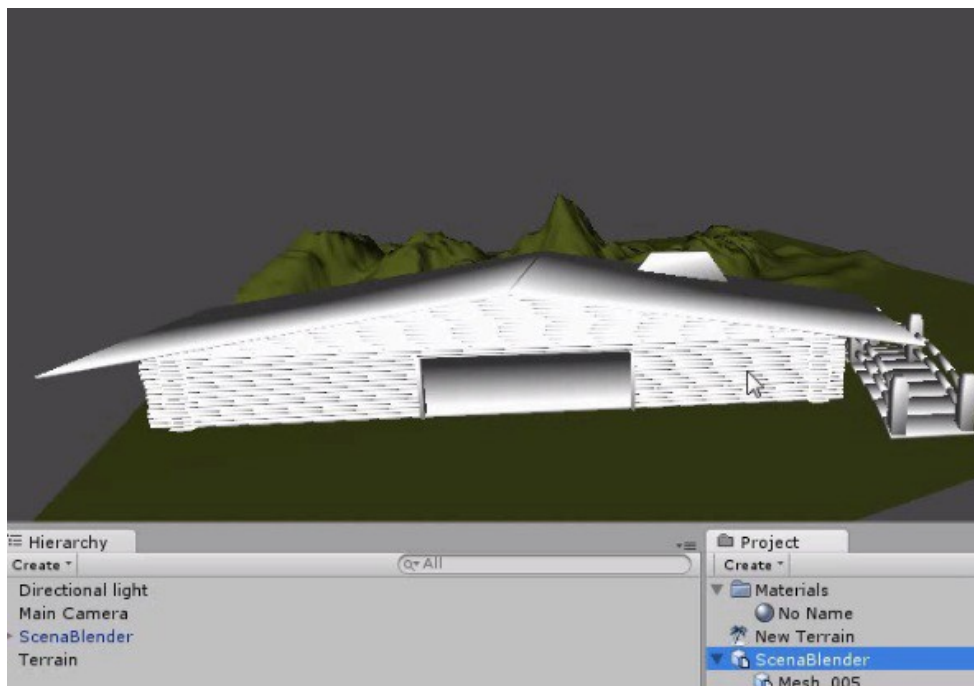
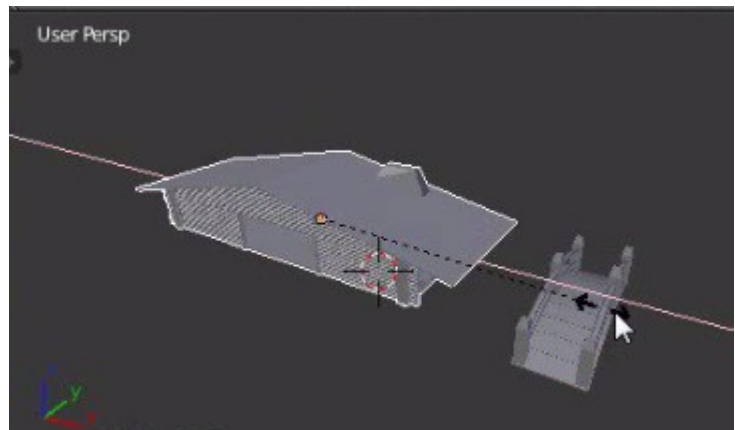


I file *.blend* (ma il discorso vale, ad esempio, anche per i file delle scene 3ds di 3D Studio MAX o di altri programmi... Unity ha una notevole **compatibilità** verso molti software, di vario tipo), possono essere usati come **“librerie” di modelli 3D**, ad esempio potreste avere dei file *.blend* chiamati “divani”, altri chiamati “ponti”, “aerei”, eccetera, da utilizzare per importare i vari elementi nelle scene dei progetti.

Adesso, faccio **doppio click sul Prefab “ScenaBlender” in Project**, ed ecco che si apre **Blender**: Unity riconosce automaticamente il programma associato ad un dato **Asset**, quindi facendo doppio click su un **file PSD** (perché, sì, potete trascinare qui anche i **file di progetto di Photoshop**, con **tanto di livelli**), si aprirà **Photoshop**, e così via... ma la vera **magia** è un'altra.

Modifico il **modello originale in Blender**, salvo la scena, torno alla finestra di **Unity** e noto che **Unity** sta lavorando...

... e, dopo un po', **la scena si aggiorna**: i modelli, inseriti poco fa come **Prefabs**, hanno ora la **nuova geometria che ho appena modificato in Blender**!



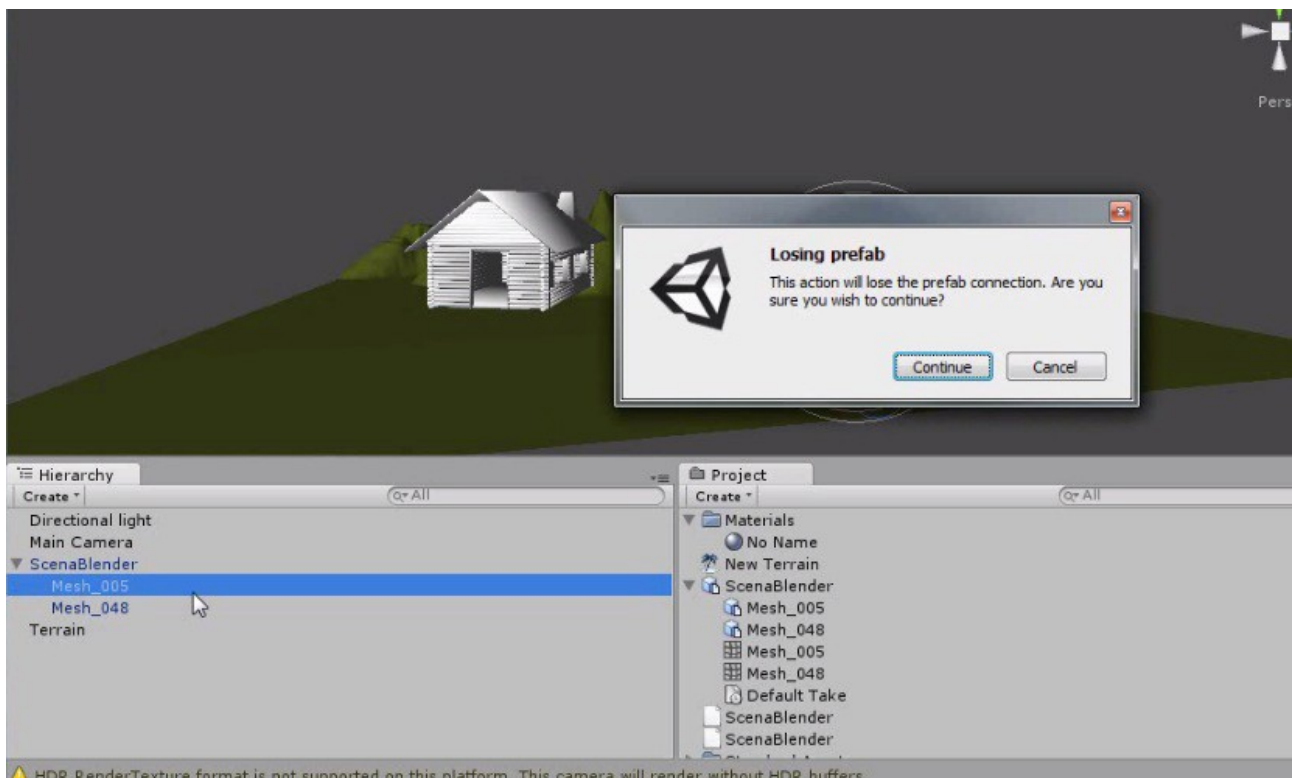
Blender è ancora aperto, è ancora “caldo”, ma è **bastato salvare** perché **Unity**, al volo, si rendesse conto delle **modifiche apportate alla scena** e le applicasse al progetto corrente!

Neanche a dirlo, **se lavorate in team** (o anche da soli, meglio a questo punto **con un doppio monitor**), tutto ciò può accelerare tantissimo il lavoro!

Tutto bello, se non meraviglioso, fino a questo punto, però... **non è tutto oro quel che luccica...**

C'è un **problema**, in particolare: l'inserimento di un **Object** del file **.blend** nella scena crea **COMUNQUE** un oggetto scena nella **Gerarchia**, importando quindi nella scena **TUTTI** gli **elementi presenti nel file .blend**, non solo il modello selezionato.

Il problema può essere risolto selezionando **le voci che non vogliamo** nella finestra **Gerarchia** (che fa riferimento alla scena corrente) e **cancellandole** (spostare il mouse nella finestra **Scene** e premere **CANC**); **Unity** ci avviserà che, così facendo, **perderemo il collegamento col Prefab**, perché **la Scena di Blender è un Prefab** e se **modifichiamo un'istanza nella Gerarchia**, questa **non è più un'istanza** ma un nuovo elemento a parte.

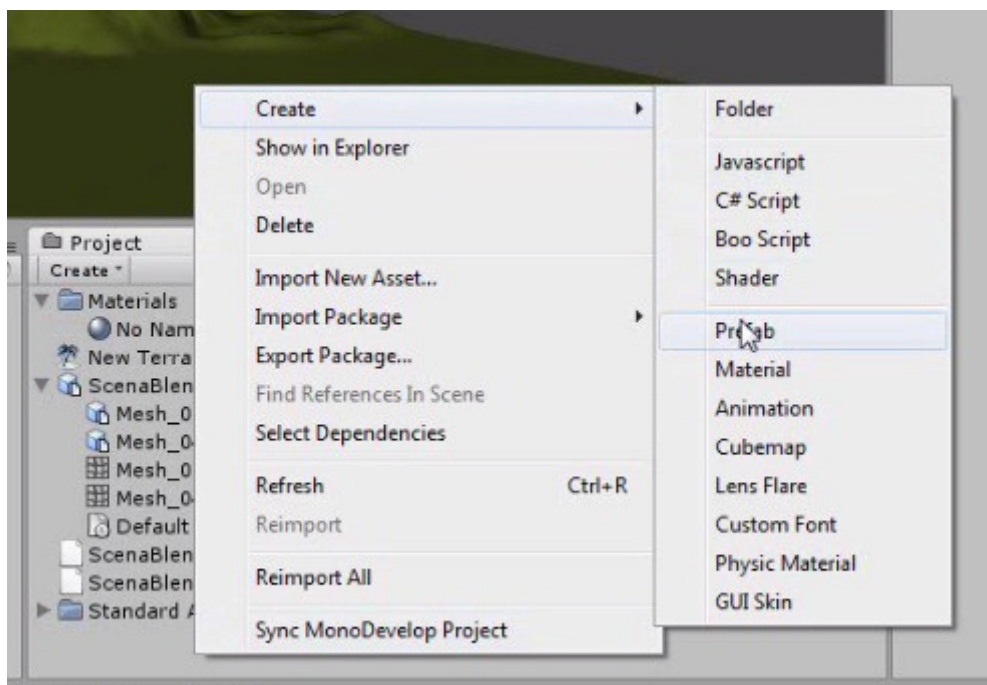


Se necessario, rivedete il capitolo precedente per pensare alle **implicazioni** che questa operazione può avere, specialmente in **progetti ricchi di elementi simili**; inoltre, perdendo il collegamento col **Prefab** perderemo anche la possibilità di **modificare al volo l'oggetto** (sia esso un modello 3D che un'immagine) **nel programma di grafica 2D o 3D** e vedere le **modifiche in tempo reale nella Scena** (le modifiche verranno applicate comunque al **Prefab** nel **Project**).

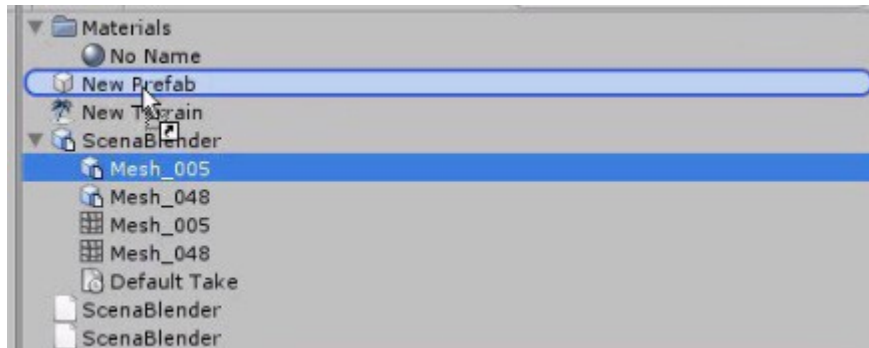
Ecco quindi un nodo importante: **un file di libreria di modelli 3D** contiene molti modelli ma per inserirne uno solo nella scena dobbiamo creare oggetti distinti, perdendo il discorso del **Prefab** e delle istanze.

Se non volete perdere il **collegamento col Prefab**, ad esempio perché volete inserire nella scena tanti oggetti dello stesso tipo provenienti da un **file di libreria**, considerate l'idea di fare **un file .blend** (o **.3ds** o altro) contenente SOLO l'oggetto che vi interessa, così potrete **farne un Prefab** e inserire tutte le istanze che volete nella scena... *OPPURE*, potete fare come sto per mostrarvi.

Create un nuovo Prefab vuoto con click destro in **Project** e **Create – Prefab**.



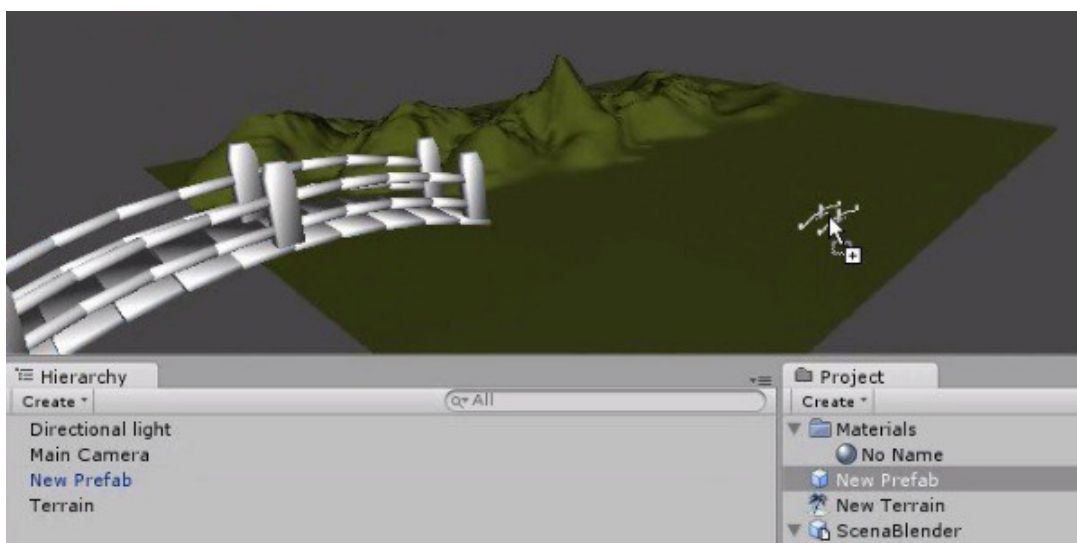
Trascinate l'oggetto presente nel file di libreria che volete replicare all'interno del nuovo Prefab, come visibile ad esempio nell'immagine seguente con **Mesh_005** da **ScenaBlender**.



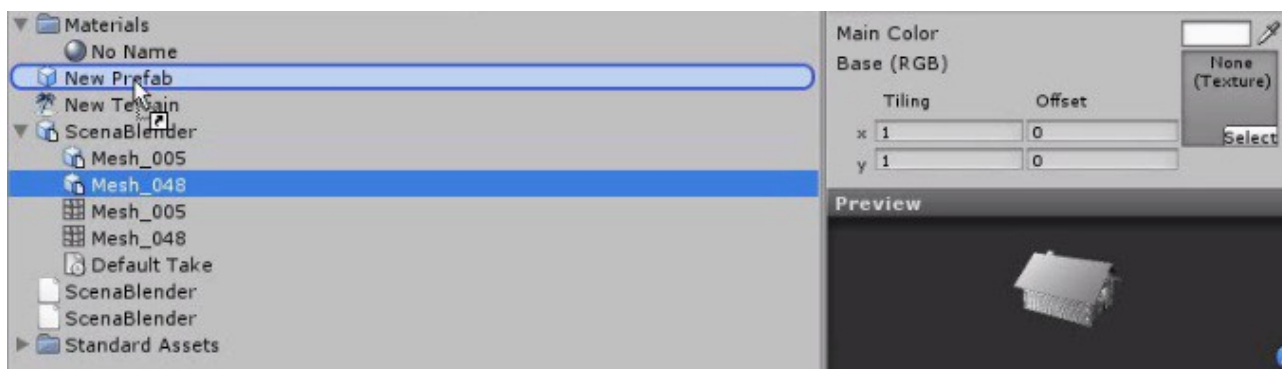
A questo punto, inserite **tutte le copie che volete del Prefab nella Gerarchia** o nella scena corrente: avete delle **istanze**, non dell'oggetto preso dal file di libreria, bensì **dal Prefab appena creato**.

Questo significa che una **modifica** fatta nel file di **libreria originale** (nel mio caso, nel file **ScenaBlender.blend**) NON si ripeterà nelle istanze appena definite, perché sono **istanze di un Prefab diverso...**

... **tuttavia** (come vedete, i **trucchetti** non finiscono mai), nessuno ci vieta di selezionare di nuovo l'oggetto presente nel **Prefab ScenaBlender** e **ri-trascinarlo nel Prefab** creato un attimo fa, di fatto **aggiornando il Prefab e tutte le sue istanze nella scena!**



Anzi, vi faccio notare che potete **sovrascrivere quel Prefab** con un modello **completamente diverso**: nell'immagine seguente, ad esempio, sto **trascinando Mesh_048** dal **Prefab ScenaBlender** sul **Prefab “New Prefab”** creato poco fa; confermo l'operazione ed ecco che **ho sostituito tutte le istanze in un colpo solo!**



* * *

Le basi di Unity 3D 3.x e 4.x – 4: un semplice script Javascript in Unity: countdown

Eccoci al **quarto capitolo** sulle basi del **motore di gioco Unity 3D**; in questo tutorial metteremo da parte, anche se parzialmente, la scena 3D, per parlare degli **script in Unity**: cosa sono, a cosa servono e come vengono creati e utilizzati.

Attenzione, prima di procedere, una piccola premessa: se siete dei **programmatore** o avete già programmato qualcosina in **Unity**, saltate questo capitolo. Qui farò infatti **un'introduzione agli script in Unity**, parlando di elementi come **istruzioni, variabili o funzioni**, tutte cose fondamentali ma ben note ai programmatori e che finirebbero per annoiare questi ultimi, quindi se è il vostro caso potete saltare tranquillamente questo capitolo... altrimenti, iniziamo!

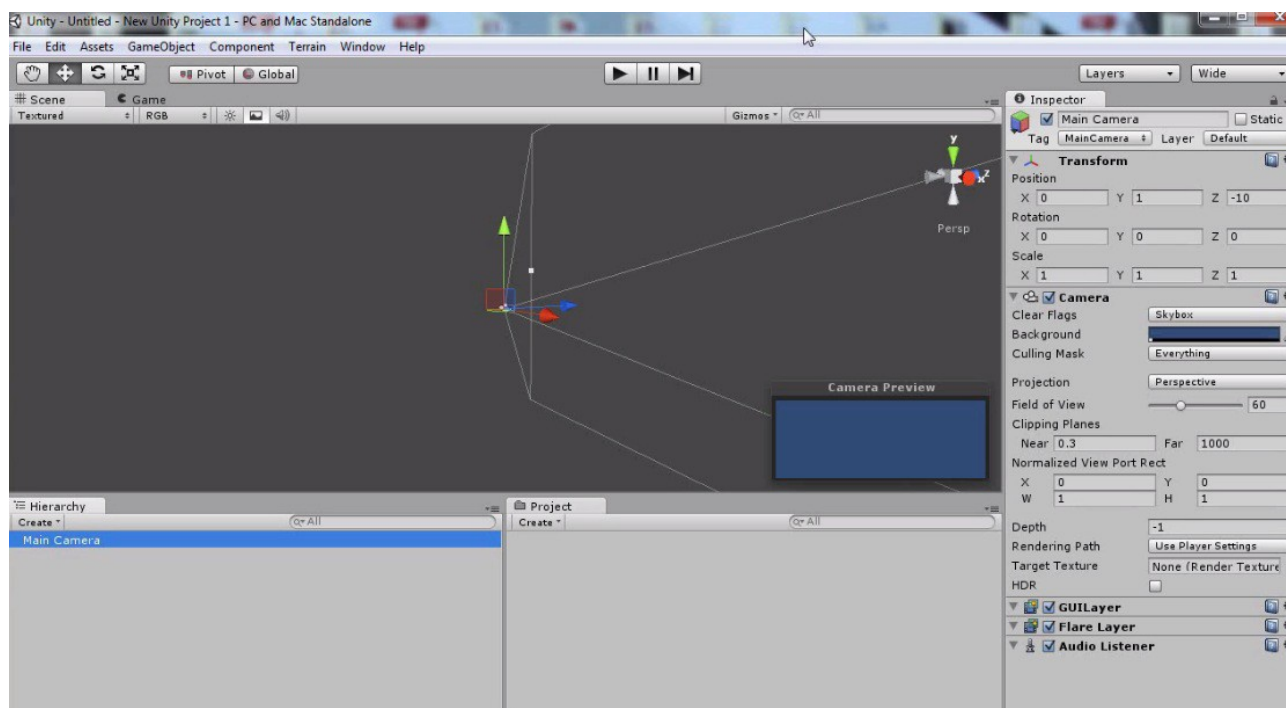
Uno script è una serie di istruzioni da fornire a uno o più oggetti di una scena.

Gli **script** sono semplici **programmi** che, più che altro, servono ad **interagire con altri programmi** o parti di essi; gli “altri programmi”, che poi sono i programmi applicativi in senso stretto, generalmente sono scritti in un altro linguaggio di programmazione, tipicamente compilato o ibrido, mentre gli **script** vengono **interpretati** dalla macchina e a volte vengono creati proprio dall'utente finale, che se ne serve per dialogare, per così dire, con i software veri e propri.

Esempi di script sono i comandi batch per il sistema operativo o per certi programmi (come Photoshop) o le macro, mentre in **Unity** gli script vengono utilizzati, come detto, per far eseguire azioni o reazioni agli oggetti (quindi **Unity** è un programma vero e proprio, un software in senso stretto, che è stato scritto e compilato dai programmatori e che va installato ed eseguito, mentre quelli che scriveremo noi sono porzioni di codice **Javascript**, che **Unity** interpreterà per gestire gli elementi del gioco).

Gli **script** servono per le funzioni più disparate: dal mostrare un conto alla rovescia sul monitor (che è quello che faremo in questo capitolo) al definire l'intelligenza artificiale dei bots; in ogni caso, si tratta di **“cose da far fare”**, seguendo un ordine e **istruzioni** ben precise, agli elementi del gioco.

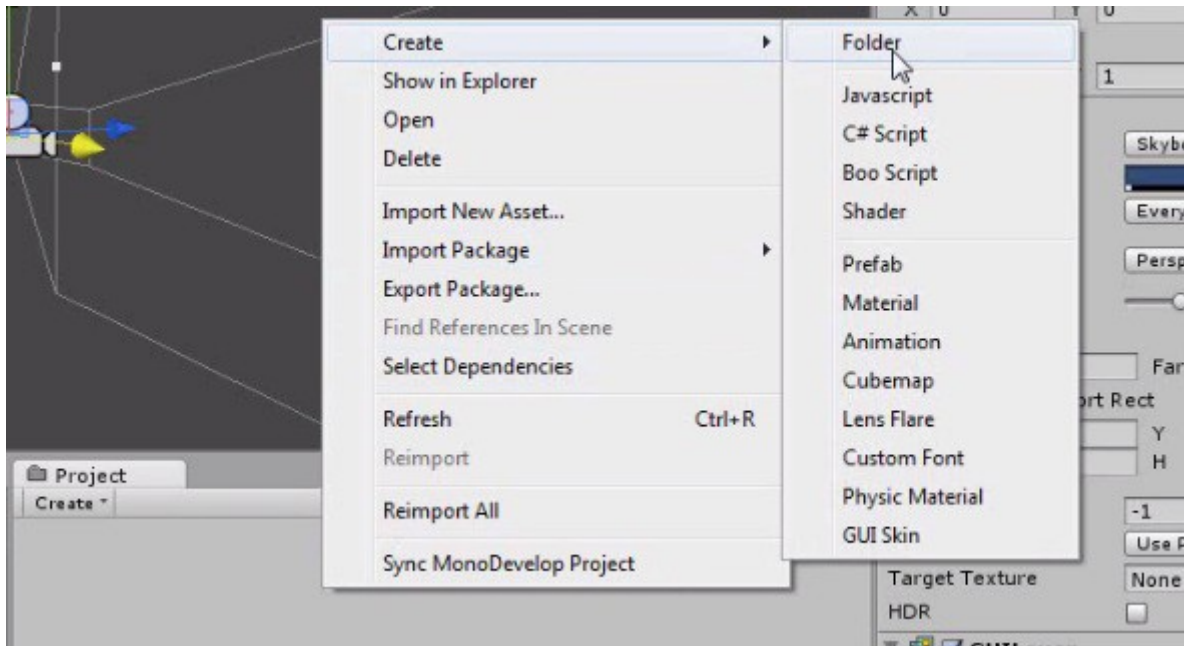
Fatte queste premesse, vediamo **come creare il primo, semplice script in Unity**, partendo da un progetto completamente vuoto (e che comprende, quindi, solo la telecamera **Main Camera**, nella scena).



In questo tutorial useremo il linguaggio **JavaScript** per realizzare i nostri script; alcuni utenti mi hanno chiesto di fare tutorials sugli script in **linguaggio C#**: probabilmente li farò in futuro, ma per iniziare meglio usare **JavaScript**, meno elegante del **C#** ma più semplice.

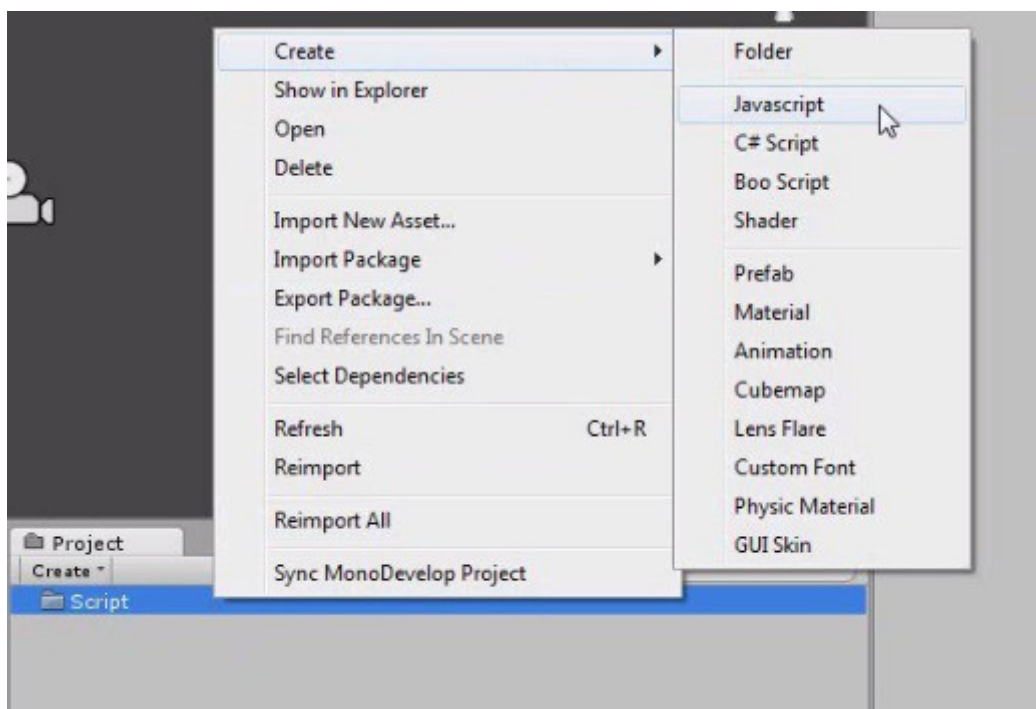
Gli **script** sono entità a parte ma per funzionare vanno associati a oggetti della scena, non per forza oggetti visuali ma anche puramente virtuali.

Facciamo le cose con ordine: clicchiamo col tasto destro del mouse sulla scheda **Project** e scegliamo **Create Folder**, creando una cartella che chiameremo, appunto, **“Script”**.



È buona norma, infatti, organizzare gli elementi del progetto in **cartelle**, a seconda della loro natura; qui abbiamo solo uno **script**, ma è meglio prendere certe buone abitudini da subito.

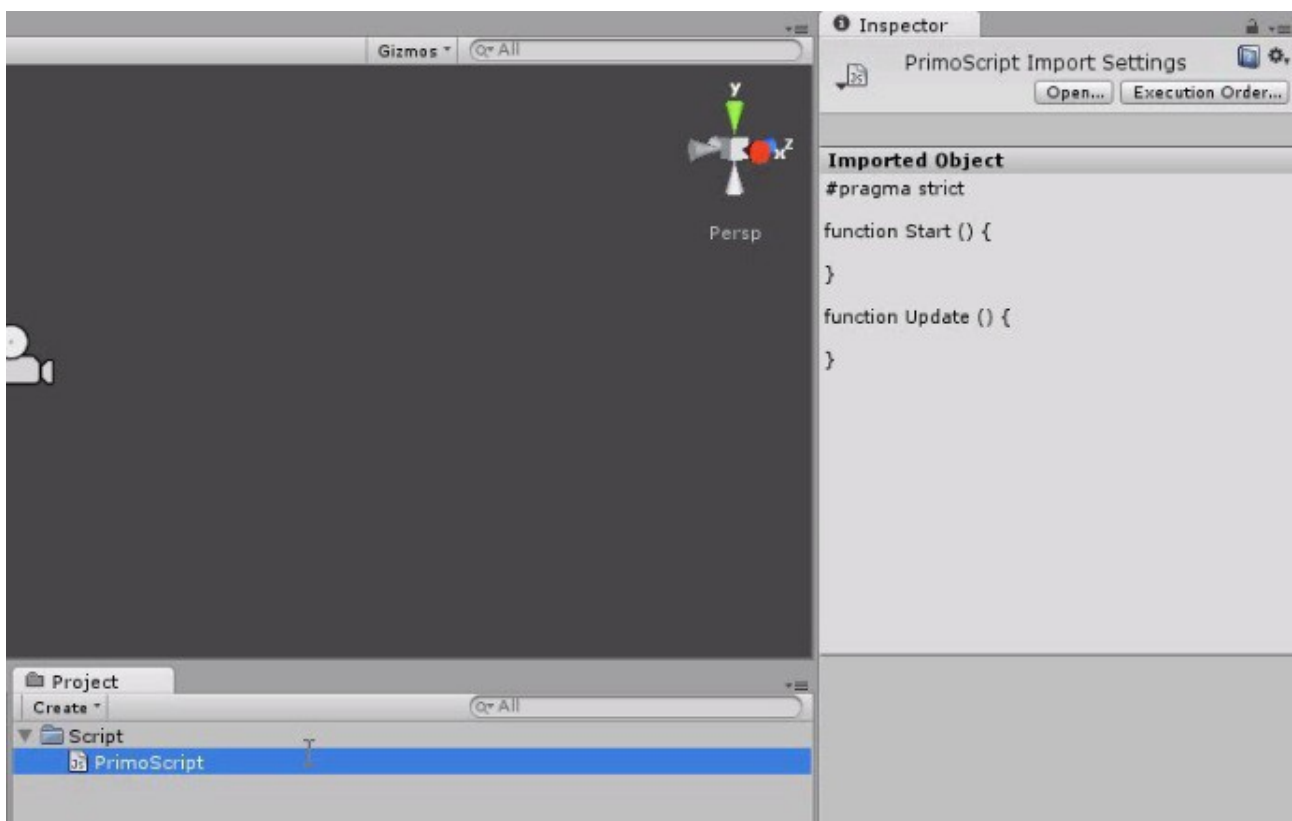
Con la cartella “*Script*” selezionata, facciamo click destro su tale elemento e scegliamo di **creare un JavaScript**.



Verrà creato un elemento con nome di default “*NewBehaviourScript*”, letteralmente “Nuove istruzioni di comportamento” (dove per “comportamento” intendiamo proprio le cose-da-fare per gli oggetti al quale verrà associato lo script).

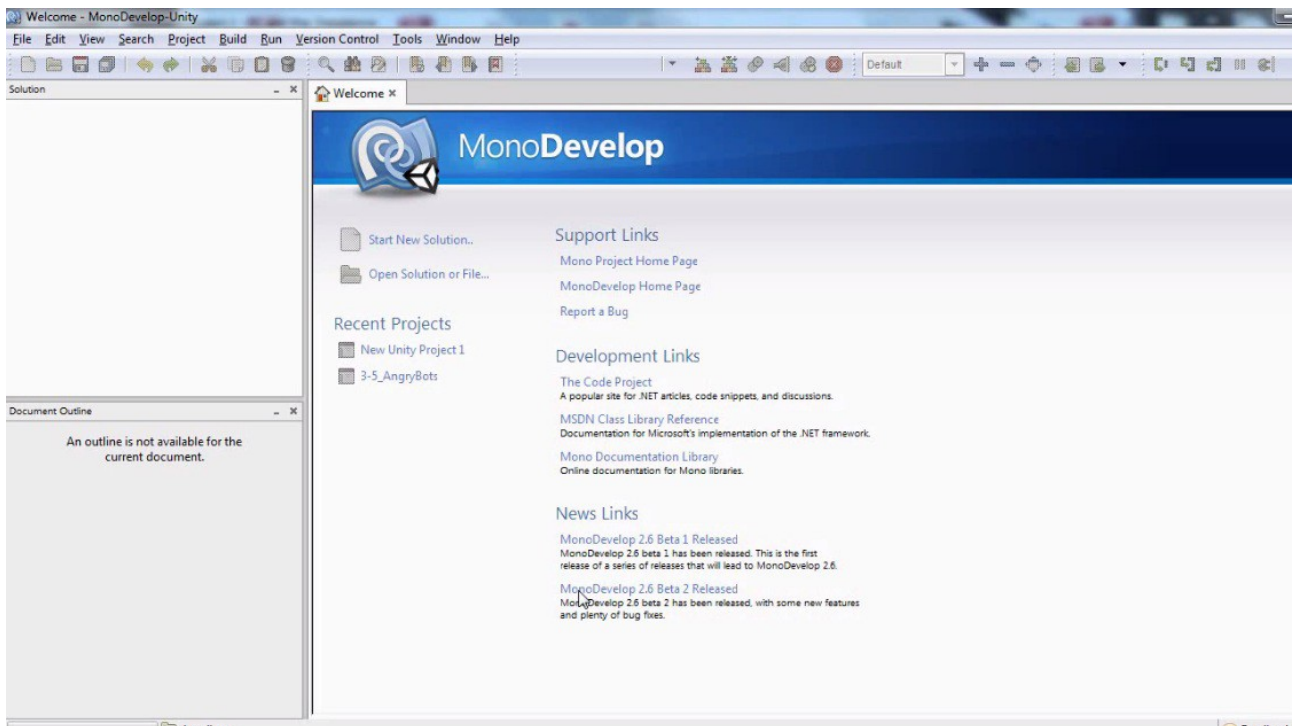
Rinominiamolo in “*PrimoScript*” e premiamo Invio, mantenendolo selezionato.

Nell'**Inspector**, anziché le schede delle proprietà, come per l'oggetto **Main Camera** (o, come visto nel capitolo precedente, il **Terrain**), abbiamo un'anteprima del **codice** dello script.



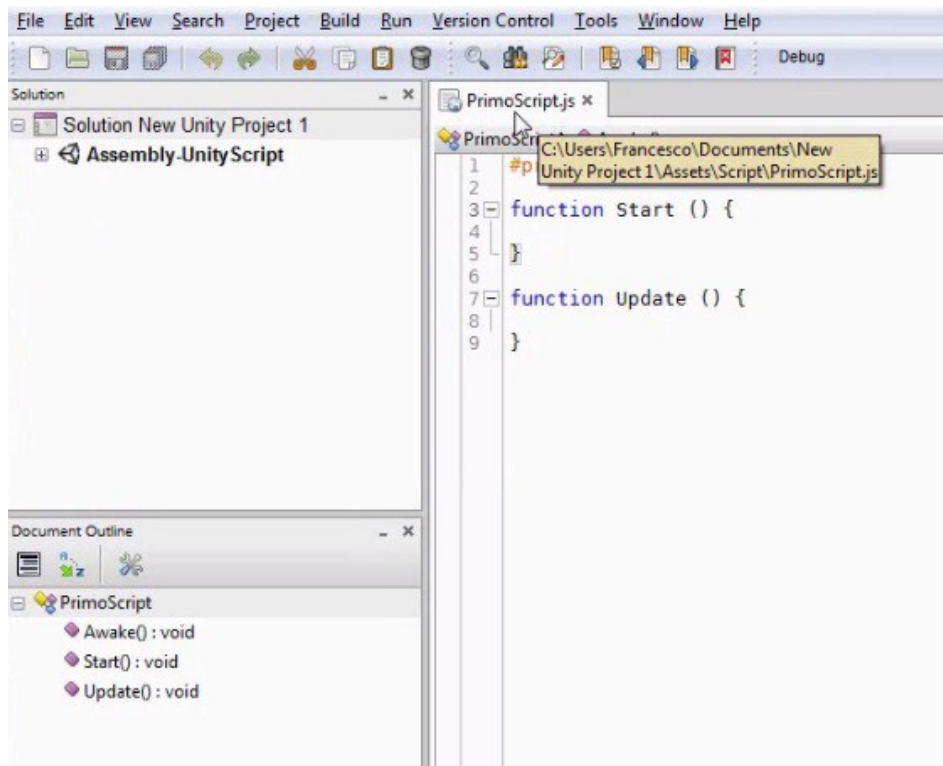
Questa **anteprima** è, però, solo un'anteprima, cioè è di **sola lettura**; per **modificarla**, facciamo doppio click su *PrimoScript*, in **Project**, oppure premiamo Invio mentre tale oggetto è selezionato.

Apparentemente non succederà nulla, ma dopo un po' (da pochi secondi a un paio di minuti, dipende da molte cose) **si aprirà un altro programma**, chiamato *MonoDevelop*.



Mono è un **IDE**, ossia un **ambiente di sviluppo integrato**, che ci consente di definire gli script con estrema facilità.

Con il doppio click su **PrimoScript**, si aprirà appunto con tale file aperto, pronto per modificarlo.



Lo script **JavaScript** è infatti un **file**, creato automaticamente su disco quando l'abbiamo definito nel **Project**, per cui potete modificarlo con qualsiasi **editor** – in realtà, potete utilizzare anche il blocco note, OpenOffice, Word, quello che volete: uno script **JavaScript** è una **sequenza di istruzioni scritte “in chiaro”**, come potete vedere.

Il mio consiglio è quello di utilizzare *Mono*, per molti motivi che scoprirete strada facendo; ve ne anticipo un paio: il testo dello script è colorato, con colori diversi se i termini utilizzati sono parole chiave del programma, inoltre a sinistra in basso abbiamo un **riepilogo delle funzioni** – vedremo presto cosa sono – e per certi elementi è previsto un utilissimo **sistema di suggerimento e completamento automatico**.

In questa prima fase, dovete conoscere **tre concetti fondamentali** di programmazione degli script (ne seguiranno altri, molto importanti, ma per il momento esaminiamo questi): **variabili, istruzioni e funzioni**.

Una **variabile** è, in un certo senso, un... **contenitore**.

Il **nome** della variabile è l'**etichetta** del contenitore, quella che ci serve per prenderlo o indicarlo ad altri, tipo “passami il contenitore blu” o “passami la scatola rossa”, e così via; il valore della variabile è il suo contenuto.

Il **contenuto** può essere di vario **tipo** e il “tipo” qui è detto “**classe**”, per cui continuando con l'esempio del mondo reale potremmo dire che il contenitore di nome “variabile1” contiene due mele, ossia due oggetti di classe “mela”.

Le **classi** di elementi... elementari (*primitivi*), nella programmazione **JS per Unity**, sono i numeri **interi**, i numeri in **virgola mobile** e le **stringhe**; le loro classi sono chiamate rispettivamente *int*, *float* e *String*.

I **numeri** vengono indicati scrivendoli direttamente, tipo 10 o, per i numeri con virgola, 2.55, quindi useremo **il punto al posto della virgola**; le **stringhe**, parole o frasi, vanno messe tra **virgolette**.

Scrivere 23.5 è diverso da scrivere “23.5”: nel primo caso si tratta di una variabile **float**, da usare con le espressioni matematiche, nel secondo di una **parola**, da stampare a video o concatenare a una frase.

Esiste poi il tipo **boolean**, che ha valori *true* e *false*, senza “” (per distinguerli dalle parole, cioè stringhe, “true” e “false”); le variabili boolean vengono usate all'interno di espressioni logiche, perché vogliono dire “vero” (**true**) e “falso” (**false**).

Come anticipato, in questa puntata creeremo un semplice **timer**.

Se pensate al **timer**, che tipo di informazione pensate di dover **memorizzare** e ricavare?

Beh... il **tempo**! Stabiliamo, in particolare, di voler esprimere il tempo in **secondi**, per cui ci servono dei numeri **interi** (quindi il tipo **int**; il **float** invece va bene se vogliamo tenere traccia di decimi e centesimi di secondo).

Il valore del **tempo trascorso** va messo da qualche parte, perché vogliamo ricavarlo, per saperlo e per mostrarlo a video; ci serve, quindi, una **variabile**, di tipo intero, come detto un attimo fa.

Abbiamo detto anche che il **contenitore** dell'informazione deve avere un'etichetta (un nome), per poterla richiamare in seguito.

Stabiliamo di chiamare il nostro contenitore “*tempoRimanente*”.

Nella finestra del programma **Mono**, con il file **PrimoScript.js** (**js** sta per “file di tipo **JavaScript**”, ovviamente), sto mettendo un paio di righe tra “**#pragma**” e “**function**

```
1  #pragma strict
2
3  var tempoRimanente : int = 30;
4
5  function Start () {
6
7  }
8
9  function Update () {
10
11 }
```

Start”, semplicemente premendo Invio mentre il cursore del mouse è lì, e sto scrivendo:

```
var tempoRimanente : int = 30;
```

Fate attenzione agli **spazi** e a **maiuscole** e **minuscole**.

Questa è un'**istruzione**, il secondo dei tre concetti da sapere.

Un'istruzione è un **comando** da dare al programma. È composta da varie **parti**, a seconda del tipo di istruzione; questa è **un'istruzione di dichiarazione e assegnamento di una variabile**.

In **JavaScript**, un'istruzione termina con il **punto e virgola**.

Stiamo dicendo a **Unity**: definiamo una variabile (*var*) chiamata **tempoRimanente** di tipo (“*classe*”) **intera** (due punti seguito dal nome della classe, quindi **tempoRimanente : int** significa “**di tipo intero**”), inoltre a questa variabile diamo il valore numerico 30.

Abbiamo appena creato un contenitore con etichetta “*tempoRimanente*” contenente un numero: 30.

Non stiamo specificando “secondi” o altro: per Unity, quel 30 è solo un numero, siamo noi che lo useremo per contare il tempo... a Unity non interessa il suo scopo: potrebbero essere 30 ore, 30 capre, 30 mele, è **solo un valore, 30, da usare con qualche espressione algebrica** (che verrà realizzata sotto forma di istruzione, nello script).

Gli **script** sono **successioni di istruzioni**, da eseguire in successione o con “salti” a varie parti, come vedremo; in realtà, anche solo la riga appena scritta costituisce uno **script**, perché quello che fa è creare una variabile... il problema è che non fa nient'altro, quindi non serve a molto!

Le **funzioni**, il terzo argomento da sapere, sono **blocchi di istruzioni all'interno degli script**.

A differenza delle istruzioni scritte “nello spazio globale” dello script, come quella appena fatta, le istruzioni scritte all'interno di una **funzione** vengono eseguite solo se la funzione viene “**chiamata**” (si può dire anche “**invocata**”) da qualche parte nel programma.

Una funzione può essere invocata tutte le volte che vogliamo e, ogni volta, **eseguirà le operazioni contenute al suo interno**.

Quando creiamo uno **script**, Unity crea automaticamente **due funzioni particolari**, **Start** e **Update**, ossia rispettivamente *Avvia* e *Aggiorna* (si trovano anche in **PrimoScript**, come avrete notato).

Ogni **funzione** ha un **nome**, che serve appunto a invocarla da altre parti del programma e a far eseguire le sue istruzioni; in un certo senso, tutti noi eseguiamo delle funzioni: quando prepariamo il caffè eseguiamo una serie di operazioni che servono, appunto, a preparare il caffè... chi ha visto il terzo episodio di Guerre Stellari ricorderà invece il famoso “ordine 66”: invocato dall'imperatore, faceva eseguire delle azioni ai cloni, azioni che quindi erano già scritte nel codice dei cloni ma che dovevano essere invocate per applicarle... ecco, lo stesso avviene negli script per i nostri elementi: noi le scriviamo, ma se non le invochiamo da qualche parte, non vengono eseguite.

La **dichiarazione** di una funzione **termina con la parentesi graffa chiusa**, senza punto e virgola dopo.

*[**NOTA**: su concetti e strumenti di programmazione c'è molto, MOLTO altro da dire, visto che esistono costrutti particolari per gestire il flusso delle operazioni, come gli if-then-else, i **cicli** for e while ed elementi più complessi come le **classi** e gli **oggetti**, tuttavia per questo capitolo – rivolto a chi non sa proprio nulla di programmazione e sta iniziando proprio con Unity e JavaScript – ci fermiamo a questi tre: **variabili, istruzioni e funzioni**.]*

La **dichiarazione** di una funzione avviene quindi con la **parola chiave function**, seguita dall'**etichetta** della funzione, da una coppia di **parentesi tonde** (che servono per fornire dei dati alla funzione... lo vedremo un'altra volta) e da una coppia di **parentesi graffe**: tutto quello che finisce racchiuso nelle parentesi è il “**corpo**” della funzione, il set di istruzioni da eseguire quando la si invoca.

Se una **variabile** viene dichiarata **dentro** una funzione, questa vivrà solo nella funzione, quindi non sarà accessibile da fuori e perderemo i dati una volta conclusa la funzione: ecco perché ho dichiarato e inizializzato “**tempoRimanente**” **fuori** da qualsiasi funzione, all'inizio dello script.

Ora le **funzioni** potranno modificarne il valore, perché la variabile è stata definita fuori ed è quindi accessibile a tutte.

Le **funzioni** *Start* e *Update* sono funzioni particolari, in uno script *Unity*.

La **funzione** *Start* viene invocata **automaticamente** dal programma non appena l'**oggetto che possiede lo script** prende vita nel progetto, quindi viene utilizzata per **inizializzare** qualche caratteristica dell'elemento o svolgere **operazioni preliminari**.

La funzione *Update* viene invocata **automaticamente** dal programma **in continuazione**, ossia ad ogni “tick”, ad ogni frame o aggiornamento del gioco, anche 25, 30, 60 o più volte AL SECONDO.

Potremmo pensare di usare la funzione *Update* e di ritardarla in qualche modo per 1 secondo dopo aver decrementato il valore della variabile *tempoRimanente*, realizzando così il nostro **Timer**.

Il problema è che **la funzione Update NON PUO' essere rallentata o congelata**: per natura, deve scorrere continuamente... possiamo, però, creare **una funzione a parte**, da eseguire **separatamente**, e quella sì che possiamo rallentarla, con un'istruzione particolare... bene, facciamolo!

Scriviamo, in un punto del file fuori da ogni funzione:

```
function decrementa()  
{  
    while(true)  
    {  
        tempoRimanente = tempoRimanente - 1;  
        yield WaitForSeconds(1);  
    }  
}
```

Fate attenzione a **maiuscole e minuscole** quando digitate nomi o comandi: **JavaScript** distingue tra minuscole e maiuscole, per cui per lui **Wait** e **wait** sono due cose completamente diverse!

All'interno della **funzione**, c'è un comando particolare, un'istruzione con un suo blocco di comandi: *while(true)* seguita da una **coppia di parentesi graffe** (quindi un **blocco di istruzioni**, proprio come per le funzioni) con dentro due istruzioni.

while è un ciclo, ossia un **blocco** che può essere **ripetuto più e più volte**, se certe **condizioni** (specificate all'interno delle parentesi tonde o in altri modi, con comandi particolari) vengono soddisfatte.

In questo caso, abbiamo “*while(true)*”, con *true* che è una parola chiave ed è anche un valore: significa “vero”, o “1”.

L'istruzione “*while(true)*” significa quindi: “**finché è vero**” (ossia: “**sempre**”) “**esegui il contenuto del blocco.**”

Il contenuto del blocco sono le due istruzioni

```
tempoRimanente = tempoRimanente - 1;  
yield WaitForSeconds(1);
```

La **prima istruzione** è chiara: **decrementa** il valore della variabile “*tempoRimanente*”.

La **seconda** sfrutta un comando **nativo** e una funzione nativa di **Unity**; il tutto vuol dire “*attendi un secondo*”.

Unity mette a disposizione molte **funzioni e comandi nativi** per svolgere varie operazioni.

È possibile conoscere queste funzioni con la **documentazione** online, i **manuali** e... i **tutorial**: in questo capitolo, ad esempio, state facendo la conoscenza di “*yield WaitForSeconds*” e il numero di secondi tra parentesi.

Ok, il senso della **funzione** è ora chiaro: **ripetere in continuazione (“while(true)”) un'operazione di decremento di un valore di variabile esterna, ritardando il termine di ciascun ciclo while di un secondo**, così da fare un decremento al secondo, all'infinito.

Problema: questa funzione, *decrementa()*, **non viene invocata** da nessuna parte, quindi non partirà mai.

Per risolvere il problema, all'interno del blocco della **funzione** *Start* scriviamo questa istruzione:

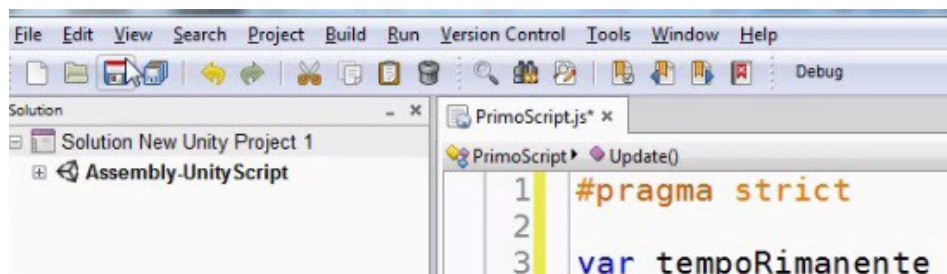
```
decrementa();
```

In pratica, stiamo invocando *decrementa* **all'avvio** (*Start*), alla creazione dell'oggetto che possiede questa funzione...

```
4  
5  
6 function Start () {  
7     decrementa();  
8 }  
9
```

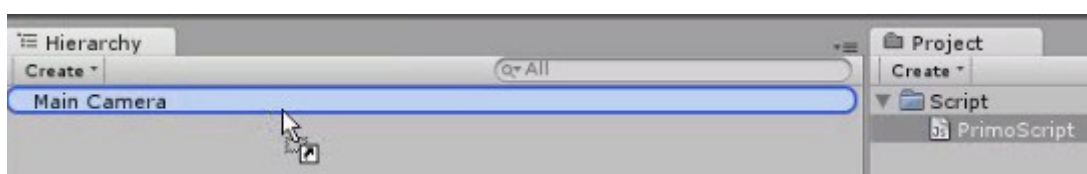
... già, ma quale oggetto? In realtà, **non abbiamo associato lo script ad alcun oggetto della scena**, quindi non verrà mai eseguito.

Prima di procedere, fate click sull'icona del dischetto in **Mono**, in modo da **salvare lo script corrente**, anche perché se non salviamo le modifiche queste non verranno applicate, all'avvio del gioco. Quando fate una modifica ad uno **script**, accanto al suo nome in **Mono** apparirà un *, come visibile nell'immagine qui sotto; assicuratevi di non avere **file con modifiche non salvate**, ossia con l'asterisco, prima di avviare il gioco.



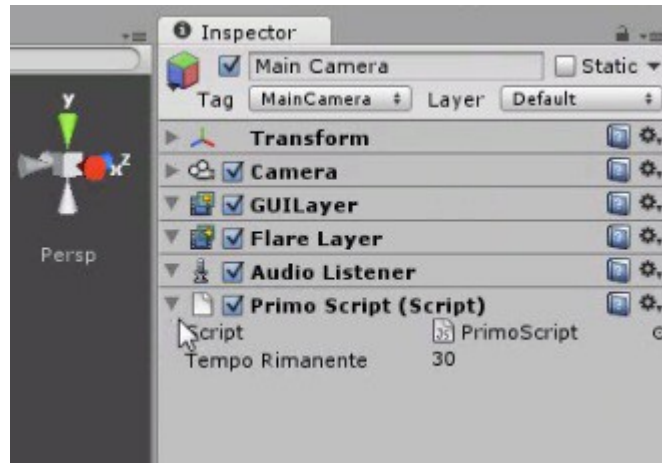
Fatto questo, torniamo a **Unity 3D**, ancora aperto... consiglio a parte: se avete due monitor, mettete **Unity** in uno e **Mono** nell'altro, anziché passare da una finestra all'altra nello stesso monitor.

Adesso, **trasciniamo lo script dal Project sull'oggetto Main Camera**, nella **Gerarchia**, come visibile nell'immagine seguente.



Selezionando ora **Main Camera**, vedremo nell'**Inspector** una scheda **Script** con il nostro **Script**: ciò significa che lo script è **associato alla telecamera**, che all'avvio eseguirà **Start** e, poi, tutto il resto.

Non solo: significa anche che **uno script**, come altri elementi, **può essere associato a più oggetti della scena** o direttamente a un **Prefab**, il che consente di riutilizzare certi codici... giusto per fare un esempio, gli **script di intelligenza artificiale** di un particolare tipo di **nemici** possono essere forniti ai **Prefab** degli stessi, per cui tutte le istanze (ricordate il discorso **Prefab-istanze?**) dei nemici avranno quei codici.



Clicchiamo ora sul pulsante **Play** per avviare il gioco... beh, non succede nulla.

Il motivo è semplicissimo: lo **script** fa quello che deve fare, ma **non c'è alcuna istruzione che gli dica di mostrare a video il valore** della variabile “*tempoRimanente*”!

Come ultimo sforzo per questo capitolo, parliamo di una **funzione nativa di Unity** che serve a gestire **l'interfaccia grafica della finestra**.

Questa funzione ci serve **per far apparire una scritta con il valore della variabile**.

Riprendiamo lo **script** e scriviamo, nello “spazio globale” (ovvero, fuori da qualsiasi funzione) dello script:

```
function OnGUI()  
{  
    GUILayout.Label(tempoRimanente.ToString());  
}
```

Salviamo lo script.

OnGUI è una **funzione speciale** che contiene le istruzioni per **scrivere o disegnare sull'interfaccia grafica 2D** (GUI, Graphic User Interface, interfaccia grafica utente) dell'elemento.

Al suo **interno** vanno poste istruzioni particolari, perlopiù native di **Unity**, che si trovano nella **documentazione (sezione GUI, appunto)** e servono ad **inserire elementi visivi** sull'interfaccia; in questo caso, stiamo utilizzando la **funzione “Label()”**, una funzione che appartiene all'elemento nativo **“GUILayout”**, per cui stiamo dicendo **“usa la funzione Label di GUI Layout”**.

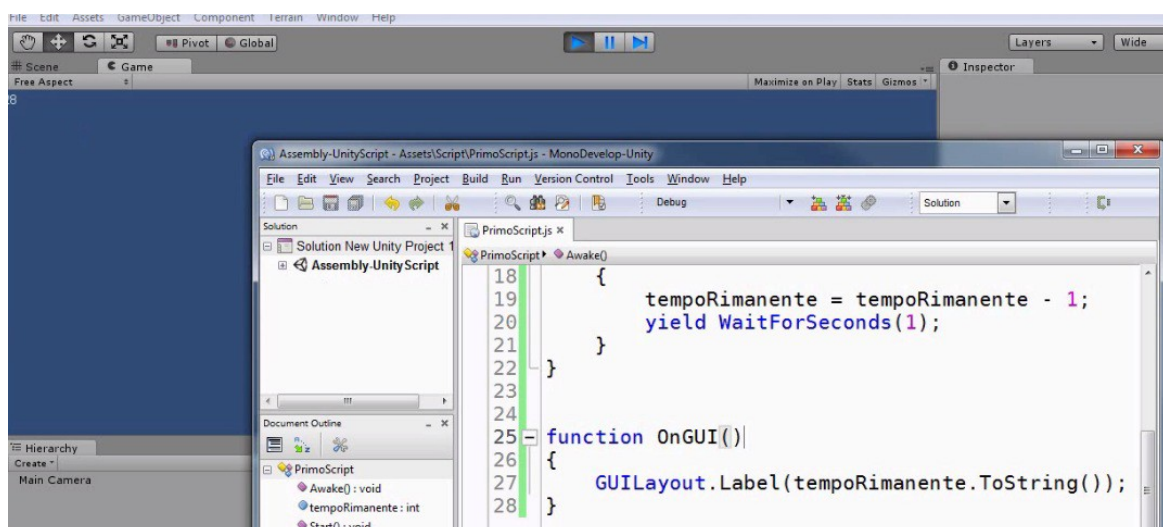
Label significa etichetta, per cui si tratta di scrivere sul **Layout**, cioè a video, quello che c'è tra le parentesi (**il parametro o argomento della funzione Label**)... e cosa c'è tra le parentesi?

C'è **“tempoRimanente.toString()”**.

Sappiamo cos'è **tempoRimanente**: è una variabile, da noi definita, di tipo numerico. Cos'è *toString()*?

È una **funzione nativa di Unity e JavaScript**, per i dati di tipo intero (numerico), che consente di **trasformare il valore numerico della variabile che la usa**, nel nostro caso *tempoRimanente*, in **una parola**, quindi ora il 30 iniziale è sempre trenta, ma questa volta è **una parola, non un numero**, e può essere **scritto a video** dalla **funzione Label di GUILayout**.

Salvato lo script, rifacciamo partire il gioco in **Unity**: questa volta, a video vedremo, **stampato in alto a sinistra, il valore della variabile tempoRimanente**.

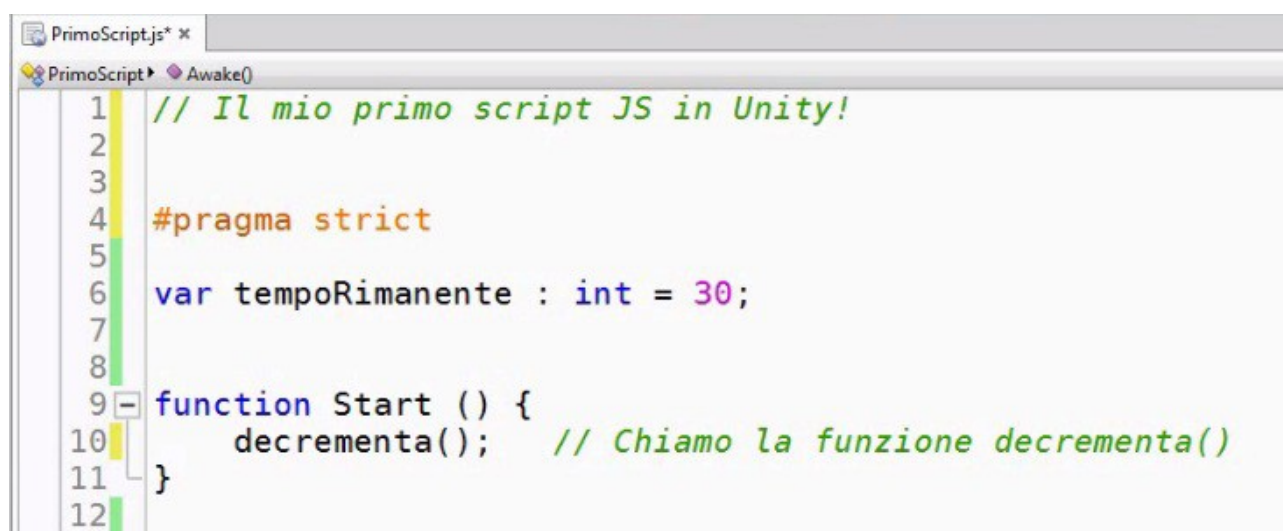


Un'ultima nota a parte (ma non troppo): i **commenti in linea**.

Se volete **prendere appunti nel file di codice**, ad esempio per ricordarvi in futuro a cosa serve un'istruzione o una variabile, potete scrivere dei **“commenti”**, che sono **righe o porzioni di riga che verranno ignorate da Unity** e che servono appunto solo **a chi legge il codice**.

Per scrivere un commento, scrivete **due /**, senza spazi, quindi il vostro commento; i due / e le parole che seguono dovrebbero apparire in **verde**.

Potete mettere un commento **a destra di un'istruzione**, per spiegare a cosa serve; nell'immagine seguente, ho messo il **commento** *“chiamo la funzione decrementa”* accanto all'istruzione **decrementa()**; , all'interno della **funzione Start**.



```
1 // Il mio primo script JS in Unity!
2
3
4 #pragma strict
5
6 var tempoRimanente : int = 30;
7
8
9 function Start () {
10     decrementa(); // Chiamo la funzione decrementa()
11 }
12
```

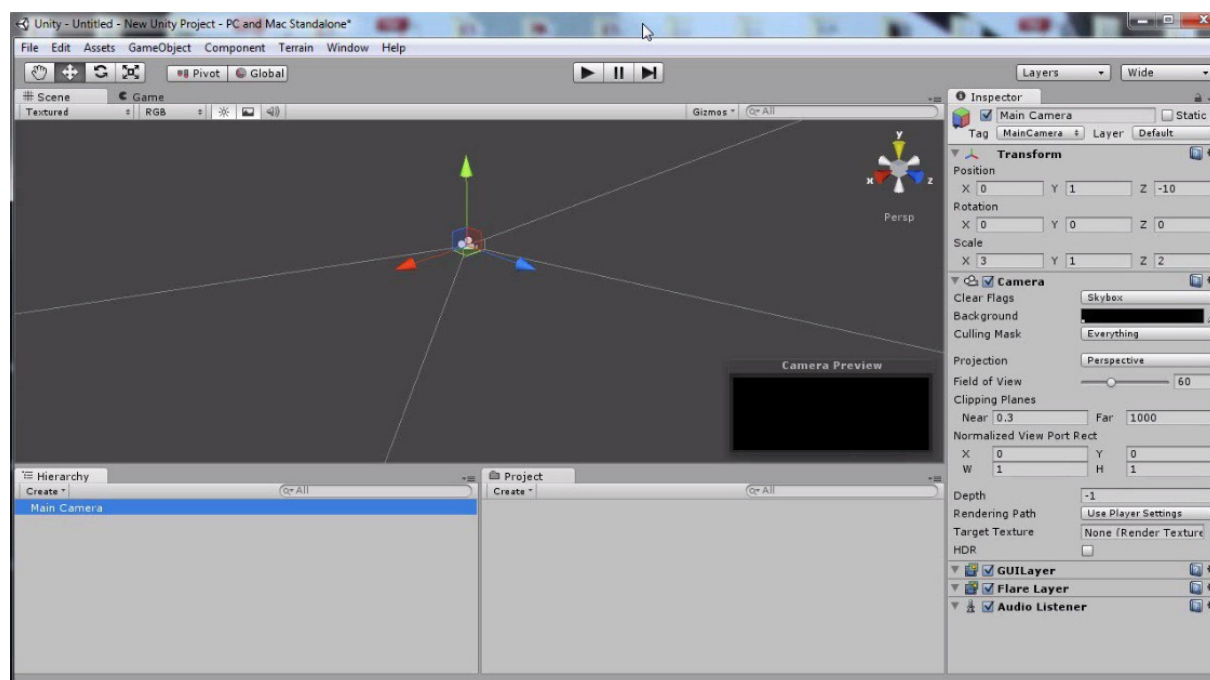
Quanto visto in questo capitolo è solo la **punta dell'iceberg per quanto riguarda gli script in Unity**, ma riprenderemo l'argomento – con progetti più interessanti – tra un paio di capitolo... per il momento, chiudiamo questa parentesi (che serviva da primo approccio con il lato **scripting di Unity**, nient'altro) e torniamo sul percorso principale, fatto per ora di tanti piccoli elementi separati che però in seguito ci serviranno per comprendere il “disegno più grande”, per così dire.

* * *

Le basi di Unity 3D 3.x e 4.x – 5: Physics Materials; pressione dei tasti; movimenti

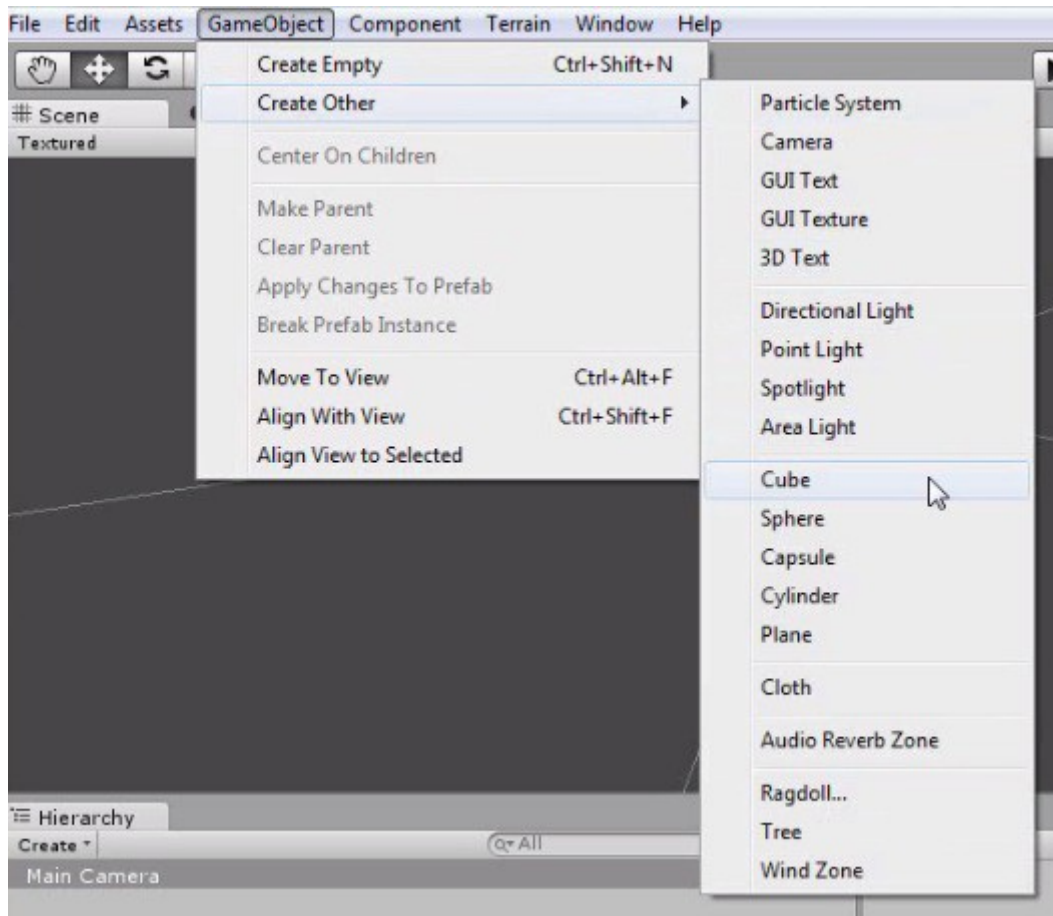
Eccoci al **quinto tutorial** sulle basi del motore di gioco **Unity 3D**; in questo capitolo vedremo come aggiungere un paio di **caratteristiche fisiche** agli oggetti della scena, come **caduta e rimbalzo**... ma non solo: vedremo anche come recuperare via codice gli **eventi** come la pressione di un tasto della **tastiera** e far effettuare delle operazioni agli oggetti mediante **script**.

Iniziamo con la prima parte e... con un **nuovo progetto**, completamente vuoto, ad eccezione, come sappiamo, dell'oggetto *Main Camera*.

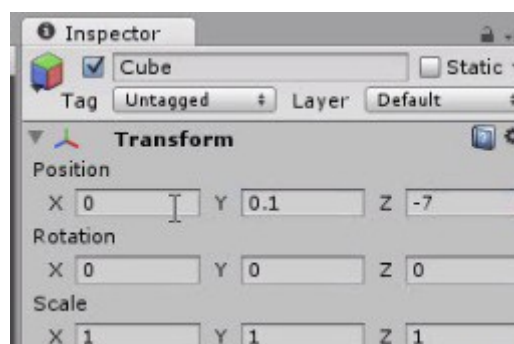


Posizioniamo questo oggetto in **X 0, Y 1 e Z -10**, mediante i campi **Transform Position** nell'**Inspector**; impostiamo inoltre 0 per tutti e tre i campi **Rotation**.

A questo punto aggiungiamo una **primitiva geometrica cubo** alla scena mediante **GameObject – Create Other – Cube**; per il nostro scopo, ossia creare una superficie di rimbalzo, andrebbe bene anche un **Terrain** o un **Plane**, ma sto aggiungendo un cubo per fare **un esempio di scaling non uniforme**.

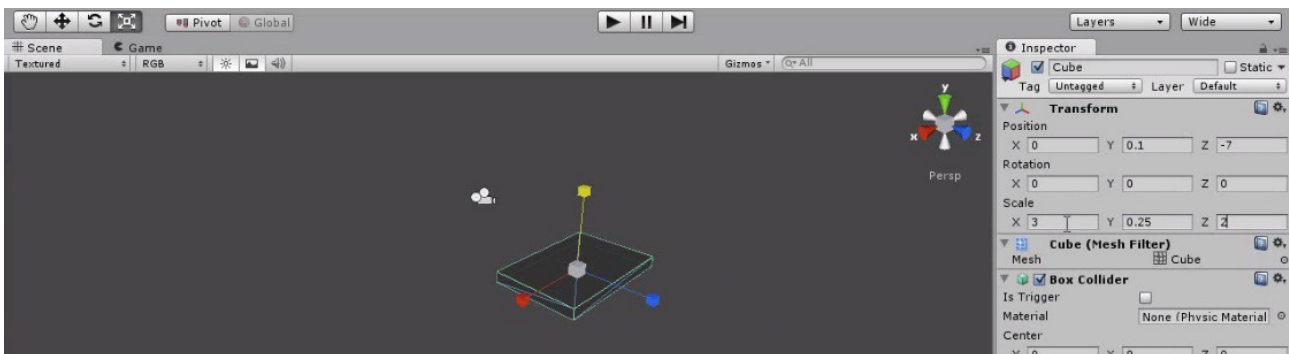


Posizioniamo il cubo **leggermente in avanti rispetto alla telecamera**, in modo da farlo entrare nell'inquadratura, ad esempio in **Position X 0.0, Y 0.1 e Z -7**.



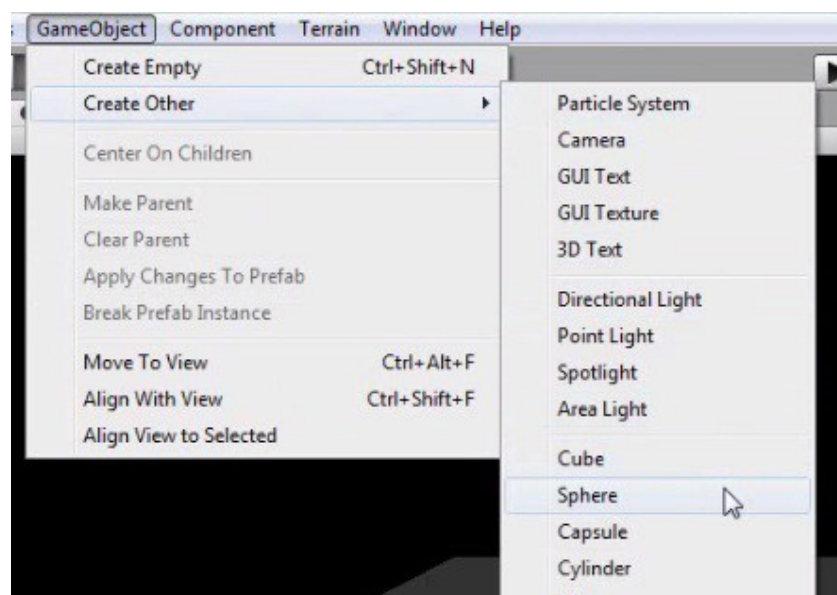
A questo punto, **per schiacciare un po' il cubo verticalmente**, possiamo procedere **liberamente** o con il manipolatore di scaling e trascinando solo la maniglia verticale in basso, oppure in **maniera parametrica** scrivendo, ad esempio, **0.25** (cioè un quarto delle dimensioni originali) nel campo **Scale Y**.

Impostiamo inoltre **Scale X a 3 e Z a 2**.



Una piccola **nota** su questi valori: le variazioni a 3 o più valori dopo la virgola, dovute perlopiù alle trasformazioni libere, non parametriche (ad esempio tra 0.2586 e 0.2587), in genere sono **impercettibili per l'utente**, tuttavia **Unity** le tiene in considerazione quando deve fare i **calcoli per collisioni e rendering**, e più valori lasciamo dietro la virgola, senza arrotondare, **più calcoli Unity dovrà fare**; se, quindi, non vi serve tutta quella **precisione, arrotondate**, ad esempio in una scena simile da 0.2587 si può passare a 0.259 o anche a 0.26 senza far male a nessuno.

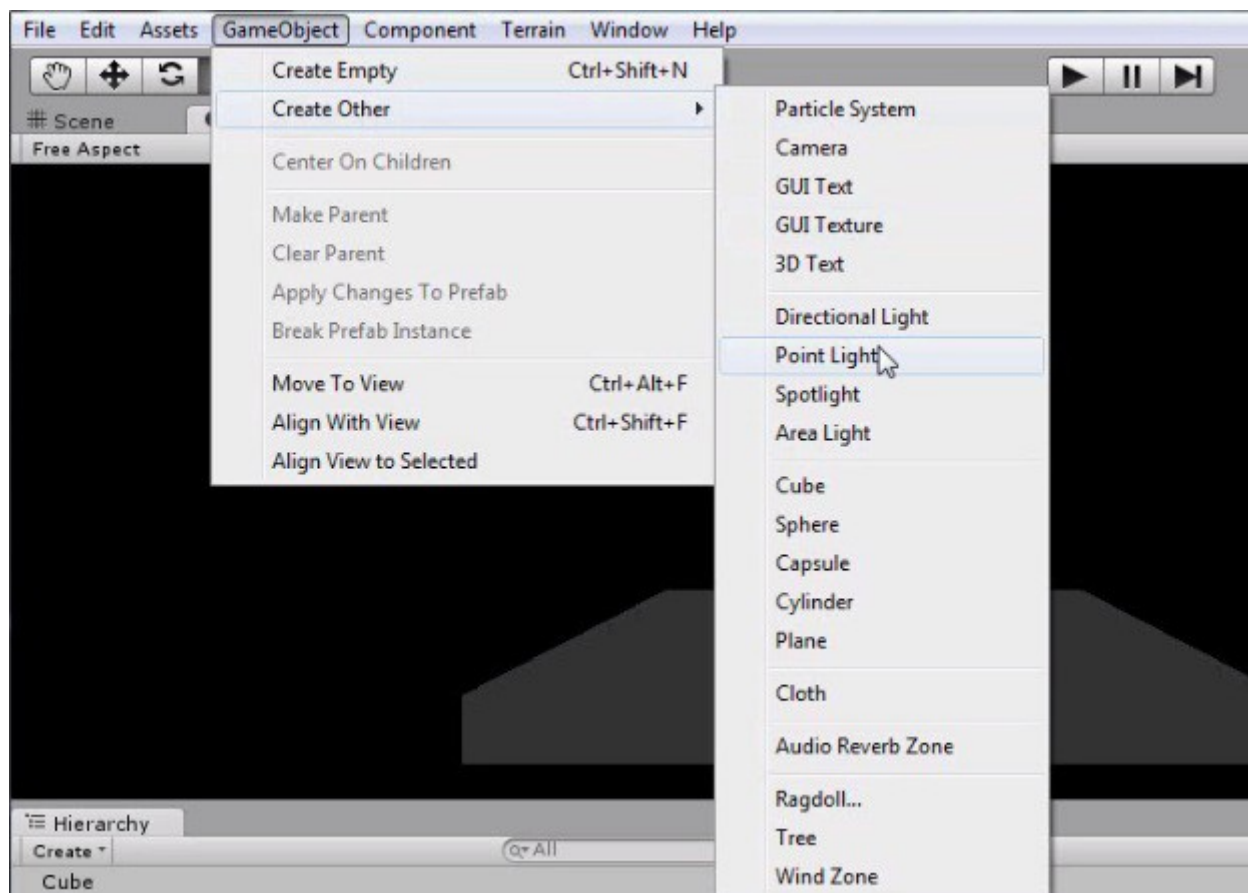
Aggiungiamo un **Game Object** geometrico, una **sfera**, con *GameObject – Create Other – Sphere*.



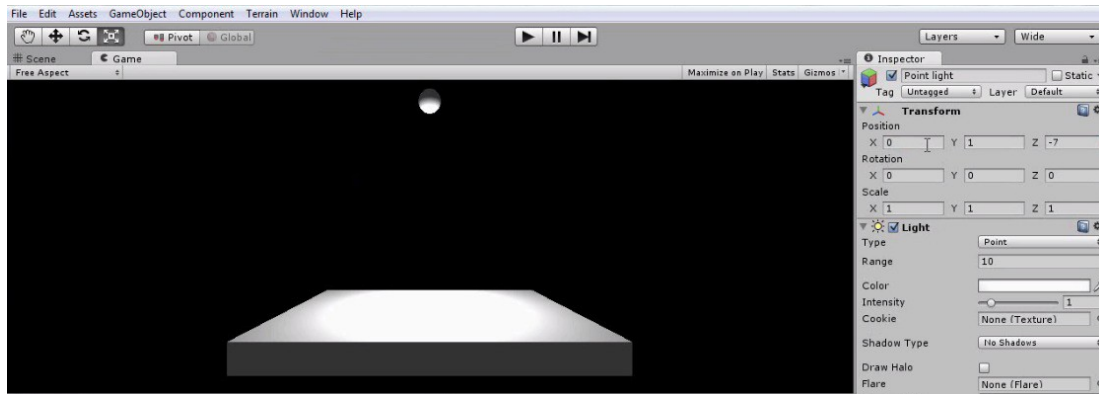
Posizioniamo questa **sfera** sulla **piattaforma** costituita dal parallelepipedo – ricavato dal **Cube** – ma leggermente **in alto** rispetto a tale oggetto, ad esempio in **Position X 0, Y 2.5 e Z -7**; questa volta, facciamo uno **scaling uniforme** impostando **0.25** per tutti e tre i campi **Scale XYZ**.



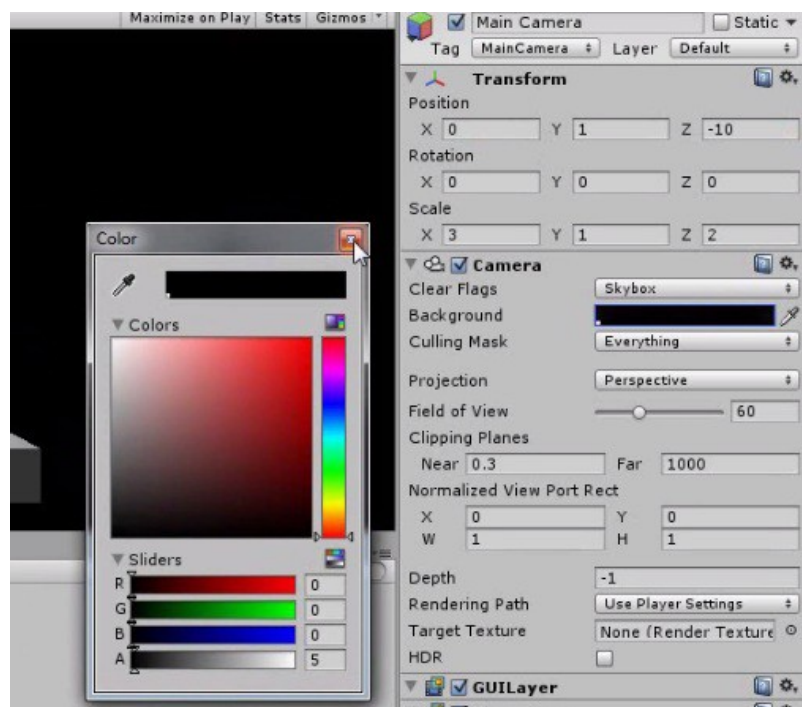
Adesso sto aggiungendo anche un **Game Object Point Light** in posizione **Position X 0, Y 1 e Z -7**, giusto per non vedere gli elementi del tutto grigi, e sto cambiando in nero lo **sfondo dell'universo virtuale**, mediante il campo **Background della telecamera**.



Nell'immagine seguente: la scena illuminata, dopo l'inserimento della fonte di luce Point Light (immagine nella pagina precedente).

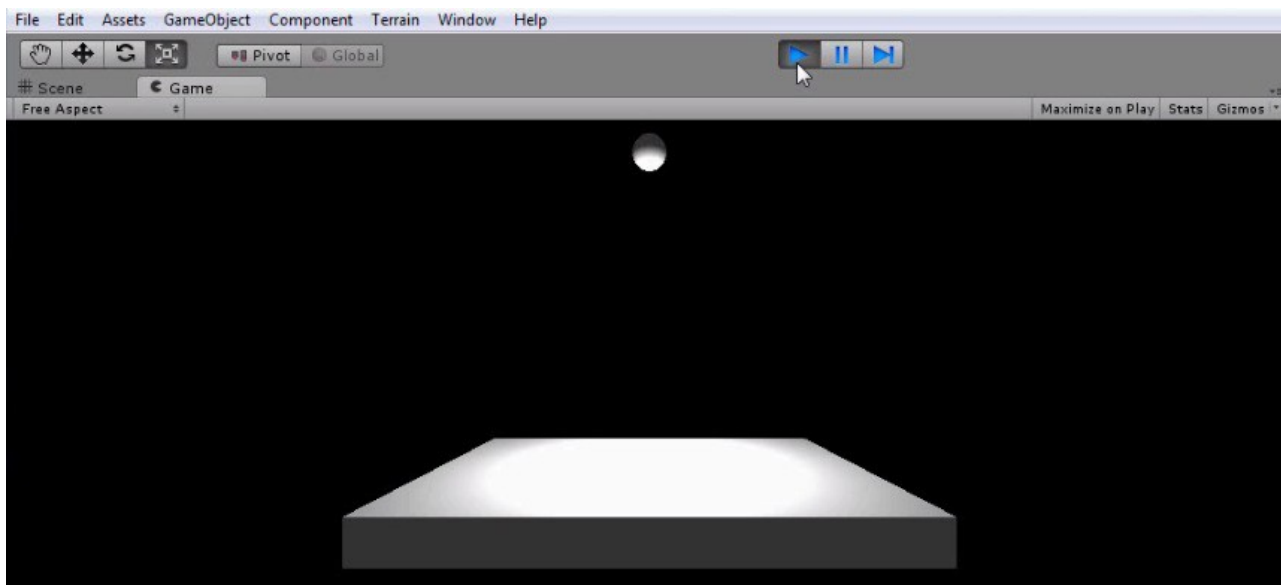


Nell'immagine a destra: cliccando sulla casella **“background”** (sfondo) apparirà una casella di scelta del colore (**Color**), dove potremo selezionare il colore da dare uniformemente allo sfondo dell'**universo virtuale** (in questo caso, nero).



Diamo un'occhiata alla **finestra Game**: abbiamo la **sfera sospesa nel vuoto** ad una certa distanza dalla **piattaforma**... che in realtà è anch'essa sospesa nel vuoto, visto che non c'è un **pavimento** di sotto!

Ad ogni modo, premendo **Play** per avviare la simulazione in real time (immagine nella pagina seguente), vediamo che... **non succede niente**.



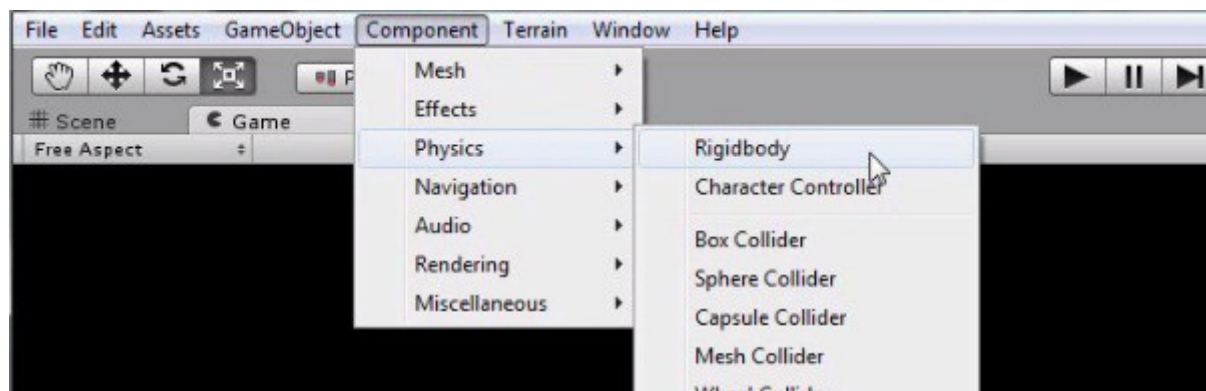
Il motivo è il seguente: **di base, gli oggetti non partecipano alla simulazione fisica della scena.**

Questa scelta è dovuta a molti fattori, principalmente di carattere pratico: la simulazione fisica richiede **molte risorse di calcolo**, quindi se non c'è bisogno (se non si dice in maniera esplicita che un certo oggetto deve avere proprietà fisiche), allora **non lo si include nelle simulazioni fisiche** della scena.

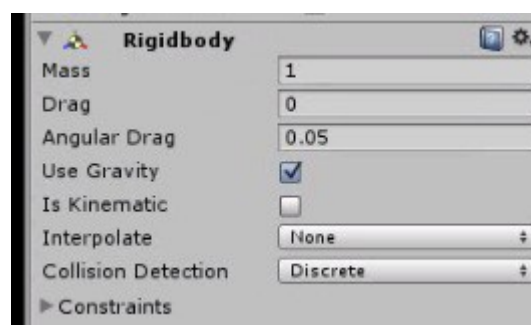
Esistono vari... “**tipi fisici**” **per gli oggetti**, a seconda della loro natura e del loro comportamento: un **fluid**o avrà caratteristiche diverse da un **tessuto**, che a sua volta è diverso da un materiale **gelatinoso** che è molto diverso da una palla o un **mattone**... iniziamo con calma con il caso che ci interessa: **la sfera**.

Le **proprietà fisiche** di un oggetto sono elementi, come le proprietà di trasformazione, il mesh filter, il mesh renderer ed altri elementi che troviamo nell'Inspector; sono, cioè, **COMPONENTI** dell'oggetto (notare la natura modulare degli elementi e delle relazioni tra gli stessi).

Per aggiungere un **COMPONENTE**, ossia una caratteristica, di **SIMULAZIONE FISICA** all'oggetto sfera selezionato, apriamo il menù **Component** e scegliamo **Physics – Rigid Body**, corpo rigido.



Vedremo apparire tale **componente** nell'**Inspector** dell'oggetto; fortunatamente, le **proprietà principali** hanno nomi intuitivi e, comunque, sono poche quelle che ci interessano.



Abbiamo in particolare il valore **Mass**, la **massa** dell'oggetto, qui intesa in... “unità di massa”: **Unity non fornisce un'unità di misura**, sta a voi decidere se 1 è un kg, un grammo, una tonnellata, e così via (d'altronde, anche per posizioni e distanze non abbiamo parlato di metri, ma di unità della scena); ovviamente, poi dovreste fare i calcoli per definire i valori degli altri elementi e renderli coerenti... se diamo 1 a un pallone da basket, di certo non potremo dare 10 a un'automobile nella stessa scena!

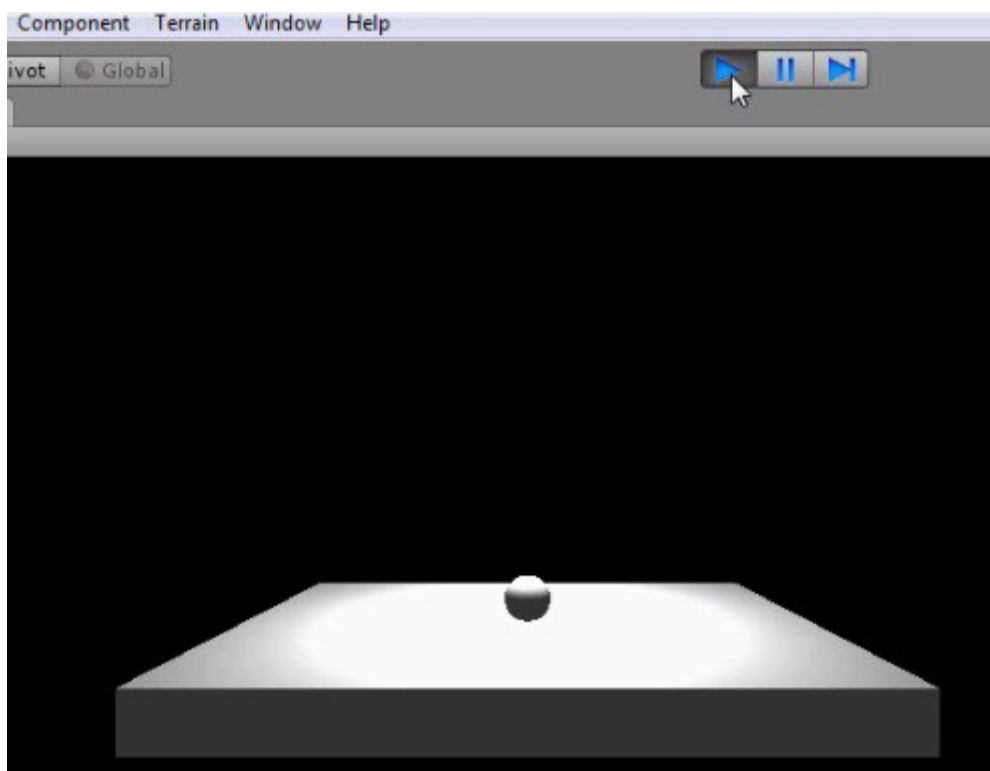
Drag e **Angular Drag** sono forze che **si oppongono al moto e alla rotazione**, frenando l'oggetto, quindi in un certo senso possiamo considerarle come **forze d'attrito** (anche se non è propriamente così), comunque per il momento non ci servono e possiamo lasciare i valori di default.

La casella **Use Gravity**, selezionata di default, ci dice che l'oggetto è **sottoposto alla gravità**.

Nota: tutti questi campi, come in generale tutti quelli presenti nell'**Inspector** di un elemento, possono essere **modificati**, in tempo reale, **via codice**, quindi ad esempio il vostro personaggio, durante il gioco, potrebbe usare un **mantello dell'invisibilità** o uno **strumento anti-gravità**, e per realizzare questi effetti non dovreste far altro che operare, **via codice**, sulle caratteristiche *Mesh*

Renderer e *Use Gravity*, gestendo lo **script** per far valere questi **effetti** per un certo periodo di tempo, oppure in base alla durata delle batterie dello strumento, fino a quando il personaggio viene colpito, eccetera... nella seconda parte di questo capitolo vi dirò come **ricavare i nomi delle proprietà di gioco** per lo scripting dal **manuale di Unity**, quindi alla fine potrete iniziare a **sperimentare** tutto quello che vi viene in mente!

A questo punto premendo **Play** vedremo, nella simulazione in tempo reale, **la pallina che cade sulla piattaforma...** fine. Non è particolarmente esaltante.



Clicchiamo sul pulsante **Stop**.

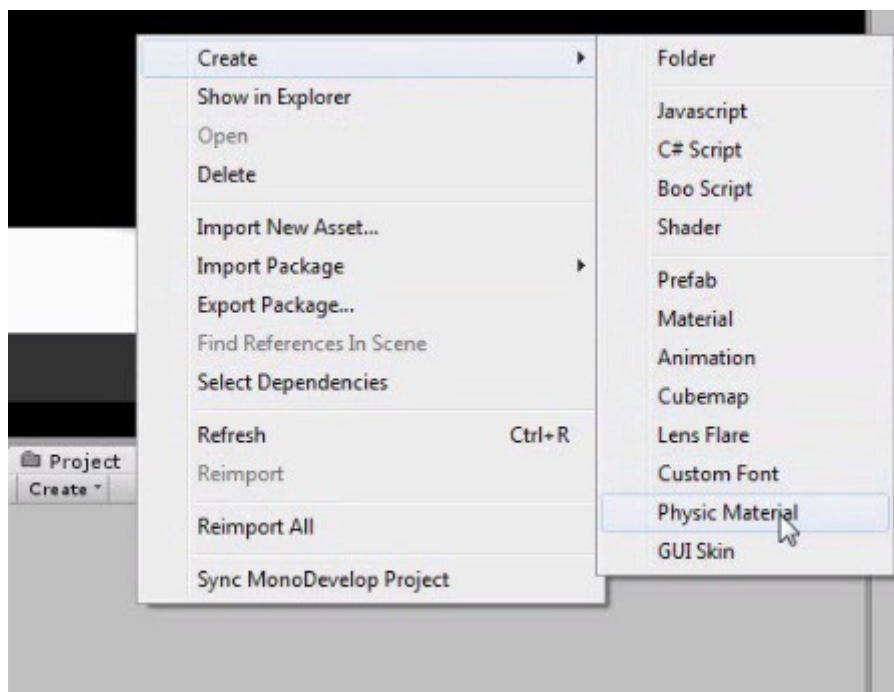
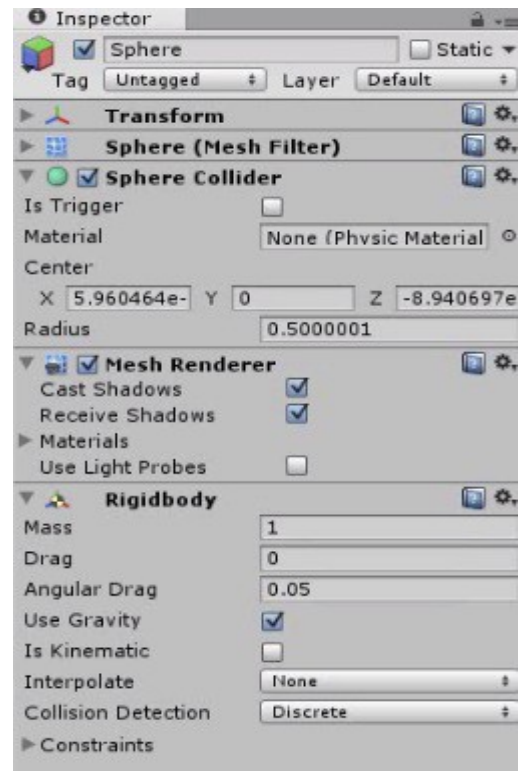
Con **Rigid Body** abbiamo detto a **Unity** che la pallina fa parte della **simulazione fisica** e le abbiamo dato qualche parametro, ma non abbiamo specificato ad esempio le **caratteristiche elastiche**, di rimbalzo, eccetera.

Queste caratteristiche vanno implementate mediante un **particolare Material**, che però non va confuso con il “**Material**” del **Renderer**, che definisce l'aspetto **visivo** dell'oggetto!

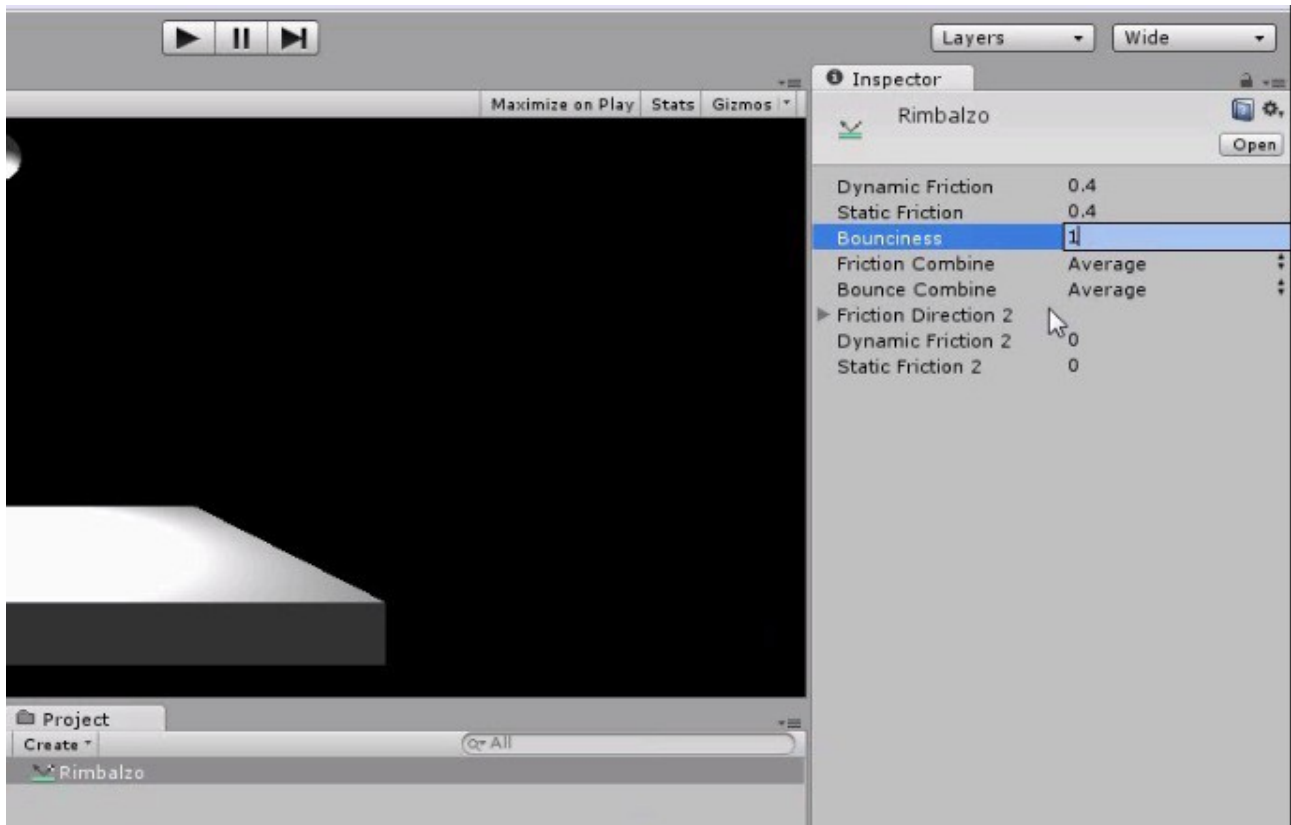
Si tratta di un **PHYSICS MATERIAL**, un materiale fisico, da definire in particolare nel campo **Material** del **Collider** dell'oggetto, nel nostro caso lo **Sphere Collider**, nell'Inspector; è diverso dai **Materials di Mesh Renderer**: in **Collider**, come suggerisce il nome, abbiamo un **Material FISICO**, per le **collisioni**, mentre in **Renderer** (visualizzazione) abbiamo i **Materials di VISUALIZZAZIONE (rendering)** dell'elemento.

I **Materials fisici** vanno inseriti nel **Project**, e questo perché da lì possono essere facilmente riutilizzati per molti elementi.

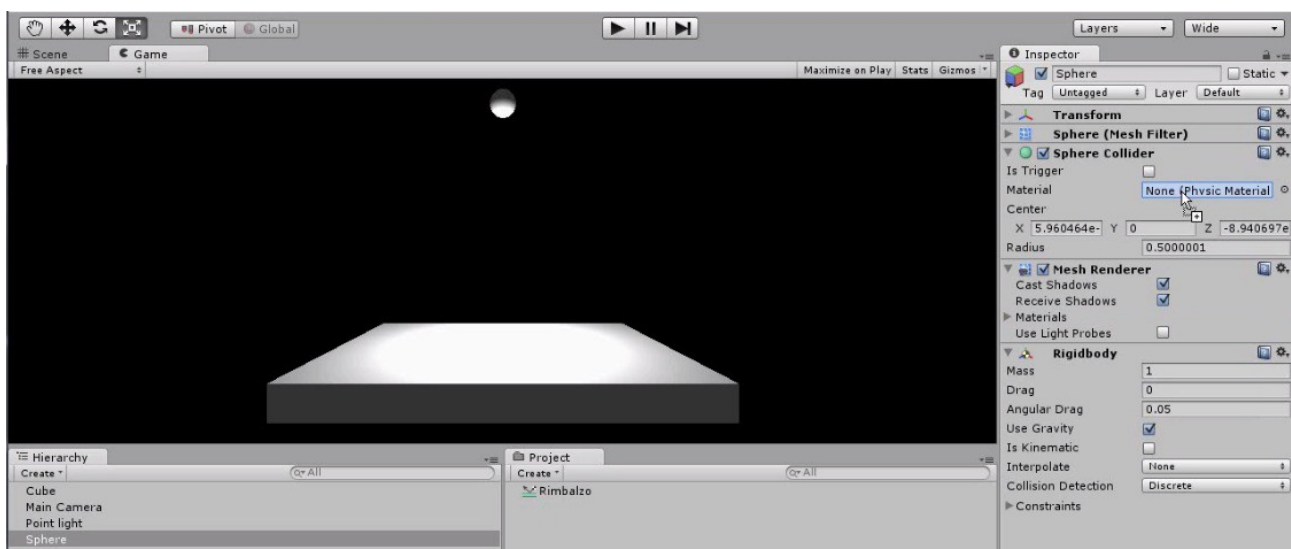
Facciamo quindi click destro sulla scheda **Project** e scegliamo **Create – Physic Material** (attenzione a non scegliere solo **Material**), chiamando questo nuovo elemento “**Rimbalzo**”.



Rimbalzo, in Inglese, si dice *Bounce*, quindi impostiamo ad esempio 1 (il massimo) come valore *Bounciness* nell'**Inspector** di questo **Material fisico**, lasciando tutti gli altri valori come di default.



Riselezioniamo l'oggetto **Sphere**, nella **Gerarchia**, quindi trasciniamo “**Rimbalzo**” dal **Project** al campo **Material** di **Sphere Collider**, per associare quel **Material fisico** alla sfera.

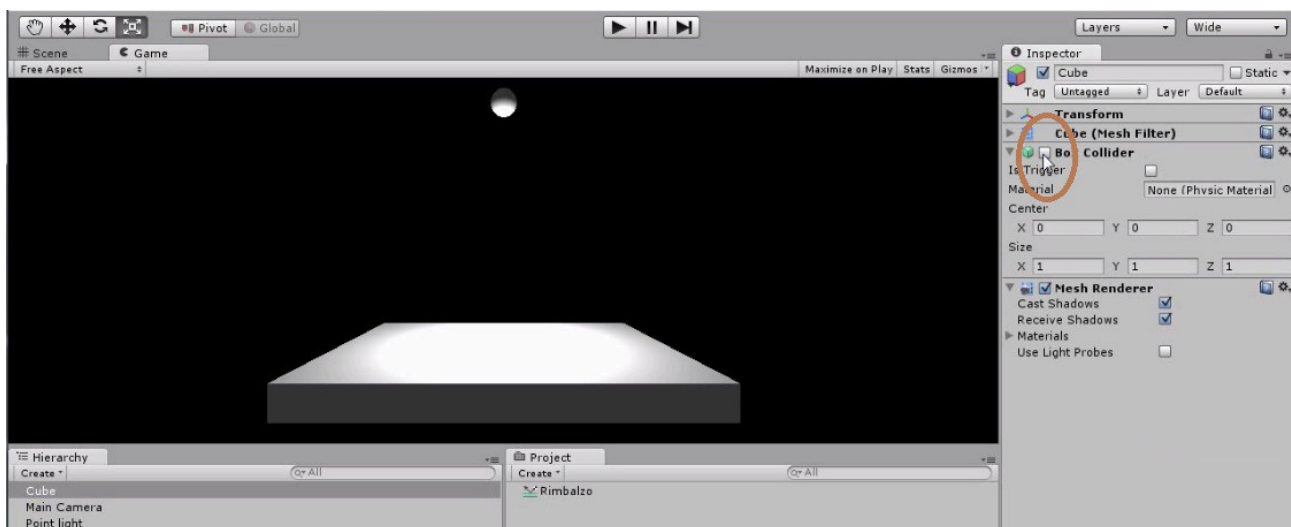


Premiamo nuovamente **Play** nella schermata di gioco ed ecco la novità: la palla ora **rimbalza** un paio di volte, con rimbalzi **smorzati**, sulla pedana!

Clicchiamo nuovamente su **Stop**.

Faccio notare che **la pedana NON è un Rigid Body** ma, essendo dotata di un **Mesh Collider**, come tutti gli oggetti di default, viene vista come ostacolo dalla palla, che cade e rimbalza rispettivamente per via di *Rigid Body* e del *Physic Material*.

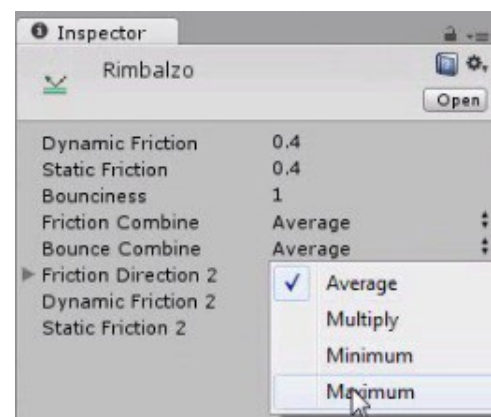
Se volete **escludere la pedana dalla simulazione fisica della collisione**, fate così: selezionate tale oggetto e, nell'**Inspector**, deselezionate la checkbox “*Box Collider*”, quindi riavviate la simulazione del gioco con **Play**.



Abbiamo visto che **la pallina fa un paio di rimbalzi**, ma per noi sono troppo **pochi**...

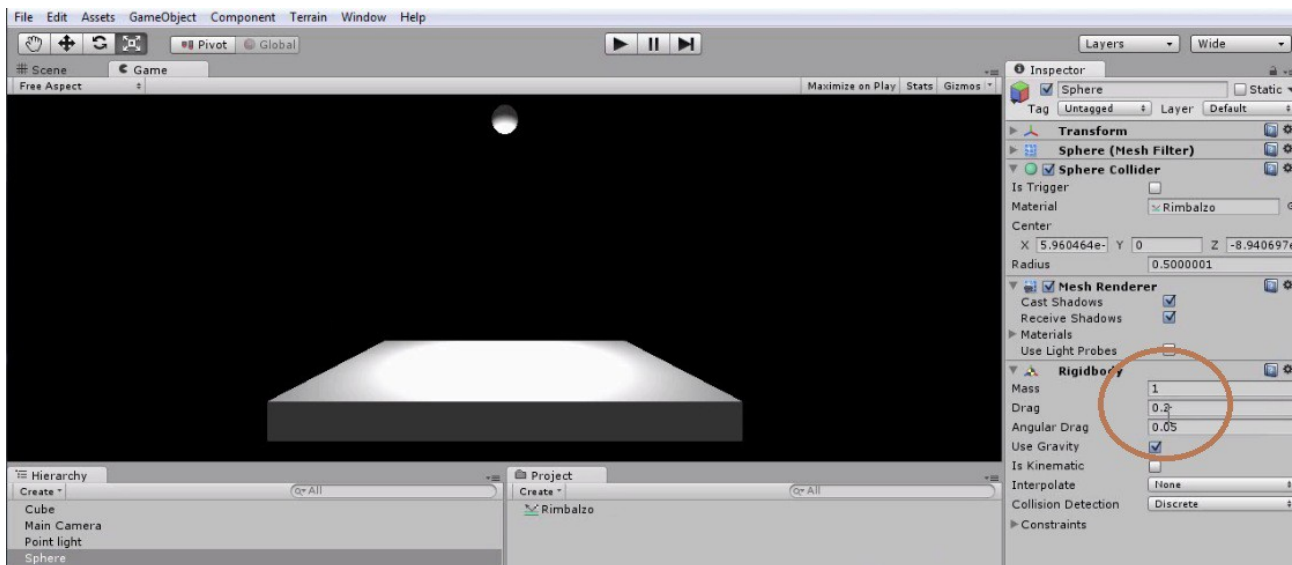
Per risolvere questo problema, cambiamo “*Bounce Combine*”, nell'**Inspector** per il **Material fisico Rimbalzo**, da “*Average*” (**medio**) a “*Maximum*”, quindi avviamo la simulazione con **Play**: vediamo che ora la palla, che ha **Bounciness 1** e **Bounce Combine Maximum**, rimbalza....

beh, ora **rimbalza troppo**! Ad ogni tocco riceve più energia e rimbalza sempre più in alto!



Ricordate il **campo Drag**, visto poco fa, che simula un po' la **forza di attrito** (in questo caso, frena l'oggetto per via dell'attrito con l'aria) e che in un **Rigid Body** è impostato, di default, a 0.0?

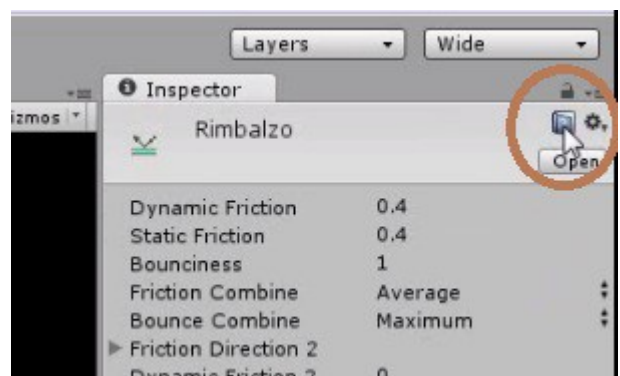
Fermiamo la simulazione, selezioniamo la palla, **impostiamo Drag a 0.1 o 0.2 per il Rigid Body** della palla e avviamo di nuovo la simulazione.



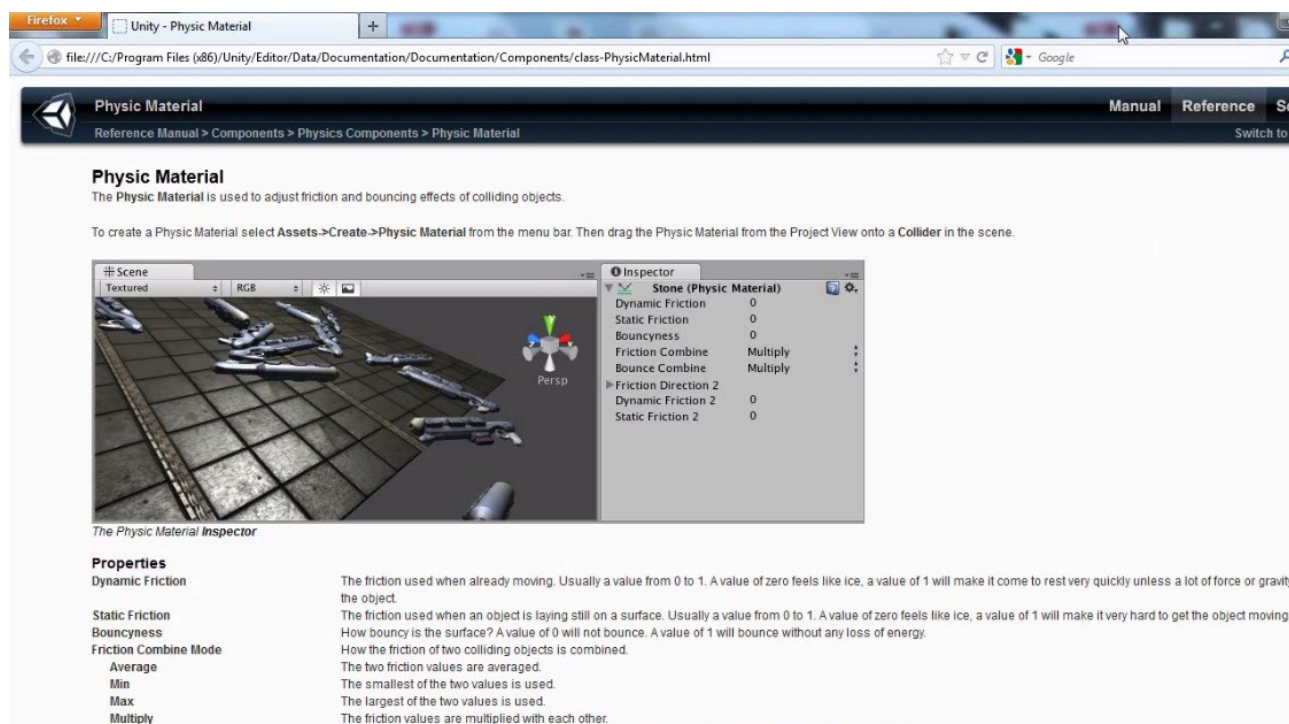
Dobbiamo pensare al **Drag** di un oggetto come al suo **attrito con l'aria**, alla sua “forza frenante”.

Finora abbiamo utilizzato campi dal significato intuitivo, tuttavia molte voci non sono così chiare... come fare per capire **cosa vogliono dire?**

È possibile **accedere all'help in linea** cliccando sull'icona del libro nei vari **Component** nell'*Inspector*, quindi se ad esempio vogliamo saperne di più sui campi di un **Material fisico** selezioniamo tale elemento e, nell'*Inspector*, clicchiamo sull'**icona del libro col punto interrogativo**.



Si aprirà una **pagina web** con il **manuale del programma** e le voci relative proprio a quell'elemento; lo stesso discorso vale per **Rigid Body** e in generale tutte le schede – o *Components* – dell'**Inspector**.



In questa prima parte di questo capitolo abbiamo parlato dei **rapporti tra Colliders, Rigid Body e Materials fisici** per implementare certe caratteristiche fisiche degli oggetti; inoltre, abbiamo visto come accedere all'**help online** per capire il significato di certi elementi e parametri; vediamo ora come **rilevare la pressione dei tasti e modificare al volo le proprietà degli elementi della scena**.

* * *

Continuiamo con la scena definita nella prima parte, quindi abbiamo la pallina che cade e rimbalza sulla pedana un certo numero di volte.

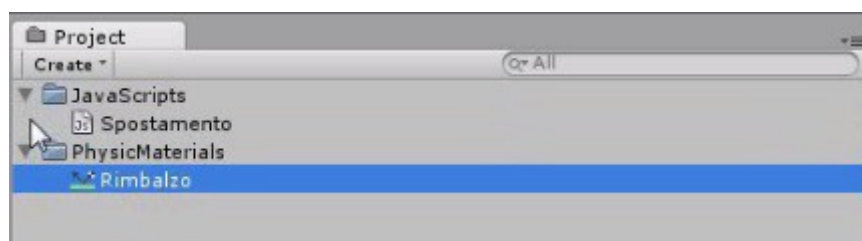
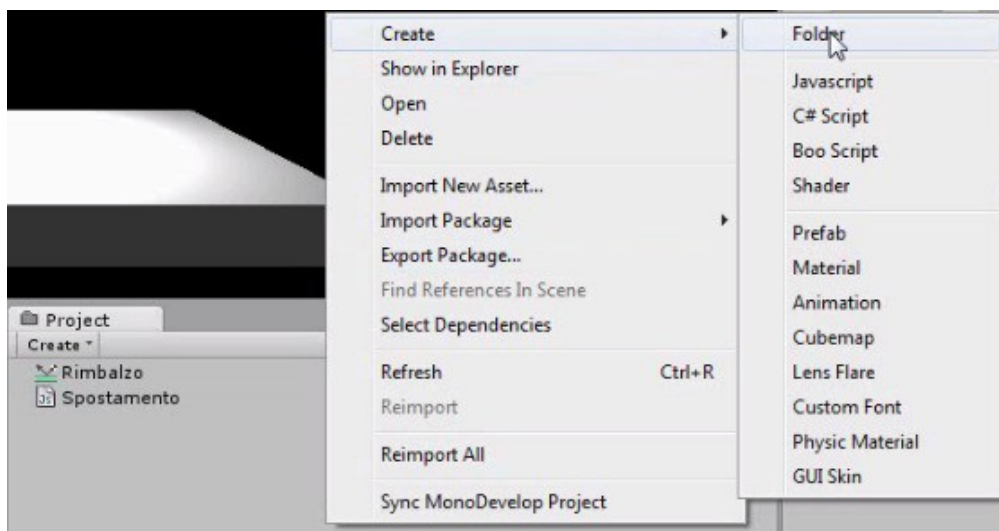
Vogliamo definire **uno script che sposti la pedana**, mantenendola comunque **sul piano orizzontale XZ**, in base alla pressione dei **tasti freccia**.

Vi anticipo che qui lo spostamento sarà un po' rozzo, nel senso che **l'oggetto si muoverà istantaneamente** nella misura data via script; tra qualche capitolo tornerò sull'argomento parlando anche delle *Forces*, le **forze**, ma per il momento limitiamoci a capire **come recuperare la pressione dei tasti della tastiera e modificare le proprietà di un oggetto**.

Dal capitolo precedente sappiamo che uno **Script** va creato nel **Project** e **associato all'elemento** che deve modificare.

Si tratta di operazioni realizzabili anche **a livello di Prefab**, ma in questo progetto abbiamo **solo una pedana da modificare con uno script**, quindi **aggiungiamo un JavaScript** nel **Project**, rinominiamolo in *“Spostamento”* e colleghiamolo all'oggetto **Cube** nella **Gerarchia**, in modo da associarlo a tale oggetto.

Prima di continuare, facciamo un po' di ordine e creiamo due **cartelle**, **“PhysicMaterials”** e **“JavaScripts”** nel **Project**, mettendo **Rimbalzo** nella prima e *Spostamento* nella seconda... in questo **Project** ci sono solo due elementi, quindi l'operazione può sembrare superflua, ma, come detto nel capitolo precedente, cerchiamo di prendere certe **buone abitudini**.



Facciamo **doppio click sullo script** per avviare l'editor associato, nel mio caso **Mono Develop**, per modificare il file.

Ragioniamo un attimo su quello che vogliamo fare, prima di inserire pezzi di codice a destra e a manca: in un certo senso, lo script deve verificare continuamente la **pressione dei tasti** e, se un tasto viene premuto, **effettuare un'operazione di spostamento**, ossia modificare i valori di **Position**.

Per “stare in ascolto”, diciamo così, degli **eventi da tastiera**, ci sono molti modi, ma noi qui utilizzeremo la funzione **Update**.

Come detto nel capitolo precedente, **Update si aggiorna automaticamente a ogni frame di gioco**.

Per verificare la pressione di un tasto utilizzeremo la funzione nativa “GetKey” di Input; riscriviamo quindi la funzione Update (che, di default, è vuota) in questo modo:

```
function Update()  
{  
    if(Input.GetKey("up")) transform.position.z += 0.1;  
}
```

Il **costrutto if** serve a verificare una condizione: “if” significa “se” e quello che abbiamo scritto vuol dire “*se – condizione, scritta tra parentesi – allora fai questo – blocco di istruzioni -*”.

A proposito del blocco di istruzioni: se l'istruzione è solo una, possiamo scriverla direttamente sulla riga dell'if e concludere tutto col punto e virgola; se, invece, al verificarsi della condizione dobbiamo eseguire più istruzioni in blocco, allora dobbiamo definire appunto un blocco, come fatto nel capitolo precedente col *while*, con le parentesi graffe.

Esempio:

```
if(Input.GetKey("up")) {  
    // Seguono le istruzioni, una per riga e chiuse da un ; , oppure  
    possiamo mettere anche degli altri blocchi, tipo altri if o dei  
    while o cicli for, annidati dentro il blocco principale, seguendo  
    quindi la giusta sintassi dei comandi  
} // Al termine, ricordarsi di chiudere i blocchi in ordine con la }
```

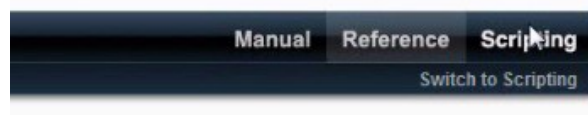
L'**if** controlla una condizione ed esegue il blocco se è verificata; **la condizione può essere multipla**, quindi possiamo dire “se si verifica questo E anche questo”, oppure “se si verifica questo O questo O entrambi”, e così via... c'è anche da dire che esiste un costrutto, l'**else**, che significa “altrimenti” e che ci consente di escludere degli eventi se se ne verificano altri e di dare delle priorità nella verifica dei tasti premuti... ho deciso di **fare un capitolo extra (il prossimo), chiamato “5-A”**, a riguardo.

Prima di proseguire, una nota: il **+=** è un modo per dire “*somma questo valore all'originale*”; se ricordate, nel capitolo precedente ho scritto “*tempoRimanente = tempoRimanente – 1;*”, ebbene, possiamo accorciare l'istruzione scrivendo “*tempoRimanente -= 1*”, che significa “sottrai il valore a destra, 1, a quello a sinistra, e sovrascrivi quello a sinistra col risultato così calcolato”.

Per dire a **Unity** di modificare i campi *Position*, aggiungendo o sottraendo dei valori, **dobbiamo sapere come si chiamano**, a livello di codice, questi campi.

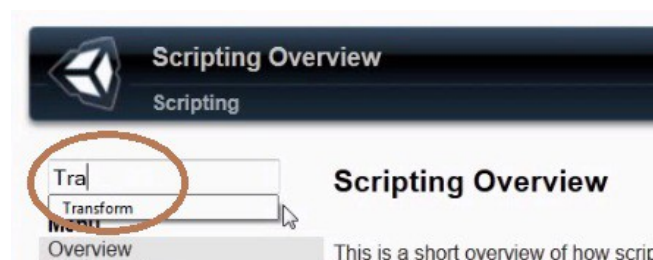
Per scoprirlo, andiamo nell'**Inspector** e clicchiamo sull'icona del libro col punto interrogativo nella sezione **Transform**.

Si aprirà l'**help online**, come visto nella prima parte di questo tutorial, sulla voce **Transform**, ma a questo punto clicchiamo, in questa pagina, sul pulsante “**SCRIPTING**” in alto a destra.

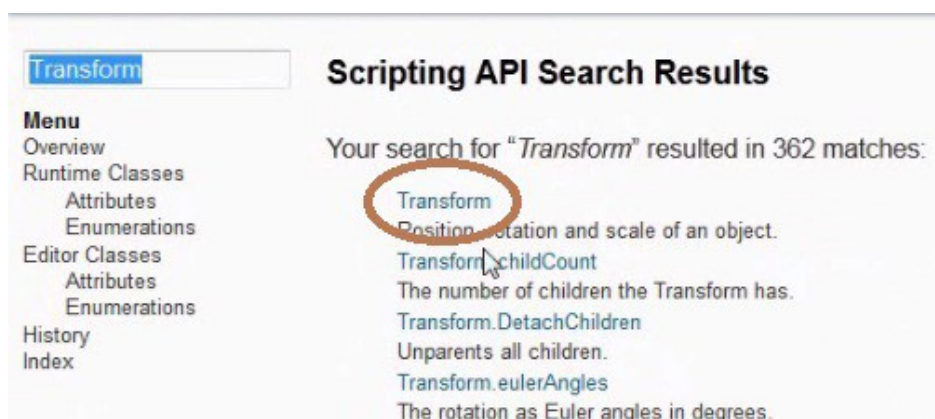


Questa è la **sezione Scripting del manuale**: qui, partendo da una voce, possiamo conoscere variabili, classi, operatori e una gran quantità di esempi di codice per i nostri progetti!

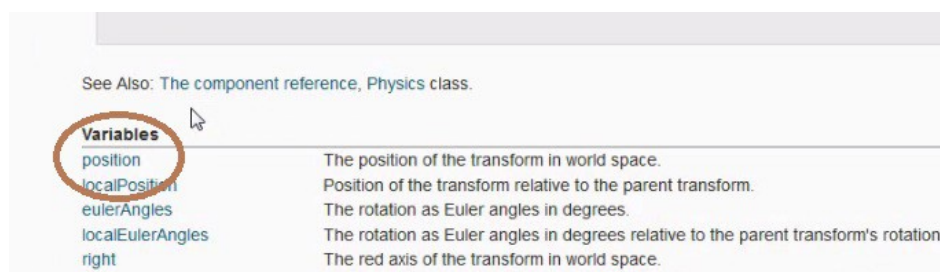
Dal momento che vogliamo saperne di più sui campi *Transform*, in particolare *Position*, scriviamo appunto **Transform** nella **casella di ricerca** che apparirà.



Scegliamo il primo risultato, **Transform**.



In questa pagina, scegliamo la **variabile position** (come potete vedere, molte voci sono dei link che rimandano ad altre pagine di approfondimento).



Questa pagina ci dice che un “**position**”, nel codice, non è un solo numero, ma **un insieme di tre numeri**: in alto troviamo “*var position : Vector3*”, quindi è di tipo “*Vector3*”, che qui significa – detto in parole povere, poverissime – **un insieme di tre numeri** (e un paio di **funzioni** native che possono essere effettuate su tale insieme).

Transform.position

var position : Vector3

Description

The position of the transform in world space.

```
// Move the object to (0, 0, 0)
transform.position = Vector3(0, 0, 0);

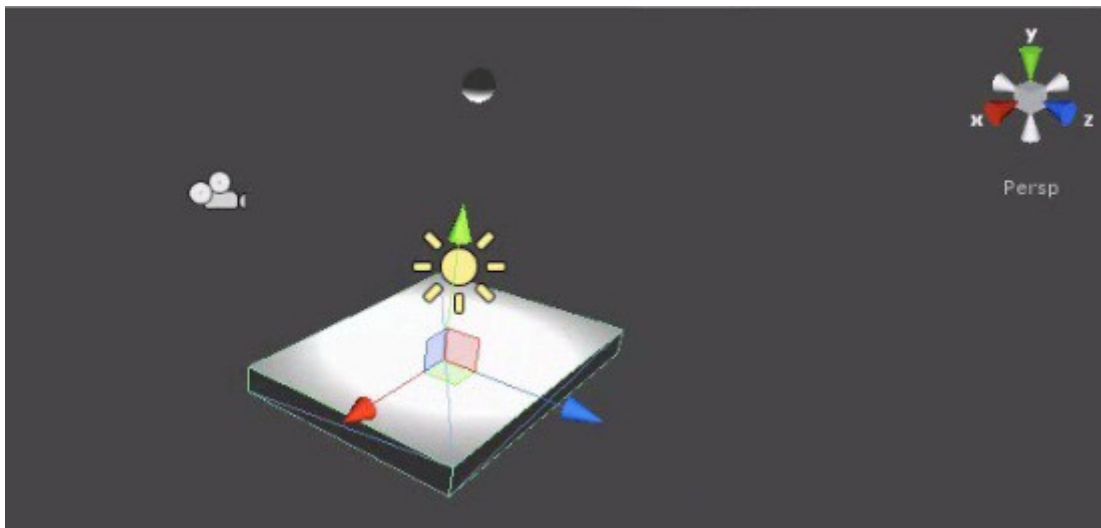
// Print the x component of the position to the Console
print(transform.position.x);
```

Le **coordinate X, Y e Z** si trovano quindi in quell'insieme; in particolare, **il valore X è il primo numero**, Y il secondo e Z il terzo... ma come fare per **accedere ai singoli numeri di questo Vector3**?

La soluzione ce la suggerisce il secondo esempio di codice nella pagina: **transform.position.x** .

Se ci interessa la **coordinata x di un oggetto**, quindi, dobbiamo scrivere questa stringa per recuperarla.

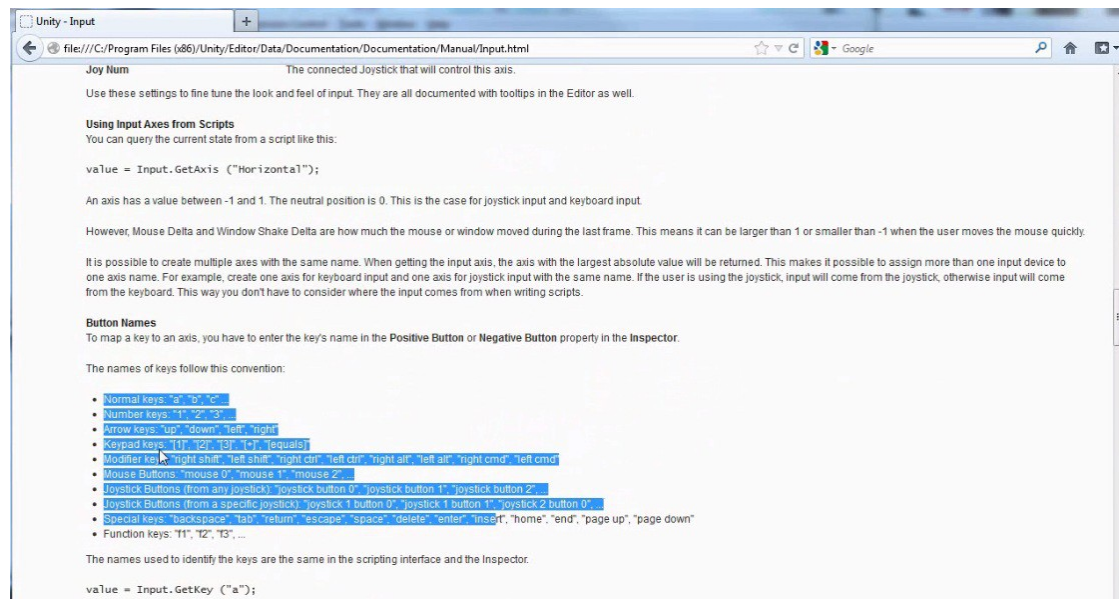
Ovviamente dobbiamo capire **su quali assi spostare l'oggetto**... diamo un'occhiata alla pedana, nella scena 3D: guardando **gli assi del manipolatore di traslazione**, vediamo che l'asse del movimento orizzontale è quello X, mentre quello avanti-dietro rispetto alla telecamera è Z.



Torniamo quindi a **MonoDevelop** e scriviamo il nostro **script**; in particolare, ***all'interno della funzione Update***, scriviamo:

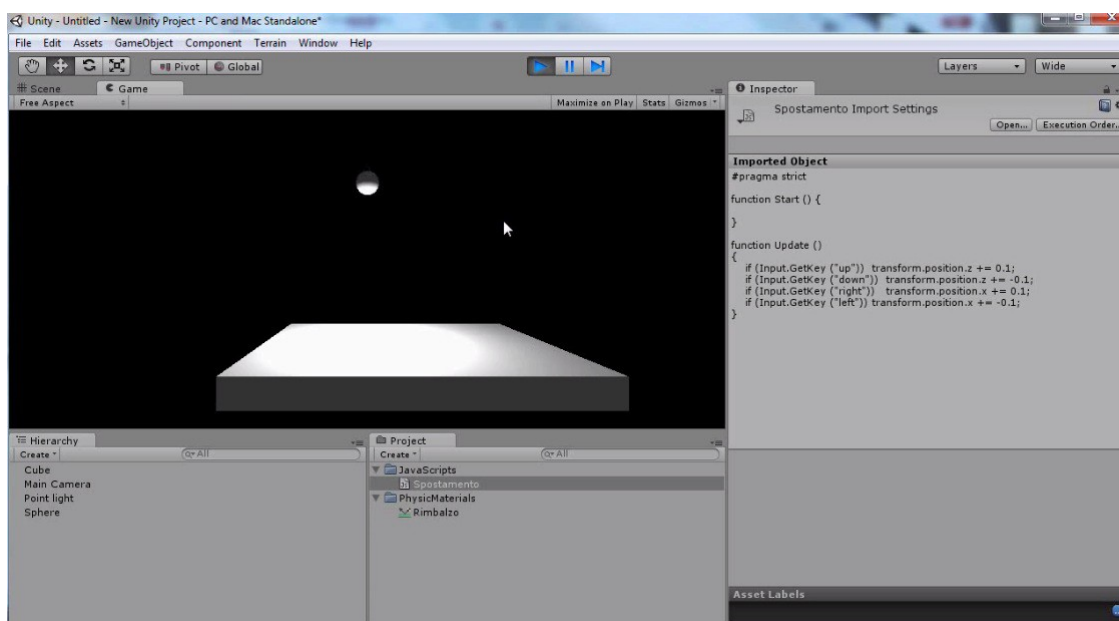
```
function Update () {  
    if (Input.GetKey ("up")) transform.position.z += 0.1;  
    if (Input.GetKey ("down")) transform.position.z += -0.1;  
    if (Input.GetKey ("right")) transform.position.x += 0.1;  
    if (Input.GetKey ("left")) transform.position.x += -0.1;  
}
```


Una nota sui **codici** che ho utilizzato per indicare i **tasti** premuti, ossia **up**, **down**, eccetera: sono **parole chiave di Unity** e trovate tutto nella sezione scripting del manuale, vista poco fa; in particolare, potete iniziare la vostra ricerca con *Input.GetKey*, la funzione che controlla **quale tasto viene premuto sulla tastiera** e che ho utilizzato all'inizio dello script, e cliccare in quella pagina sul link **“Input Manager”**, che contiene una lista di parole chiave associate ai tasti.



Salviamo lo script per rendere effettive le modifiche e torniamo a **Unity**.

Guardate ora che succede se avvio il gioco con **Play: premendo un tasto freccia, la pedana si sposta!**



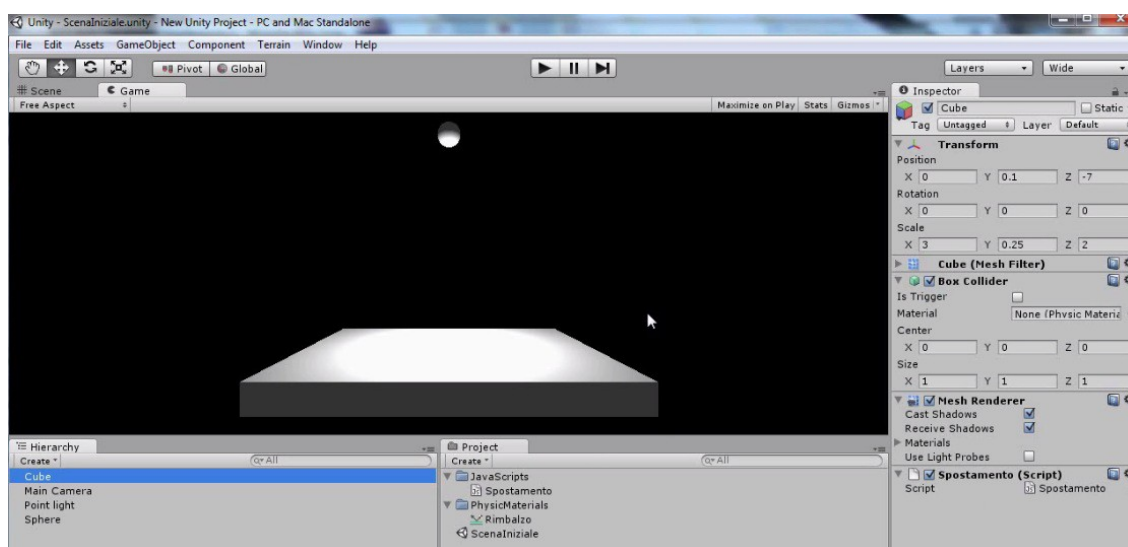
Si sposta anche se **premiamo due tasti freccia contemporaneamente**, perché – per come l'ho scritto – il blocco **Update** verifica ad ogni istante se i singoli tasti vengono premuti e, se sì, effettua le operazioni; non ci sono quindi **comandi di priorità o di esclusione**, ad esempio “se premo frecciasu fai solo quello e ignora gli altri”, per i quali servirebbero ad esempio l'else o altre condizioni... chi sa qualcosa di **programmazione** può muoversi già da solo e personalizzare questo script, mentre per gli altri questo può rappresentare un problema... lo risolveremo nel prossimo capitolo, chiamato “**5A**” perché non è la sesta puntata del corso introduttivo, ma un **breve capitolo extra** per parlare proprio di **if-else** e delle **condizioni in AND e in OR**, rivolto esclusivamente a chi non sa nulla di programmazione, mentre i programmatori possono saltarlo a piè pari.

* * *

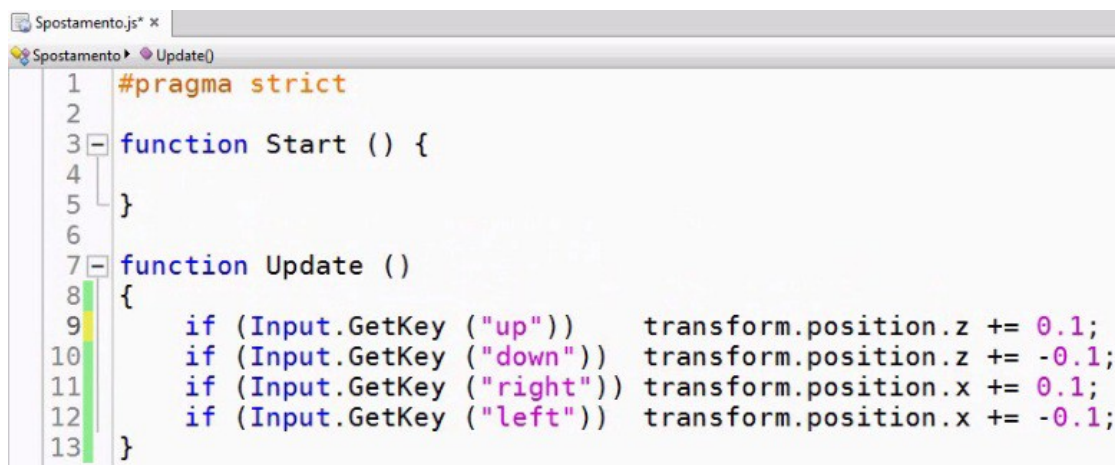
Le basi di Unity 3D 3.x e 4.x – 5A: extra sulla programmazione (IF-ELSE)

Questo capitolo del corso di base su Unity 3D si chiama “**5A**” perché non si tratta, a livello di argomenti propri di **Unity**, del sesto tutorial, quanto di un **capitolo integrativo** su certi **elementi di programmazione**, per chi inizia proprio da zero.

Nel **quinto capitolo** abbiamo trattato infatti il **costrutto if**, un costrutto **condizionale**, che serve cioè a verificare se certe condizioni, espresse tra parentesi, vengono verificate o no; l'abbiamo trattato in maniera elementare, mostrando un'istruzione **if** per ciascun tasto da controllare e con una sola condizione da verificare per volta, quindi qui cercheremo di approfondire l'argomento.



Riprendiamo l'esempio visto nel capitolo precedente, dove avevamo **la funzione Update** definita da quattro **if**, ciascuno dei quali verifica la **pressione di un tasto** ed effettua, in caso affermativo, la relativa operazione.



```
1 #pragma strict
2
3 function Start () {
4
5 }
6
7 function Update ()
8 {
9     if (Input.GetKey ("up"))    transform.position.z += 0.1;
10    if (Input.GetKey ("down"))  transform.position.z += -0.1;
11    if (Input.GetKey ("right")) transform.position.x += 0.1;
12    if (Input.GetKey ("left"))  transform.position.x += -0.1;
13 }
```

Premessa: molte delle istruzioni che mostrerò qui potrebbero essere **scritte in molti modi differenti** (ed in effetti ne riscriveremo parecchie nel corso del tutorial), ma le mostrerò per **scopi puramente didattici**, per parlare dei costrutti e delle parole chiave... quindi **guardate la Luna, non il dito!**

Supponiamo di voler aggiungere la seguente regola allo script: se l'utente preme **contemporaneamente** la barra spazio e la freccia in avanti, anziché spostarlo sull'asse Z, spostiamo l'oggetto verticalmente, sull'asse Y.

Per dire al costrutto *if* di **verificare che due o più condizioni siano verificate** (e TUTTE devono essere vere), dobbiamo usare un **operatore logico**... senza scendere in dettaglio, diciamo che vogliamo usare un **controllo AND, la congiunzione “e”** in Inglese, qui rappresentata da due **&**.

Se tutto questo vi sembra confusionario o astratto, **guardate l'esempio di codice**: sono sicuro che, appena vedrete la scritta, vi apparirà chiarissimo.

In particolare, ***all'interno della funzione Update***, dopo i quattro *if* già presenti (immagine sopra), scriviamo:

```
if (Input.GetKey ("up") && Input.GetKey ("space")) transform.position.y +=
0.1;
```

Questa riga vuol dire appunto *“se vengono premuti il tasto freccia in su E la barra spazio, allora sposta sull'asse Y di 0.1”*.

L'**AND** può essere esteso a un **numero arbitrario di condizioni**, basta **concatenare** alla condizione altri **&&** e condizioni da verificare; un esempio di condizione è “se il giocatore raggiunge la base e ha in mano la bandiera nemica e la sua bandiera è nella propria base” (quindi tre condizioni) “assegna il punto alla squadra”, e così via.

L'**operatore logico OR** richiede invece che **ALMENO una delle condizioni espresse sia vera**.

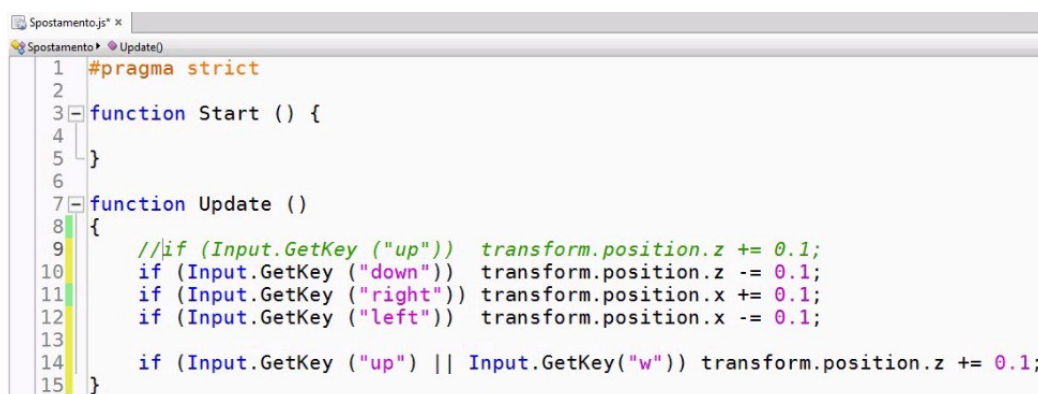
Diciamo quindi che, per spostare in avanti la pedana, l'utente può premere la sola freccia in su **OPPURE** il tasto w (per simulare lo schema di controllo WASD).

Da notare che la condizione è verificata anche se l'utente preme contemporaneamente sia la freccia in su che la w, perché l'OR dice **“ALMENO una condizione”**, su un numero arbitrario di condizioni.

L'operatore **OR** va scritto con due caratteri |, la stanghetta verticale.

L'istruzione diventa quindi:

```
if (Input.GetKey ("up") || Input.GetKey("w")) transform.position.z += 0.1;
```



```
1 #pragma strict
2
3 function Start () {
4
5 }
6
7 function Update ()
8 {
9     //if (Input.GetKey ("up")) transform.position.z += 0.1;
10    if (Input.GetKey ("down")) transform.position.z -= 0.1;
11    if (Input.GetKey ("right")) transform.position.x += 0.1;
12    if (Input.GetKey ("left")) transform.position.x -= 0.1;
13
14    if (Input.GetKey ("up") || Input.GetKey("w")) transform.position.z += 0.1;
15 }
```

*[NOTA: abituatevi a **separare le condizioni**, messe in **AND** o in **OR**, mediante le **parentesi tonde**, che vi aiutano proprio perché isolano blocchi di condizioni e rendono **più facile sia la scrittura che l'interpretazione delle stesse**.]*

Ricordatevi sempre di **salvare lo script**, quando state per fare delle prove.

Dobbiamo parlare anche dell'**else**, per implementare il costrutto **“if-else”**, cioè “se la condizione è verificata fai questo, **ALTRIMENTI** fai quest'altro.

È possibile fare anche degli **“else if”**, per **verificare condizioni esclusive in cascata...** ma andiamo con ordine, con l'**if-else semplice**, vedendo subito un **esempio**.

Sempre **all'interno della funzione Update**, scriviamo:

```
if (Input.GetKey("v") && renderer.enabled == true)
{
    // Blocco dell'if, se la condizione è verificata
    renderer.enabled = false;
}
else
{
    // Se invece l'utente preme enter ma renderer era false, quindi l'if
    non era verificato
    renderer.enabled = true;
}
```

Qui notiamo la presenza del **==** e facciamo anche la conoscenza di un nuovo elemento, **renderer**.

Per prima cosa parliamo dell'**==**: quando vogliamo **verificare che un valore sia ESATTAMENTE uguale a un altro**, nell'**if** non possiamo scrivere **“A = B”**, perché **il solo = è l'operatore che serve ad ASSEGNARE il valore di B ad A**; per **CHIEDERE**, anziché **ASSEGNARE**, se A ha lo stesso valore di B, dobbiamo scrivere **A == B**.

La risposta alla domanda è un **booleano**: **true**, cioè vero, se i due valori sono uguali, **false** altrimenti.

L'altro elemento è **renderer.enabled**.

“renderer” non è altro che la checkbox **“Mesh Renderer”**, nell'**Inspector** dell'oggetto che utilizza lo **script**.

Trattandosi di una checkbox, ha una proprietà: **enabled**, cioè selezionata, attiva.

La scritta **“renderer.enabled”** indica quindi lo **stato della checkbox**, che può essere o selezionata o deselezionata... e, in Unity, se la checkbox è selezionata, si dice che è a **TRUE**, cioè è vera, attiva, altrimenti è **FALSE**, falsa, non verificata, spenta.

Il senso di **“if (renderer.enabled == true)”** è quindi: **“se il renderer è attivo”**, ovvero se la casella **Mesh Renderer** dell'oggetto è selezionata.

Nel nostro script, questa condizione è messa in **AND** con la pressione del tasto **v**: se viene premuto il tasto **v AND la casella è selezionata**, allora esegui l'istruzione che segue.

L'**else** serve a dire: se invece la condizione appena descritta (*GetKey(“v”) AND renderer.enabled==true*) **NON** si avvera, esegui quanto c'è scritto in questo blocco qui di seguito.

Mandando in esecuzione il gioco (dopo aver salvato lo script), notiamo però un problema: se premiamo il tasto **v** (per “visibilità”), la pedana sembra **lampeggiare**, senza però diventare **invisibile** come previsto... perché?

Notate **quant'è subdolo l'if** all'interno di questa particolare funzione, **Update**, così come l'abbiamo scritto: **ad ogni esecuzione di update**, cioè ad ogni frame, chiede: **“è stato premuto il tasto v mentre renderer era attivo?”** - ma in realtà, se noi **NON stiamo premendo “v”**, anzi siamo proprio fermi, l'*if* non finisce di porre la domanda: appena chiede “è stato premuto il tasto v?”, **risponde “no” e salta il resto del controllo** (si dice che la prima condizione in **AND**, non verificata, **sta cortocircuitando tutto l'if**, uscendo prima del dovuto perché già all'inizio non è valida); a quel punto entra in gioco l'**else**: siccome l'*if* NON è valido, **esegue l'else** associato, e... rende attivo il **renderer**, visualizzando la pedana!

Una soluzione consiste nel **controllare anche l'else**.

Cambiamo il **blocco else** così:

```
else if (Input.GetKey("v") && renderer.enabled == false)
{
    // Se invece l'utente preme enter ma renderer era false, quindi l'if
    non era verificato
    renderer.enabled = true;
}
```

Adesso (dopo aver salvato lo script), se nel gioco premiamo **v** e, se il **renderer** era attivo, il primo **if** lo metterà a **false**.

Se non premiamo nulla (o premiamo qualche altro tasto, ma non **v**), questi due blocchi **if-else** NON vengono attivati, perché entrambi hanno come condizione di attivazione la **pressione di v** più un'altra condizione.

Se premiamo **v** con **renderer enabled a false**, l'**if** chiede: “è stato premuto **v** (... sì...) e l'*enabled* era a *true*?” - no! Quindi vediamo l'**else if**: “è stato premuto **v** (... sì...) e l'*enabled* era a *false*?” - sì! **Esegui l'istruzione nel blocco else-if**.

NOTA a margine: il **Mesh Renderer** si occupa **solo di visualizzare la geometria** della pedana, non di rimuovere la geometria dalla scena o di togliere le proprietà di collisione; **la geometria diventa quindi invisibile, ma c'è** e le sue proprietà fisiche restano.

Una considerazione importante: questo problema, molto subdolo, **si verifica perché Update viene invocato automaticamente ad ogni frame da Unity** e, quindi verifica tutte le istruzioni presenti nel blocco.

C'è un modo per “fregarlo”... anzi, più di uno e, tra l'altro, uno di questi consente **una buona ottimizzazione**: vediamoli!

Il blocco *if* del tasto **v** e dell'**enabled** va messo **in una funzione a parte** che viene chiamata SOLO quando viene premuto il tasto **v**.

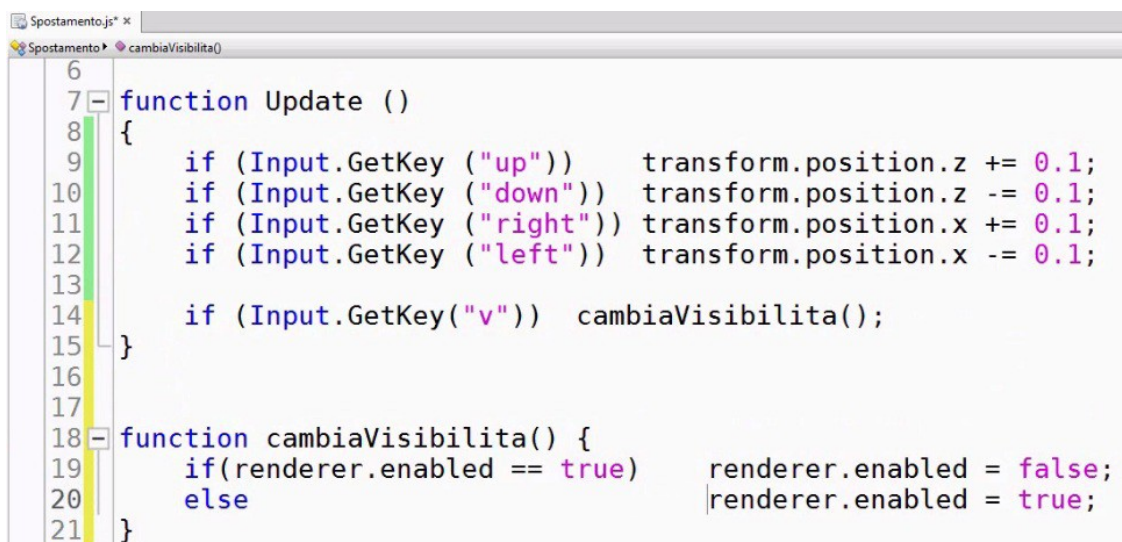
Vi scrivo il codice: in **Update**, cancelliamo il blocco **if-else** e scriviamo solo:

```
if (Input.GetKey("v")) cambiaVisibilita();
```

Dopodiché, nello spazio globale **fuori da qualsiasi funzione**, scriviamo:

```
function cambiaVisibilita() {  
    if(renderer.enabled == true) renderer.enabled = false;  
    else renderer.enabled = true;  
}
```

Ecco quindi come si presentano **Update** e **cambiaVisibilita**, adesso:



```
6  
7 function Update ()  
8 {  
9     if (Input.GetKey ("up"))    transform.position.z += 0.1;  
10    if (Input.GetKey ("down"))  transform.position.z -= 0.1;  
11    if (Input.GetKey ("right")) transform.position.x += 0.1;  
12    if (Input.GetKey ("left"))  transform.position.x -= 0.1;  
13  
14    if (Input.GetKey("v"))  cambiaVisibilita();  
15 }  
16  
17  
18 function cambiaVisibilita() {  
19     if(renderer.enabled == true)    renderer.enabled = false;  
20     else                             renderer.enabled = true;  
21 }
```

Ecco, questa volta possiamo usare il semplice **if-else** senza temere problemi... ma attenzione... perché non abbiamo finito! **Possiamo ottimizzare ancora questa funzione!**

Riscriviamo la **funzione** in questo modo:

```
function cambiaVisibilita() {  
    renderer.enabled = !renderer.enabled;  
}
```

L'**operatore !** è un altro operatore logico, in particolare il **NOT**, la **negazione**, che se messo di fronte a un qualcosa che dovrebbe restituire **true** restituisce **false**, e viceversa.

Nella funzione, stiamo scrivendo: **“a renderer.enabled assegna... il contrario di renderer.enabled”**. Se infatti la casella era selezionata (**true**), assegniamo **NOT true**, cioè **false** e viceversa.

Ma non è finita qui...

A ben vedere, ora che vi ho fatto conoscere il NOT, **questa funzione è superflua**. Eliminiamola, **cancelliamo tutto il blocco**.

Torniamo nell'**update** e **sostituiamo** la riga:

```
if (Input.GetKey("v")) cambiaVisibilita();
```

con:

```
if(Input.GetKey("v")) renderer.enabled = !renderer.enabled;
```

Adesso quell'**if** viene chiamato quando si preme **v**, ignorando lo stato di **renderer** in quel momento, quindi evitiamo errori subdoli come prima; in più, ogni volta che **v** viene premuto, **inverte lo stato di *renderer.enabled*, sia esso *true* o *false***.

L'ottimizzazione del codice è un punto importante nella **programmazione dei videogiochi** perché consente di risparmiare calcoli, quindi tempo e risorse... c'è un bel passaggio dal blocco **if-else** o dall'uso della funzione esterna a questa singola, semplicissima riga di codice, in effetti.

L'importante, ovviamente, è **verificare la logica delle istruzioni**, per non commettere certi errori, come prima. Se non avete mai programmato, tutto ciò può procurare una gran confusione, ma **è solo questione di pratica**.

* * *

Le basi di Unity 3D 3.x e 4.x – 6: istanziazione Game Objects mediante script

In questo **sesto capitolo “ufficiale”** sulle **basi di Unity 3D**, vedremo come inserire al volo degli **oggetti in una scena “istanziandoli”** (cioè creando delle copie da un Prefab originale) mediante uno **script in JavaScript**.

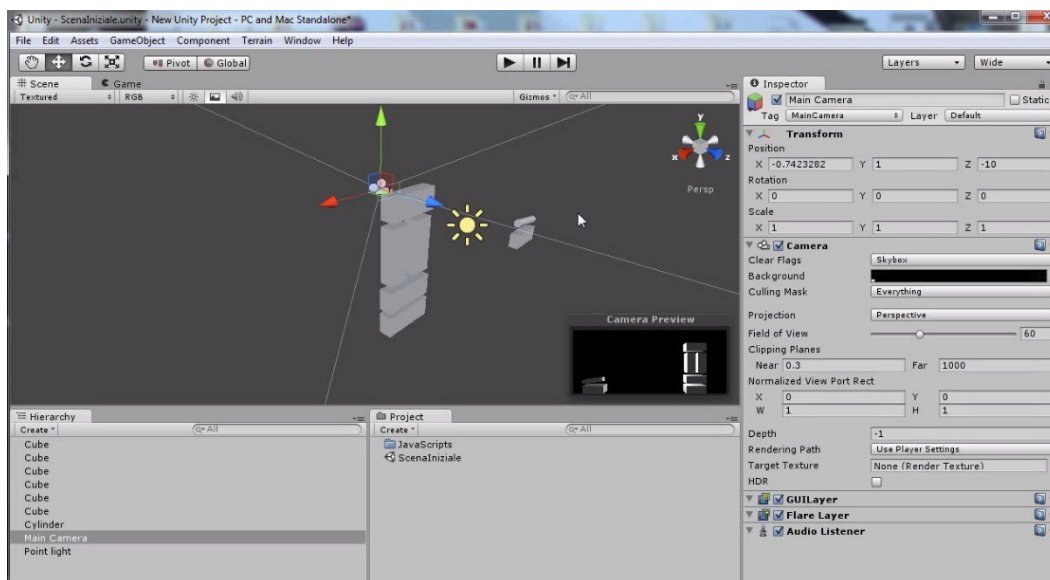
Sarà un modo per riprendere alcuni argomenti come i **Prefab**, i **Rigid Body** e le **coordinate locali** di un elemento.

Una piccola premessa: il **codice** è, per natura, **sequenziale**, nel senso che per descriverlo dovrò dirvi le istruzioni una per una, tuttavia a volte ci sono istruzioni che vengono completate o che assumono un senso **solo più avanti nel codice**, quindi all'inizio la descrizione dello **script** e anche delle **funzioni del programma** può sembrare **confusionaria** e addirittura frustrante (è come se vi dicessi di mettere dei mattoni in certi punti senza spiegarvi il perché o mostrarvi il disegno di quello che vogliamo realizzare), però **vi invito a non gettare la spugna**, a mettere i mattoni e aspettare il completamento del tutorial, perché a quel punto si può osservare il risultato in azione, si ha il **“quadro completo”** della composizione e tutti gli elementi, che presi singolarmente non dicevano molto, appariranno chiarissimi. Questo vale, in generale, **anche per i tutorial futuri**.

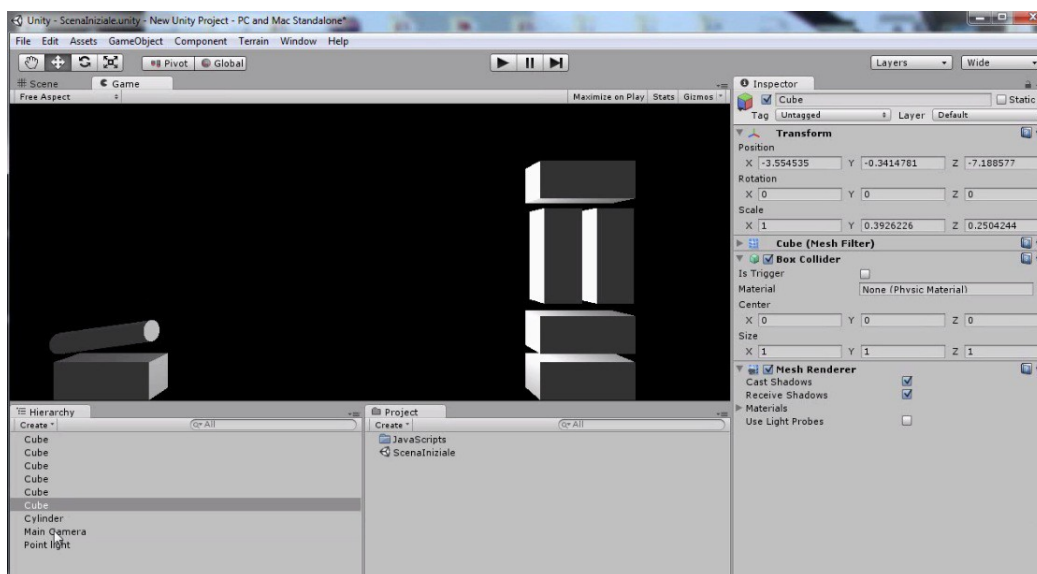
La **scena** che utilizzeremo (immagine nella pagina seguente) è molto **semplice**: è composta da un alcuni **cubi schiacciati lungo alcuni assi**, in modo da ottenere la forma di **parallelepipedo** (uno rappresenta la base di un “**cannoncino**”, se vogliamo, gli altri **pilastri ed elementi da abbattere**), più un **cilindro** che rappresenta, in maniera decisamente stilizzata, un **cannoncino**... scusate se non ho importato dei **modelli degni di questo nome**, fatti magari con **Blender**, ma voglio utilizzare degli oggetti che potete inserire facilmente per conto vostro nella scena, in modo da riprodurre quello che faccio nel **tutorial**: dovete solo inserire dei **Game Objects Cube** e **Cylinder** e

Raccolta di tutorial “*UNITY 3D tutorials – Volume 1*” dal sito *francescomilanese.com* --- © 2015

schiacciarli e ruotarli con gli strumenti visti nei capitoli precedenti fino a ottenere una configurazione simile a questa.



Nell'immagine sotto: la scena vista dalla Telecamera virtuale; è importante che tutti gli elementi siano allineati lungo quello che, in questo caso, è l'asse X (vedere immagine precedente).

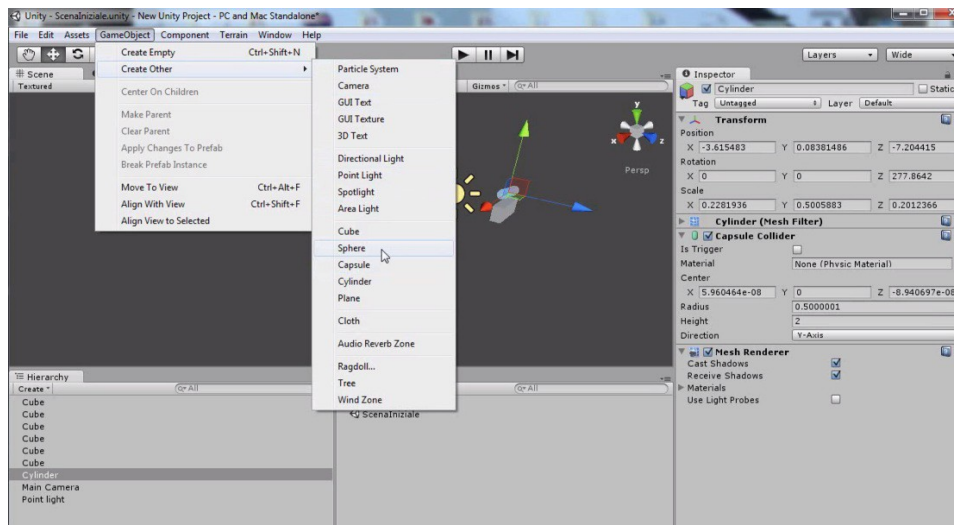


Di base, queste primitive **non hanno sistemi fisici associati** né altri elementi: partiamo proprio da questo punto.

Vogliamo **far sparare delle sfere al cilindro**, nella direzione data dall'oggetto (tra parentesi: qui non gestirò la **rotazione del cannoncino** perché bisognerebbe trattare anche lo **spostamento del Pivot Point**, il **perno** intorno al quale ruotare l'elemento, che di default coincide col centro dello stesso... ne parleremo un'altra volta).

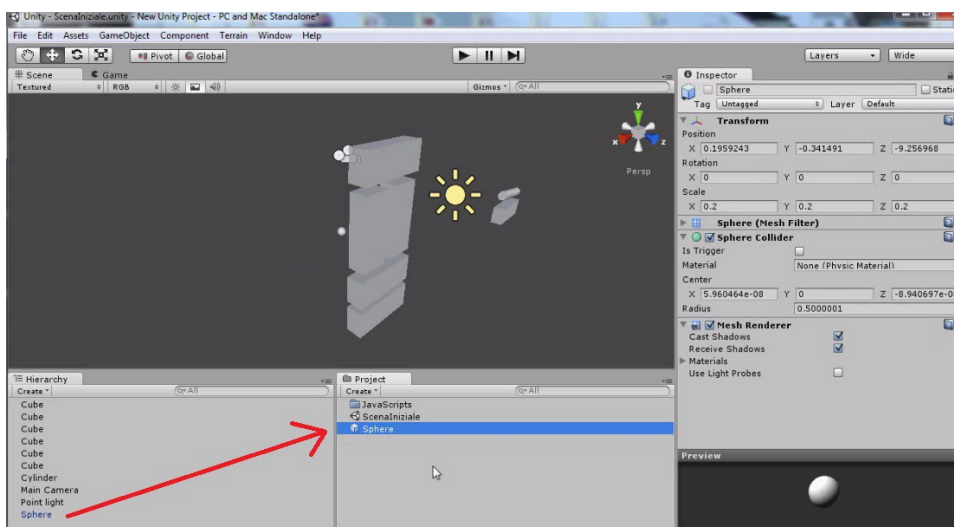
Per prima cosa ci serve, quindi, **un prototipo di proiettile**.

Inseriamo **una sfera** nella scena (immagine seguente: creazione di una mesh sferica)...



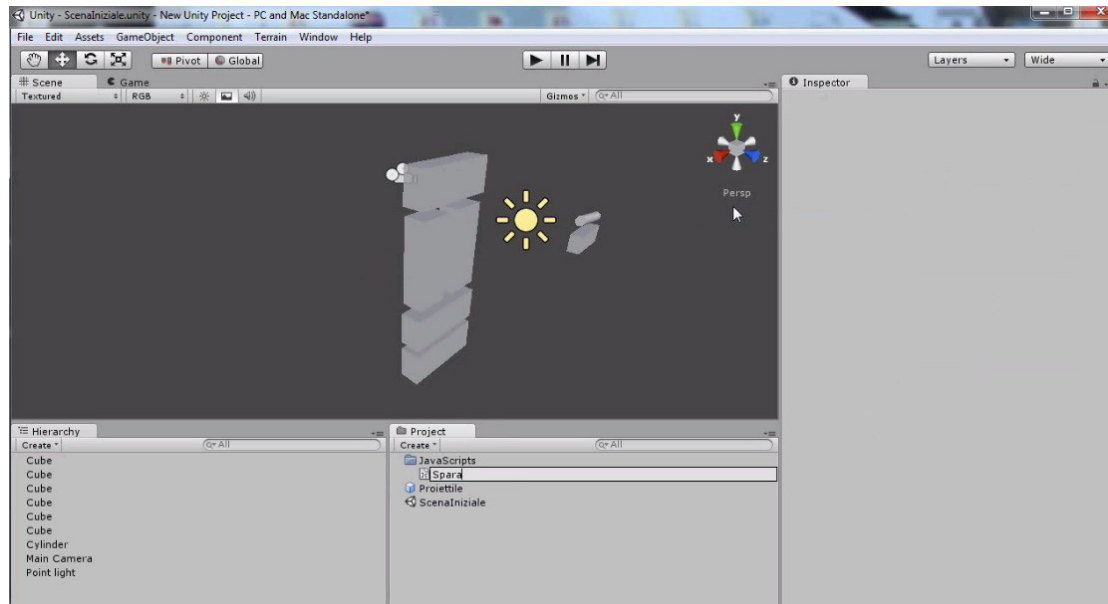
quindi trasciniamo tale elemento dalla **Gerarchia** al pannello **Project** (immagine seguente): si tratta di una **scorciatoia per creare un Prefab** di un elemento direttamente da una sua istanza, senza dover fare **Create – Prefab** nel **Project**.

Adesso la sfera nella **Gerarchia** è scritta in **blu** proprio perché **istanza di quel Prefab**.



Possiamo **cancellare la sfera dalla Gerarchia** e, quindi, dalla scena corrente: **adesso non ci serve un'istanza del Prefab** nella scena, non più.

Creiamo quindi uno **script JavaScript** e chiamiamolo, ad esempio, **“spara”**, quindi apriamo l'editor (nel mio caso, **MonoDevelop**) per definire il **codice**.



All'interno di **Update**, scriviamo un'istruzione per **chiamare una funzione** – che definiremo tra pochissimo – quando l'utente preme, ad esempio, la **barra spazio**:

```
if(Input.GetKey("space")) spara();
```

Fuori da qualsiasi blocco funzione, nello **script**, scriviamo quindi:

```
function spara(){  
    // Crea un'istanza partendo da un prototipo e fornendogli posizione  
    e orientamento di chi la produce (il cilindro-cannone)  
    var istanza : Transform;  
    istanza = Instantiate(prototipoProiettile, transform.position,  
        transform.rotation);  
    // Aggiunge una forza all'istanza col vettore dato da direzione *  
    energia  
    istanza.rigidbody.AddForce(istanza.transform.forward * energia);  
}
```

Analizziamo il codice appena scritto: per prima cosa, stiamo **creando una variabile** di un tipo particolare, nativo di **Unity 3D**, che è il **tipo Transform**.

Questa **variabile**, che ha nome **istanza**, è definita, ma **non è inizializzata**, nel senso che non le stiamo assegnando ancora alcun elemento.

Il tipo **Transform** è un tipo **generico** (potremmo dire “astratto”), che ci consente di legare alla variabile elementi di varia natura; noi, ad esempio, **le legheremo un Prefab** (quello della **sfera**) che, tra l'altro, ha un **componente Rigid Body**... quindi, definiamo **istanza** come elemento di tipo **Transform**.

Subito dopo, **inizializziamo DAVVERO questa variabile**, assegnandole il risultato di una **funzione nativa di Unity, Istantiate**, che sostanzialmente **crea un oggetto** partendo da alcune specifiche, che vanno messe in ordine tra le parentesi tonde della chiamata alla **funzione** (sono i “**parametri**” della funzione).

In particolare, la **firma** (ovvero, l'insieme dei parametri da passare alla funzione) di **Istantiate** è: ***oggetto prototipo, vettore posizione, vettore orientamento***.

In pratica, quando si chiama **Istantiate** bisogna fornirgli una **variabile** che rappresenta **l'oggetto da riprodurre** – variabile che qui chiamiamo **prototipoProiettile** – più delle coordinate di **posizione** un **orientamento** iniziali.

Per questi ultimi due parametri, utilizziamo **Position** e **Rotation dell'oggetto genitore**, che **spara il proiettile** (e che possiede questo **script**): ecco il senso di **transform.position** e **transform.rotation**, i valori posizione e orientamento **dell'oggetto transform**.

Vi ho fatto scrivere **prototipoProiettile**, ma questa variabile **non è definita** da nessuna parte nel codice... beh, è il momento di farlo: **all'inizio dello script**, sotto la riga *pragma* e fuori da blocchi di funzioni, scriviamo:

```
var prototipoProiettile : Transform;
```

quindi **un altro elemento di tipo Transform**.

Non inizializziamo però questo oggetto... vedrete presto perché.

Mentre ci siamo, in questo punto **aggiungiamo anche questa istruzione:**

```
var energia : int = 1000;
```

che, neanche a dirlo, è un **valore dell'energia del proiettile** al momento dell'uscita.

Torniamo quindi nella **funzione spara**: sotto l'inizializzazione dell'istanza, c'è l'istruzione:

```
istanza.rigidbody.AddForce(istanza.transform.forward * energia);
```

Ormai abbiamo capito che quando scriviamo **.qualcosa** dopo un **oggetto**, ci stiamo riferendo o a una sua **funzione** (se ci sono le **parentesi tonde**, come nel caso di **.AddForce**), oppure a un suo **campo o attributo**, come nel caso di **.igidbody**... ecco, questo **.igidbody**, in particolare, indica a Unity di accedere al **Componente Rigid Body dell'oggetto**.

I **Components** sono quelli definiti nell'**Inspector**, ed infatti scrivendo ad esempio **istanza.renderer** possiamo accedere al **componente Mesh Render dell'oggetto**, come visto nel capitolo 5-A, giusto per fare un esempio.

Noi non abbiamo ancora definito né l'oggetto **prototipoProiettile** né tanto meno un **componente Rigid Body per tale oggetto**, però stiamo già scrivendo queste istruzioni per modificarli!

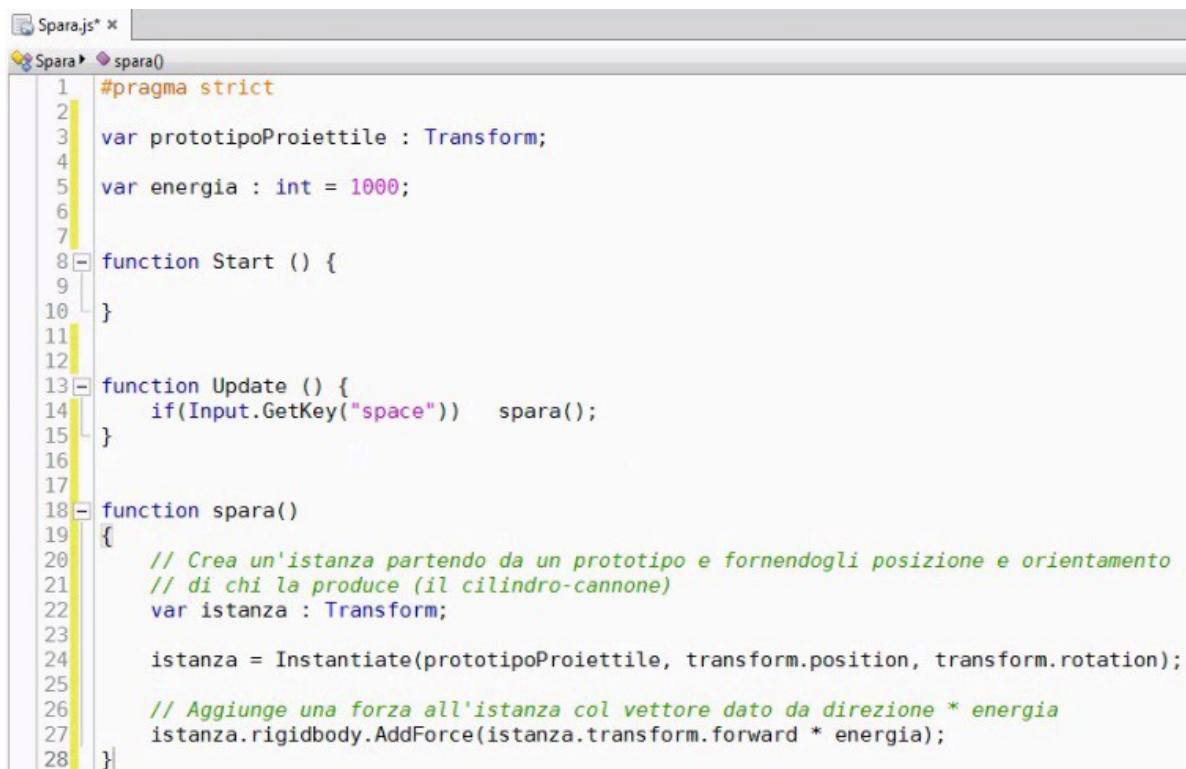
Esaminiamo l'istruzione: **dell'istanza, prendiamo il *igidbody*** e, di tale componente, **chiamiamo la funzione *AddForce*** (*igidbody.AddForce* significa appunto “*invoca la funzione AddForce di rigidbody*”), una **funzione** – tanto per cambiare – **nativa di Unity** che prende in input un valore, in particolare un **vettore**.

Il **vettore**, che qui è un insieme di **tre coordinate**, indica **la direzione dello sparo**... ma non solo: indica anche **la sua potenza**.

Il **parametro** che stiamo passando è ***istanza.transform.forward***, cioè la direzione in avanti **PROPRIA (quindi LOCALE)** dell'oggetto **istanza**, moltiplicata per l'energia, che è solo un **fattore moltiplicativo** per aumentare la spinta dell'oggetto (altrimenti verrebbe sì sparato in quella direzione ma **cadrebbe subito, perché privo di energia**).

La **variabile *energia*** l'abbiamo definita all'inizio e possiamo darle **il valore che vogliamo**; io qui ho messo 1.000 dopo aver fatto delle **prove**, comunque in seguito possiamo cambiarlo... ANZI, possiamo **cambiarlo anche durante il gioco** e vedremo come farlo nella seconda parte di questo **tutorial!**

Forse non ci crederete, ma con lo **script** abbiamo finito... quindi finalmente possiamo dare un'occhiata al quadro generale: definiamo un **prototipo di proiettile** (ancora non legato al **Prefab della sfera**, ma ci arriviamo subito), quindi se l'utente preme la **barra spazio**, rilevata da **Update**, creiamo **un'istanza del proiettile** e la **spariamo** nella direzione “in avanti” del cilindro con **energia 1.000**.

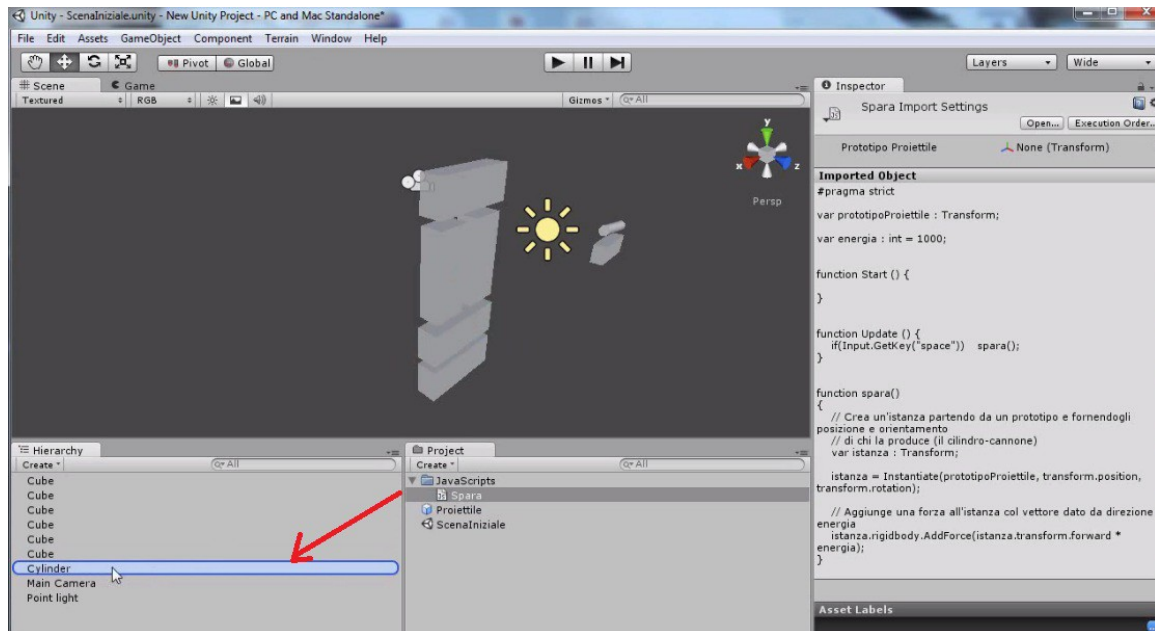
A screenshot of a code editor showing a JavaScript script named 'Spara.js'. The script is for a Unity game and defines a 'spara()' function. It includes comments in Italian explaining the logic: creating a projectile instance from a prototype and applying a force vector. The code is as follows:

```
1 #pragma strict
2
3 var prototipoProiettile : Transform;
4
5 var energia : int = 1000;
6
7
8 function Start () {
9
10 }
11
12
13 function Update () {
14     if(Input.GetKey("space"))    spara();
15 }
16
17
18 function spara()
19 {
20     // Crea un'istanza partendo da un prototipo e fornendogli posizione e orientamento
21     // di chi la produce (il cilindro-cannone)
22     var istanza : Transform;
23
24     istanza = Instantiate(prototipoProiettile, transform.position, transform.rotation);
25
26     // Aggiunge una forza all'istanza col vettore dato da direzione * energia
27     istanza.rigidbody.AddForce(istanza.transform.forward * energia);
28 }
```

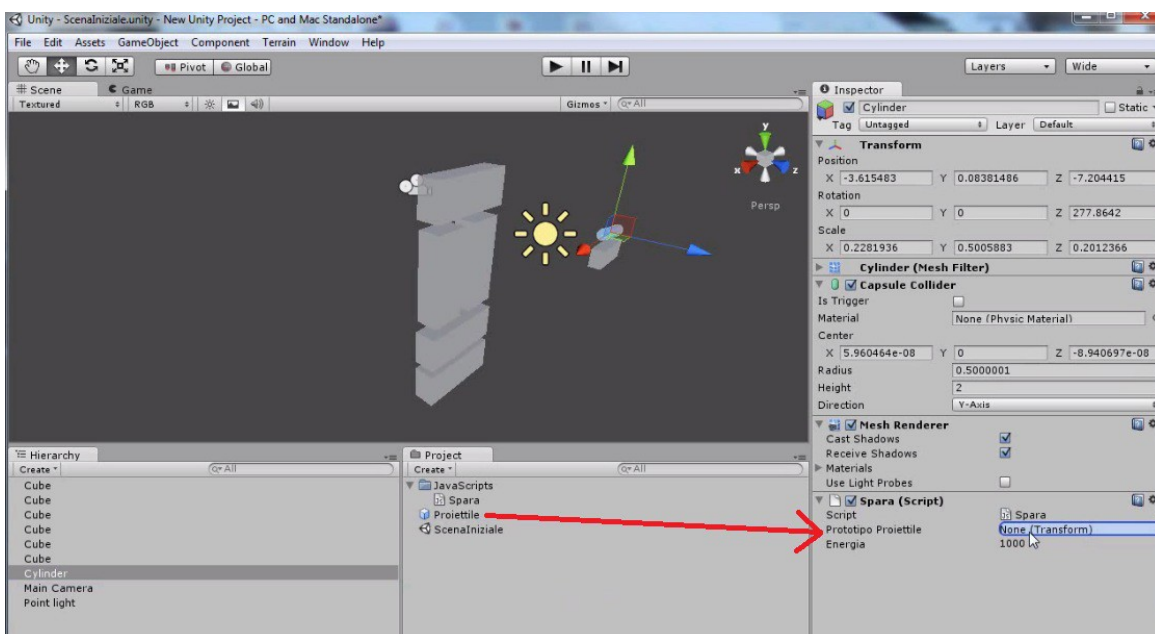
Questo “in avanti del cilindro” **contiene un errore subdolo** che risolveremo a breve... prima voglio mostrarvelo, così, in caso di problemi simili con altri progetti, vi verrà in mente questa soluzione... **sbagliando si impara!**

Salviamo lo **script** e torniamo alla **finestra principale di Unity**.

Trasciniamo lo **script** sull'oggetto **cilindro**, in modo da **associarlo al cannone**, quindi tutte le **istruzioni** tipo **transform.rotation** eccetera si riferiscono proprio a questo oggetto, che possiede lo **script**... ma non abbiamo ancora definito l'oggetto per la variabile **prototipoProiettile**!



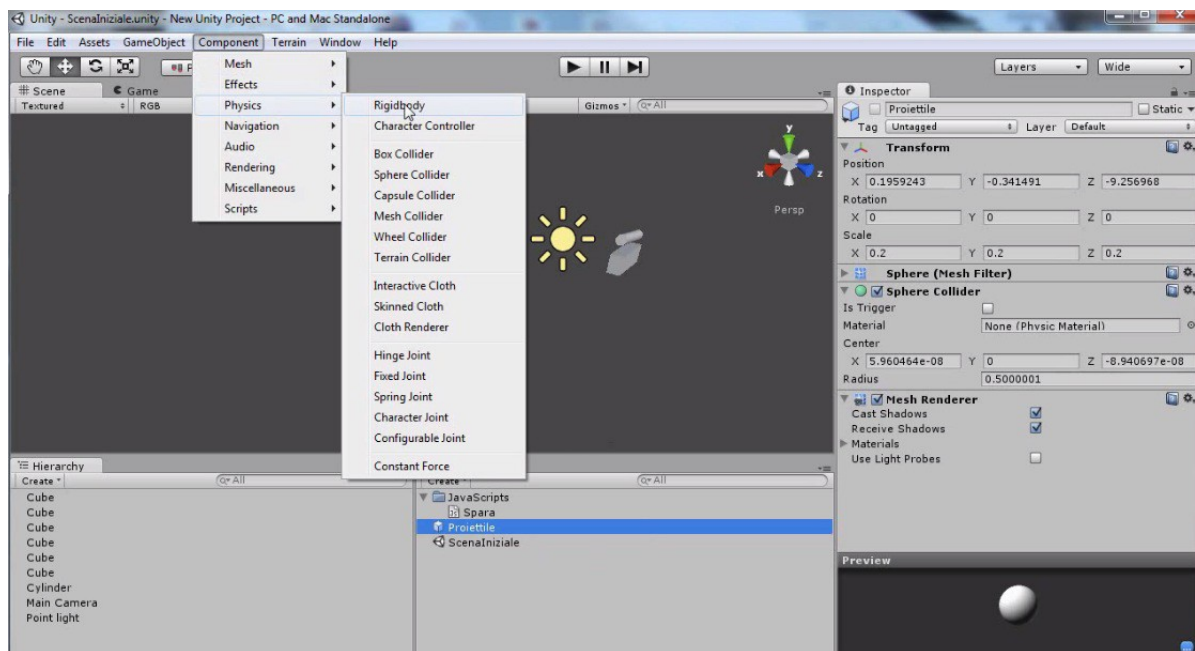
Ecco la magia: sempre **nella finestra di Unity**, con l'oggetto **cilindro** selezionato, che ora contiene lo **script** come suo **Component** in basso nell'**Inspector**, clicchiamo col tasto sinistro sul **Prefab “Proiettile”** nell'**Inspector** e, tenendolo premuto, **trasciniamolo** sullo slot accanto a **“prototipoProiettile”** nel **Component script** nell'**Inspector**, e rilasciamolo: fatto!



Abbiamo appena associato un **Prefab**, un oggetto di gioco, ad una **variabile** definita in uno **script** per un altro elemento, in questo caso un **Game Object**!

Quindi adesso quando **Instantiate**, nello **script**, prenderà come primo parametro **prototipoProiettile** e ne creerà uno al volo, userà proprio il **Prefab Proiettile**, che è stato **associato alla variabile dello script**!

Ancora una cosa prima di fare un **test**: selezioniamo il **Prefab proiettile** e associamogli un **nuovo Component Rigid Body** scegliendo tale voce proprio dal **menù Component**.

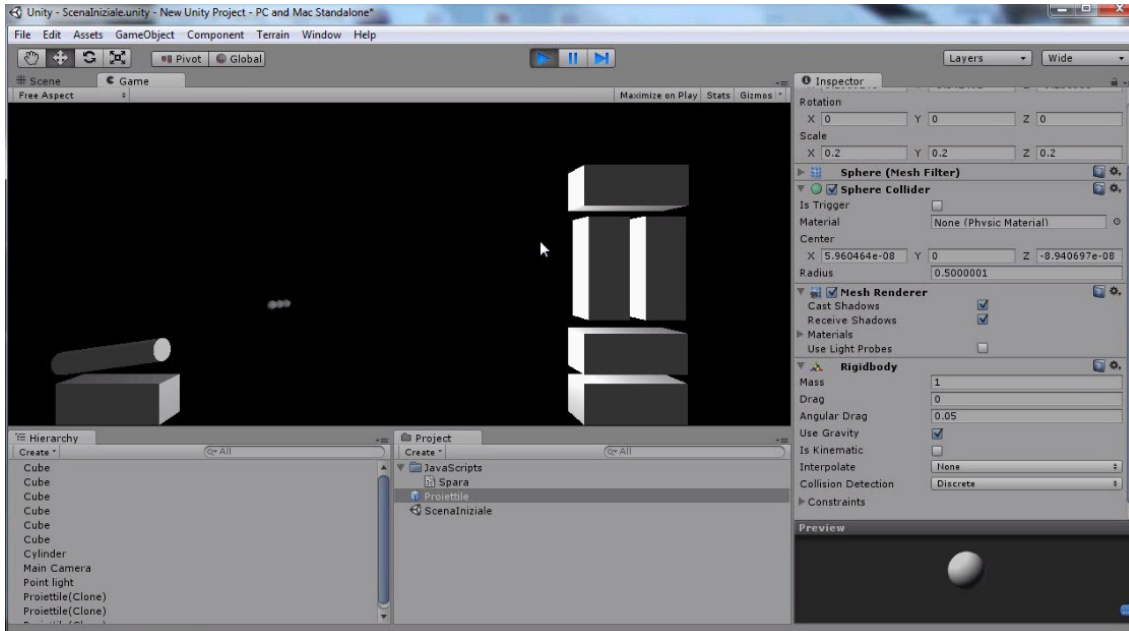


Questo va fatto perché **lo script accede**, se ricordate da **istanza.rigidbody**, al **Component Rigid Body** dell'istanza del prototipo, quindi **il prototipo deve avere un Rigid Body**.

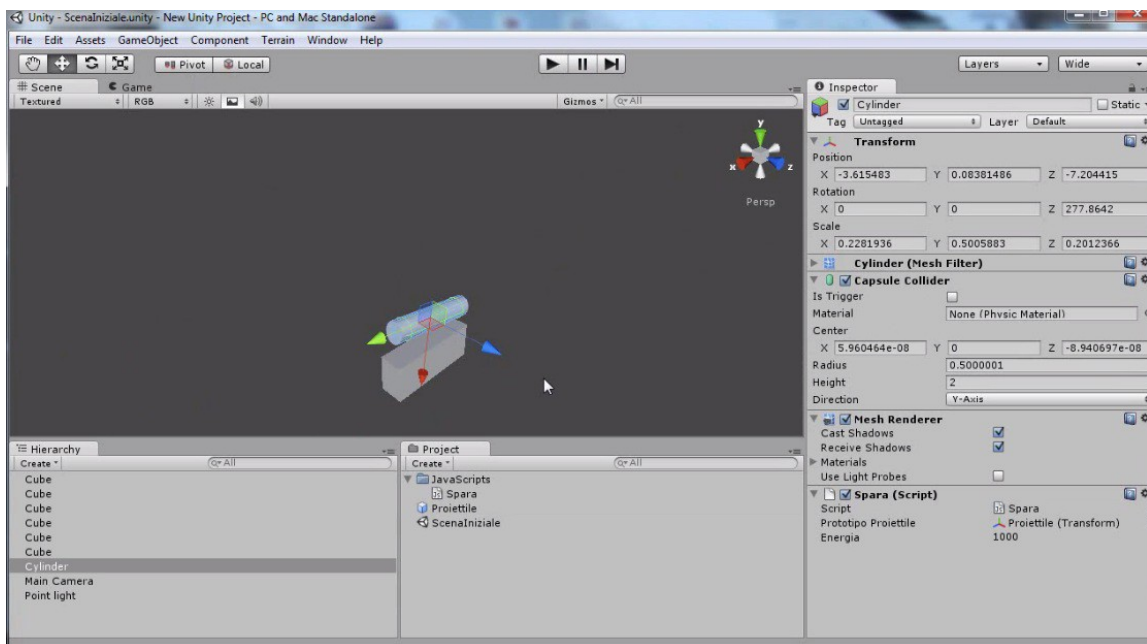
Potete modificare a piacere (o quasi) i **parametri del Rigid Body**; io sto lasciando quelli di default per la prova (ovviamente, se cambiate **drag** o la **massa**, dovrete cambiare anche **energia** nello **script**, per adeguarla ai nuovi valori).

A questo punto possiamo provare il tutto.

Premiamo **Play** e, appena **Unity** entra in modalità gioco, premiamo la **barra spazio** per vedere che succede: **il cilindro spara** effettivamente delle sfere, ma tali oggetti **vanno di lato** rispetto al cannone, non in avanti! Eppure abbiamo scritto “**istanza.transform.forward**”: cosa c'è che non va?



Per scoprirlo, uscite dalla modalità gioco, selezionate il **cilindro**, selezionate lo strumento di **traslazione** e la **modalità LOCAL** dall'apposito selettore.



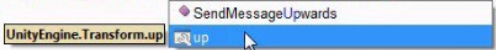
Dove punta l'asse **Z LOCALE**, ossia il *“forward”* locale dell'oggetto? Punta **di lato**... proprio dove vanno a finire i proiettili!

Qual è invece l'asse **LOCALE** che esce dal cannoncino, quindi la giusta direzione per lo sparo?

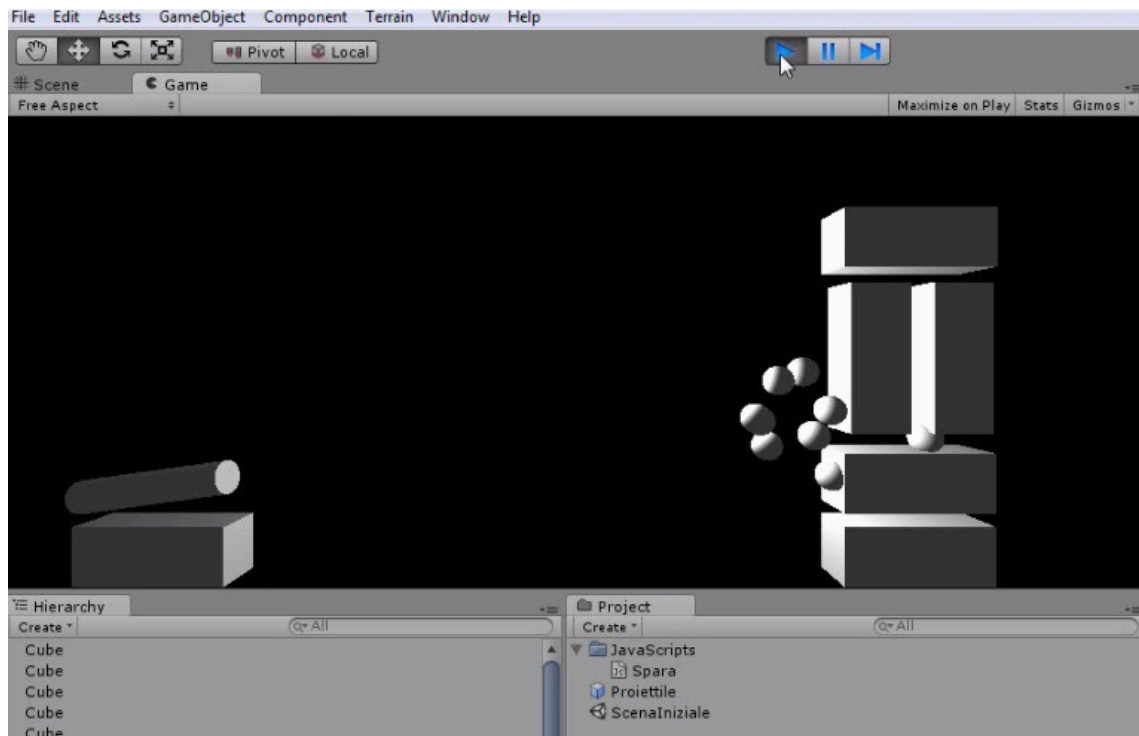
È l'asse **Y**, che nel codice di **transform** è detto *“up”*, sopra (d'altronde, nel sistema di **riferimento GLOBALE** della scena, l'asse verticale, che *“punta verso sopra”*, è l'asse **Y**).

Nel **codice dello script**, quindi, sostituiamo *istanza.transform.forward* con *istanza.transform.up*, dopodiché eseguiamo di nuovo il gioco con **Play** e spariamo un paio di colpi.

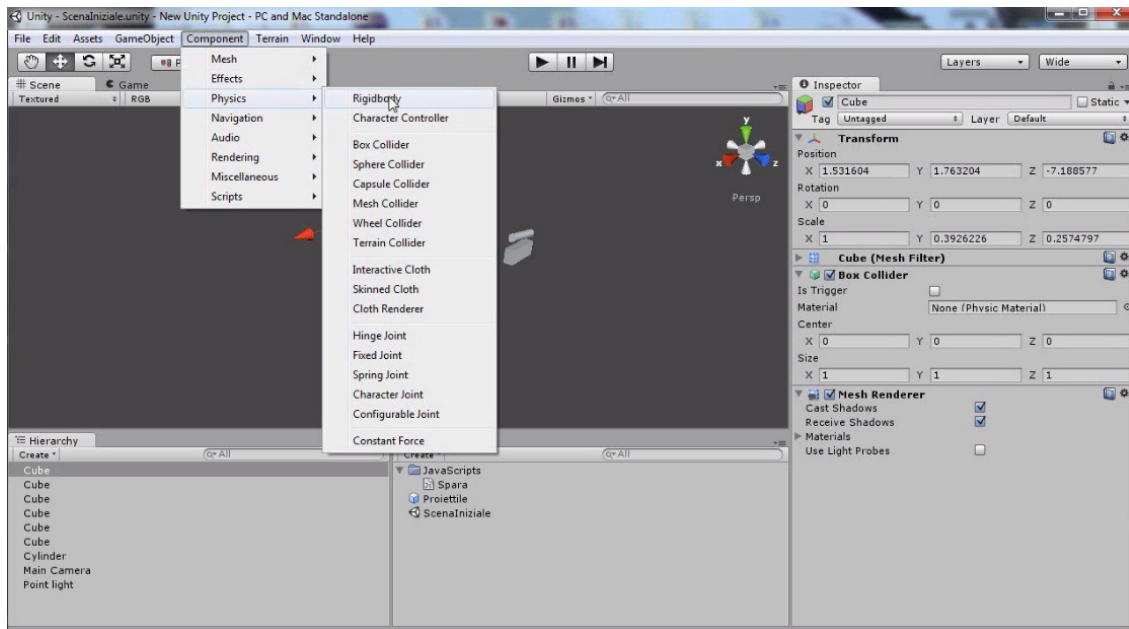
```
17
18 = function spara()
19 {
20     // Crea un'istanza partendo da un prototipo e fornendogli posizione
21     // di chi la produce (il cilindro-cannone)
22     var istanza : Transform;
23
24     istanza = Instantiate(prototipoProiettile, transform.position, trans
25
26     // Aggiunge una forza all'istanza col vettore dato da direzione * en
27     istanza.rigidbody.AddForce(istanza.transform.up * energia);
28 }
```



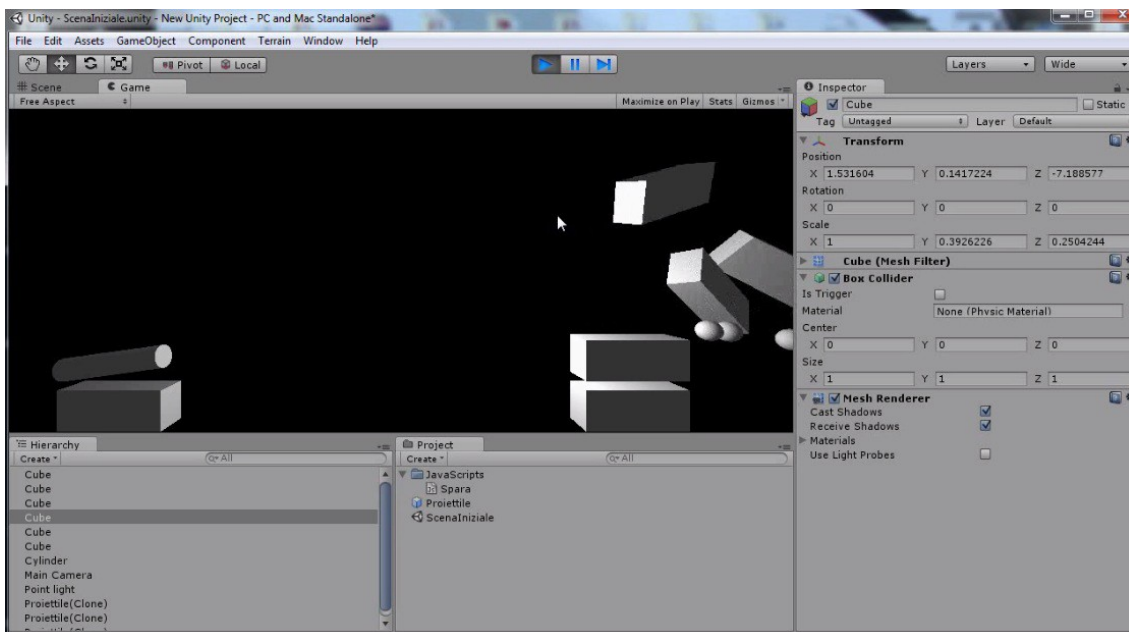
Ora i **proiettili** vanno nella **giusta direzione**, però **rimbalzano** senza spostare i bersagli.



Trattiamo i **bersagli**: per farli **reagire alle collisioni** della sfera, non dobbiamo far altro che dar loro un **Component Rigid Body**, impostando **massa** e altre caratteristiche a piacere. **Tutto qui.**



Per ottimizzare il lavoro, anche i **bersagli** dovrebbero essere dei **Prefab**: in questo modo, sarà sufficiente **dare il Component Rigid Body al Prefab** per ritrovarlo in un colpo solo in tutti gli oggetti creati partendo da quel modello.



Il **Rigid Body** fa cadere gli oggetti, quindi dobbiamo aggiungere delle mesh non **Rigid Body** oppure un **Plane** oppure un **Terrain** alla base, altrimenti i **pilastri cadranno nel vuoto...** ma questi sono dettagli che dipendono dal progetto: il punto è che i **bersagli**, per **reagire agli urti**, devono avere un **Component Rigid Body**.

Anche il secondo punto da discutere, quello relativo all'**energia del proiettile**, è molto semplice: bastano **poche righe di codice nello script**.

All'interno della funzione *Update* (che, in questo momento, contiene solo l'*input getkey* sul tasto “space”, la barra spazio, per richiamare la funzione “*spara()*”), aggiungiamo le seguenti istruzioni:

```
if(Input.GetKey("1")) energia = 1000;  
if(Input.GetKey("2")) energia = 2000;
```

Questo funziona perché “**energia**” è una **variabile globale dello script**, in quanto definita fuori da qualsiasi blocco funzione, come **prototipoProiettile...** ma possiamo fare di meglio!

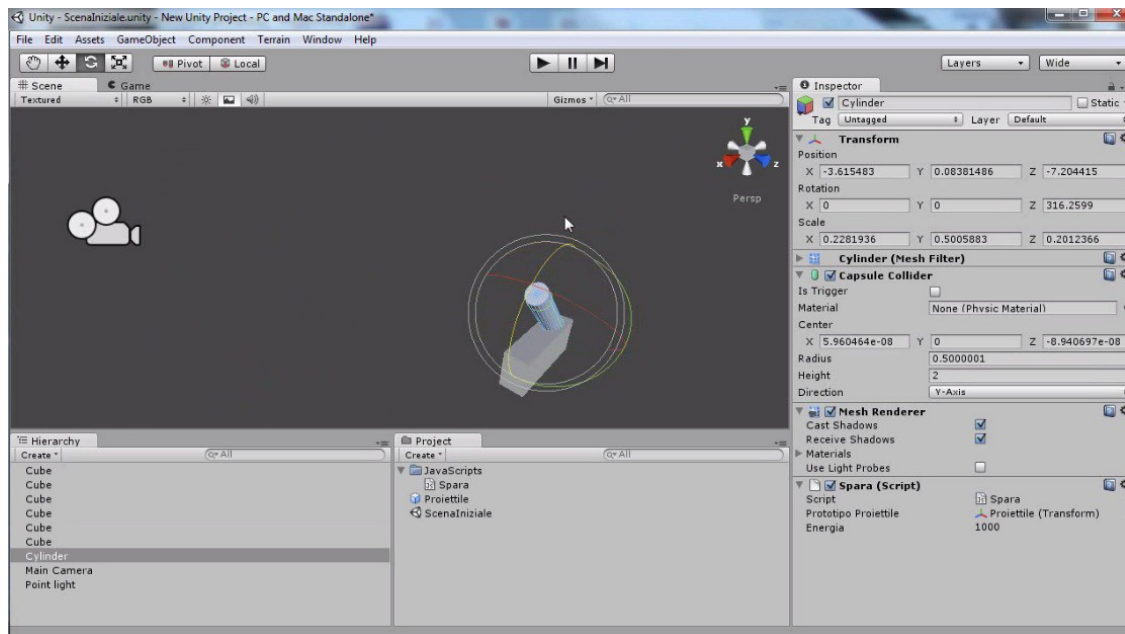
Togliamo modificare le due righe di codice appena inserite **scrivendo invece**:

```
if(Input.GetKey("+")) energia += 1000;  
if(Input.GetKey("-")) energia -= 1000;
```

In questo modo, **premendo + incrementeremo di 1000 il valore dell'energia**, mentre **premendo – lo abbasseremo di 1000**.

Come esercizio, potete scrivere delle istruzioni (sempre all'interno di questo script JS) per **mostrare a video**, con una scritta, il **valore dell'energia**, come fatto con il countdown in un tutorial precedente.

Come anticipato all'inizio, **NON tratteremo la rotazione del cilindro** (immagine nella pagina seguente): potete provare a farla, come esercizio, ma per farla bene bisogna trattare l'argomento del **Pivot Point**, il perno delle trasformazioni, che vedremo un'altra volta.



Qualche altra **considerazione**: l'esempio che vi ho mostrato è, almeno a livello visivo, molto grezzo, tuttavia contiene molti **elementi importanti** sia dal punto di vista della **programmazione scripting** (pensate a quante **funzioni o attributi** avete scoperto), sia dal punto di vista della “logica” dell'editor di Unity (su tutte, l'associazione al volo del Prefab *Proiettile* alla variabile *prototipoProiettile*) e anche un buon ripasso di concetti visti nei capitoli precedenti.

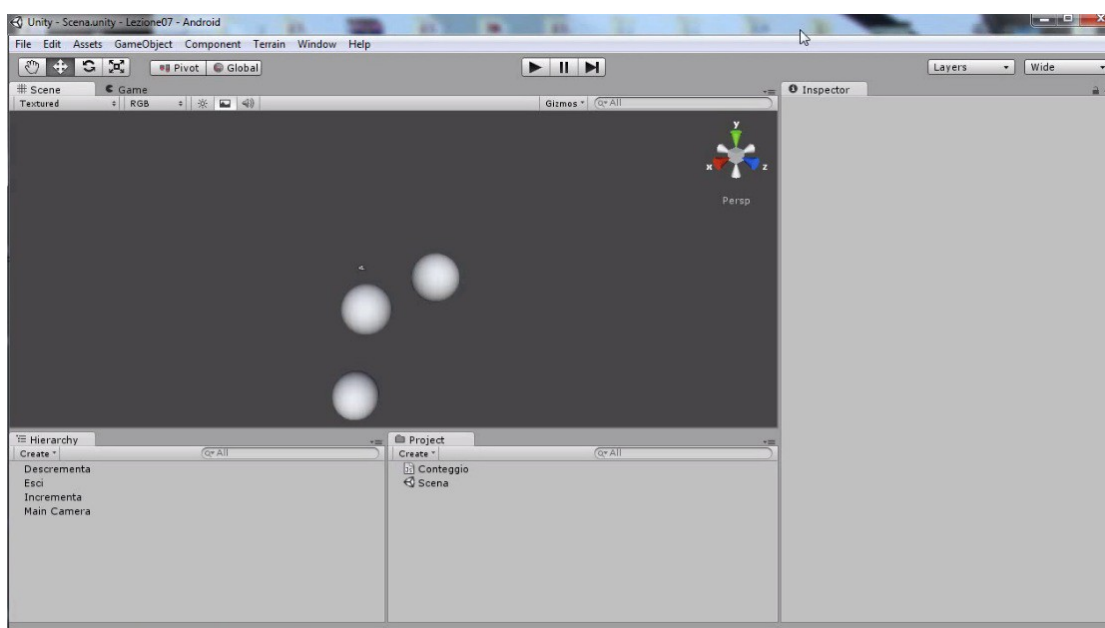
Nel prossimo capitolo vedremo qualche elemento di interazione con un dispositivo fisico, in particolare uno **smartphone con Android**; vedremo **come salvare sul dispositivo dei dati**, concetto alla base del salvataggio delle partite e dei punteggi, e come **compilare il gioco e passarlo sul proprio cellulare**.

Le basi di Unity 3D 3.x e 4.x – 7: Salvataggio e caricamento dati su disco (pc, tablet, ...)

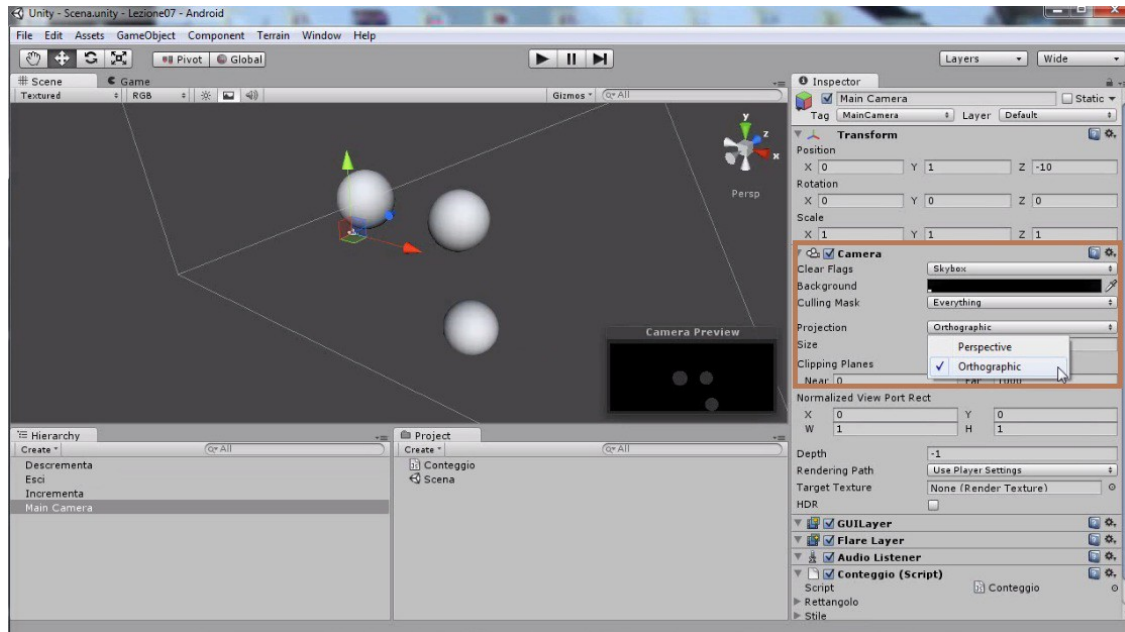
In questo capitolo vedremo **come salvare dei dati nel dispositivo** e come **recuperarli** ai riavvii successivi; vedremo, inoltre, il processo di **compilazione e trasferimento su uno smartphone** di un'applicazione realizzata in **Unity**.

Iniziamo con il **salvataggio dei dati sul dispositivo**, operazione che va gestita **via codice** ma che, per fortuna, è **molto semplice** da realizzare.

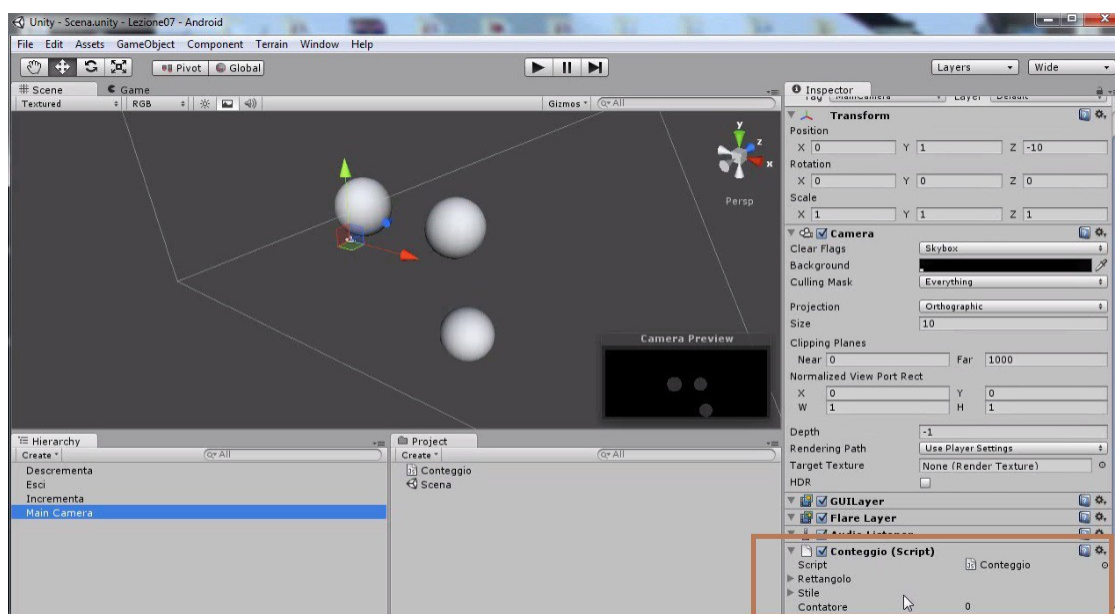
La **scena** che utilizzerò contiene **tre semplici sfere**, che simuleranno dei **pulsanti**, ma al centro apparirà anche una **scritta**, gestita dalla **GUI** con gli strumenti visti nel capitolo dove veniva realizzato, con la **funzione OnGUI**, il **countdown**.



I nomi delle sfere sono molto importanti: **Decrementa** per quella a sinistra, **Incrementa** per quella a metà schermo a destra, **Esci** per quella in basso a destra, in tutti i casi con le iniziali maiuscole; la **modalità di ripresa della telecamera** non è la solita **Prospettiva** ma **Ortho**, da impostare nell'**Inspector** dell'oggetto **Main Camera**, nella sezione **Camera**.



La scritta è il valore di una **variabile**, “**contatore**”, presente nello **script** “**conteggio**”, associato all'oggetto **Main Camera** (in basso a destra, nell'immagine seguente).



Con un **click**, mediante **touch** sul cellulare (in questo **tutorial**, un **Samsung Galaxy S**, ma il discorso vale per tanti modelli, inclusi gli **iPhone**) **sulla sfera di sinistra**, abbasseremo il valore della variabile “**contatore**”, mentre con un tocco su quella di **destra** lo **incrementeremo**; per concludere, un tocco sulla **terza sfera**, in basso a destra, **termina l'applicazione**, ma solo dopo aver **memorizzato sul dispositivo il valore** della variabile, in modo da ritrovarlo al prossimo riavvio.

La **scena 3D** è quindi molto **semplice** e tutto viene gestito dallo **script** “*conteggio*” associato alla **Main Camera**, per cui esaminiamo tale **script** con l'**Editor**, nel mio caso **MonoDevelop**.

Partiamo dall'elemento più semplice che, in realtà, non è la dichiarazione della variabile, ma la **funzione OnGUI** (da definire in spazio globale, fuori da qualsiasi altra funzione), che si limita a **scrivere tale valore a video**, ma lo fa **al centro** del dispositivo, e lo fa al centro di **QUALSIASI dispositivo**, indipendentemente dalle **dimensioni** o dalla **risoluzione**... ecco come:

```
function OnGUI()  
{  
    GUI.Label(rettangolo, contatore.ToString(), stile);  
}
```

Analizziamo l'**unica istruzione** presente.

La funzione **GUI.Label** serve a **disegnare un'etichetta di testo (Label) a video**, e lo fa prendendo **tre parametri**: un **oggetto Rect**, un rettangolo, che definisce l'**area di disegno** della scritta, più ovviamente **la stringa che vogliamo mostrare** (e siccome **contatore** è una variabile intera, dobbiamo usare la funzione nativa degli interi in **Unity ToString()** per ottenere, **da quel valore numerico, una stringa**), più un altro oggetto particolare, un oggetto **GUIStyle**, chiamato qui, con poco sforzo di fantasia, “**stile**”.

È chiaro che tutti questi oggetti devono essere stati **definiti** da qualche altra parte (per motivi di performance: **inutile ricreare i Rect o gli Style ad ogni aggiornamento della schermata**), per cui ora spostiamoci nella **parte iniziale dello script**, fuori da qualsiasi funzione, dove troviamo le dichiarazioni:

```
var rettangolo : Rect;  
var stile : GUIStyle;
```

che sono appunto le **dichiarazioni delle due variabili**.

L'**inizializzazione** avviene invece nella **funzione Start**, per cui all'interno di questa **function** abbiamo:

```
stile.fontStyle = FontStyle.BoldAndItalic;  
stile.normal.textColor = Color.white;  
stile.alignment = TextAnchor.UpperRight;
```

```
10 function Start () {  
11     stile.fontStyle = FontStyle.BoldAndItalic;  
12     stile.normal.textColor = Color.white;  
13     stile.alignment = TextAnchor.UpperRight;  
14  
15     rettangolo = new Rect(Screen.width/2-100, Screen.height/2-50, 200, 100);  
16  
17     if(PlayerPrefs.GetInt("ValoreContatore") != null)  
18         contatore = PlayerPrefs.GetInt("ValoreContatore");  
19     else     contatore = 0;  
20 }  
21
```

“**stile**”, quindi, definisce lo **stile del testo**, e lo fa con parametri propri dell'oggetto **GUIStyle**; ora, non posso descrivervi a fondo questi parametri, perché ci vorrebbero pagine e pagine solo per **GUIStyle**, ma nei capitoli precedenti abbiamo visto come esplorare la documentazione online di Unity, che potete utilizzare per conoscere i **parametri** e i **valori** possibili... in questo caso, definiamo lo **stile grassetto e corsivo** con **BoldAndItalic**, più il **colore bianco** per la scritta con **Color.white** (quindi **Color** è un oggetto nativo di **Unity**) e l'**allineamento a destra** all'interno del rettangolo di disegno con **TextAnchor.UpperRight**.

Parliamo ora delle istruzioni relative al **rettangolo**, che si trovano sempre in **Start**:

```
rettangolo = new Rect(Screen.width/2-100, Screen.height/2-50, 200, 100);
```

L'oggetto **Screen** è, tanto per cambiare, un oggetto **nativo di Unity**, che fa riferimento allo “**schermo**”, al **display** di visualizzazione e che mette a disposizione, in particolare, i due valori (o attributi) **width** ed **height**, ossia **larghezza ed altezza**.

Il **rettangolo di disegno** ha larghezza 200 e altezza 100, inoltre la creazione di un **oggetto Rect** (che si fa con **new Rect** e i parametri richiesti) richiede, appunto, quattro parametri: le **coordinate x e y** all'interno della schermata e la **larghezza e l'altezza del rettangolo da disegnare**, per cui per piazzarlo al centro della schermata scriviamo appunto

```
rettangolo = new Rect(Screen.width/2-100, Screen.height/2-50, 200, 100);
```

Questo è tutto per quanto riguarda il **“disegno” dei valori a video (sul display)**, ma, prima di procedere con la gestione della variabile **contatore**, una piccola nota...

Le funzioni che riguardano il **rilevamento del tocco su schermo** e l'associazione del tocco al **Game Object** corretto... **NON** verranno trattate in questo tutorial, ma **nel prossimo**.

Il motivo è semplice: già solo per la **gestione del salvataggio dei dati**, argomento di questo **tutorial**, dobbiamo vedere un bel po' di codice, classi e funzioni varie, quindi in questo capitolo ci concentreremo sul passaggio successivo, ossia il **salvataggio e il recupero del valore di contatore dalla memoria del dispositivo**.

In ogni caso, qui di seguito trovate il **codice necessario** (presente nella funzione **Update** e in una funzione a parte) da ricopiare per provare subito questo **tutorial** (comunque dovremo dare una sbirciatina a un **else if** per vedere la gestione del **salvataggio dei valori sul dispositivo**):

```
function Update () {  
    // Rileva il tocco  
    if((Input.touchCount>0) && (Input.GetTouch(0).phase ==  
    TouchPhase.Ended)) {  
        var gameObject : GameObject =  
        GetClickedItem(Input.GetTouch(0).position);  
        if(gameObject.name == "Decrementa") contatore -= 1;  
        else if(gameObject.name == "Incrementa") contatore += 1;  
        else if(gameObject.name == "Esci") {  
            PlayerPrefs.SetInt("ValoreContatore", contatore);  
            Application.Quit();  
        }  
    }  
}
```

```
// Rileva l'elemento cliccato
function GetClickedItem(pos: Vector2) {
    var camera : Camera = Camera.main;
    var p : Vector3 = camera.ScreenToWorldPoint(Vector3 (pos[0], pos[1],
        camera.nearClipPlane));
    var hit : RaycastHit;
    if(Physics.Raycast(p, Vector3(0.0, 0.0, 1.0), hit)) return
        hit.collider.gameObject;
    else return null;
}
```

Una piccola nota su **Vector Raycast**: le sfere sono **di fronte alla telecamera lungo l'asse Z globale positivo**; ecco perché il vettore di **Raycast** è **0, 0, 1** (in riferimento a **XYZ**)... nella prossimo capitolo torneremo meglio su tutto questo; per il momento, lo stiamo dicendo qualora dovessero sorgere problemi mettendo le sfere in altre **posizioni rispetto alla telecamera** (ad esempio, **ruotando** tutto verso il basso).

Parliamo ora della variabile globale **contatore**.

La **variabile** va dichiarata, ma **NON inizializzata**, all'inizio dello **script**, insieme a **rettangolo**, **stile** e tutte le altre **variabili globali** che vogliamo dichiarare.

All'interno della funzione Start, abbiamo un controllo **if** particolare, per **assegnare il valore iniziale a contatore**:

```
if(PlayerPrefs.GetInt("ValoreContatore") != null) contatore =
    PlayerPrefs.GetInt("ValoreContatore");
else contatore = 0;
```

Per spiegarvi cos'è **PlayerPrefs** devo dirvi, prima, come **Unity** memorizza i valori sui dispositivi, o meglio: **quali funzioni ci mette a disposizione per la memorizzazione e il recupero**, nascondendoci il lavoro sporco di dover interagire con **il software e l'hardware di ogni sistema**.

“PlayerPrefs” sta per **“Preferenze del giocatore”**, ossia **“i vostri dati”** (i valori che volete **memorizzare**, eccetera).

Un **PlayerPref** è una **coppia di dati**: una **stringa**, che è il nome della variabile, e un **valore**, che può essere di tipo intero, float o stringa.

A seconda del **tipo** di dato, si useranno funzioni **GET** (per il **recupero**) e **SET** (per **impostare** il valore) diverse; ad esempio, **GetString** se vogliamo recuperare un dato di tipo **Stringa**, **SetInt** se vogliamo **scrivere** un valore in una variabile di tipo **intero**.

Il **nome** serve, chiaramente, a **recuperare il valore**, proprio come il **nome di una variabile**.

Nel nostro caso, vogliamo **salvare solo un dato**: il **valore**, numerico intero, della **variabile contatore**. Definiremo quindi il **PlayerPref “valoreContatore”** (e con questo **nome-stringa** potremo recuperarlo) e il **tipo intero**, reso evidente dall'utilizzo di funzioni **GetInt** e **SetInt**.

In fase di **avvio del gioco**, quindi, il **controllo if** in **Start** chiede: *“esiste una variabile chiamata valoreContatore salvata sul dispositivo in uso?”* (questa domanda viene fatta chiedendo: “il **PlayerPrefs** **valoreContatore** è diverso dal **tipo null**, ovvero **esiste?**”); se questo elemento esiste, allora **prendi il suo valore** (con **GetInt**) e assegnalo alla **variabile contatore**.

La **non esistenza del valore** si ha al **primo avvio** dell'applicazione: in quel momento, **non è mai stato salvato su disco** il valore di contatore, quindi il **PlayerPrefs** **valoreContatore** non esiste, ed ecco che entra in gioco l'**else**: se non c'è, **assegna il valore iniziale 0** alla variabile **contatore**.

La **scrittura del valore**, prima dell'uscita, si trova in un **else if** della funzione di **controllo del tocco** (che vedremo meglio nella prossima puntata); in particolare, **se viene toccata la sfera Esci**, abbiamo:

```
PlayerPrefs.SetInt("ValoreContatore", contatore);  
Application.Quit();
```

```
32     var gameObject : GameObject = GetClickedItem(Input.Get  
33     if(gameObject.name == "Decrementa")      contatore  
34     else if(gameObject.name == "Incrementa")  contatore  
35     else if(gameObject.name == "Esci")  
36     {  
37         PlayerPrefs.SetInt("ValoreContatore", contatore);  
38         Application.Quit();  
39     }  
40 }  
41
```

PlayerPrefs.SetInt si occupa di creare, se non presente, il dato di nome **“ValoreContatore”** sul dispositivo, altrimenti se già c'è (dalla seconda esecuzione in poi) semplicemente lo **sovrascrive**, inserendo il valore numerico intero dato dal **secondo parametro**, la **variabile contatore** (da notare che **GetInt** prende **un solo parametro**, ossia il **nome del dato** da recuperare, mentre **SetInt** ne prende **due**: il **nome** del dato da sovrascrivere e il **valore**, dato da una **variabile dello script**, da salvare su disco).

Segue l'istruzione nativa **Application.Quit()**, letteralmente *“esci dall'applicazione”*, che **termina il gioco**.

Come potete vedere, **la gestione del salvataggio dati è semplicissima**, indipendentemente dal **dispositivo** scelto, sia esso un **computer**, una **console**, un **iPhone** o uno **smartphone** o un **tablet con Android**: è sufficiente definire **i nomi dei dati** e recuperarli o scriverli con un paio di **funzioni native**, senza preoccuparsi del resto.

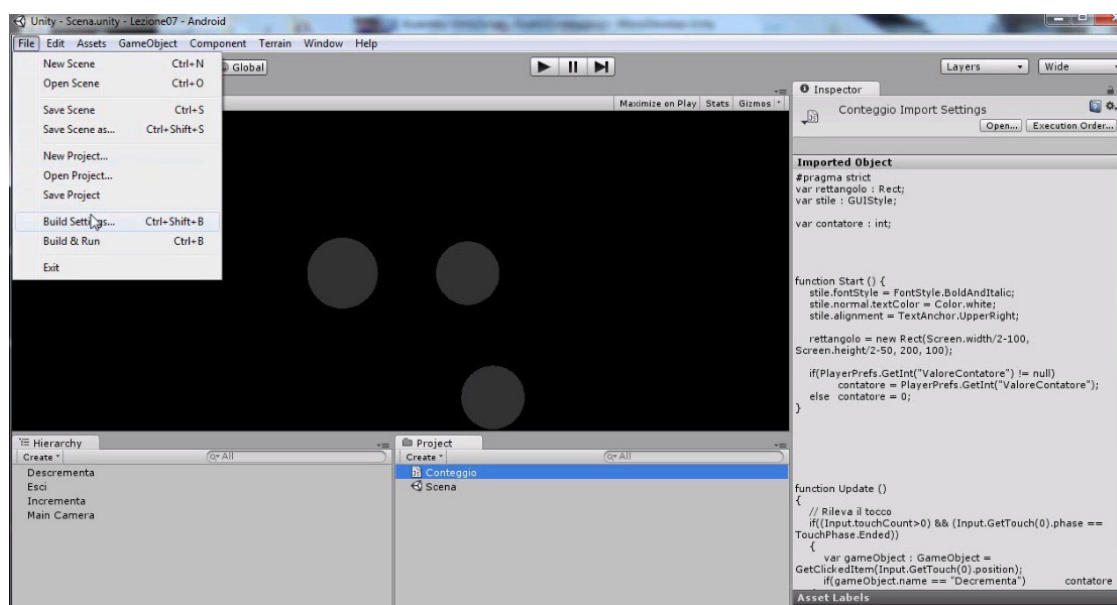
```
27 function Update ()
28 {
29     // Rileva il tocco
30     if((Input.touchCount>0) && (Input.GetTouch(0).phase == TouchPhase.Ended))
31     {
32         var gameObject : GameObject = GetClickedItem(Input.GetTouch(0).position);
33         if(gameObject.name == "Decrementa")    contatore -= 1;
34         else if(gameObject.name == "Incrementa")    contatore += 1;
35         else if(gameObject.name == "Esci")
36         {
37             PlayerPrefs.SetInt("ValoreContatore", contatore);
38             Application.Quit();
39         }
40     }
41 }
```

Questo sta alla base anche dei **savegame più complessi**: a seconda dei **progetti**, magari dovrete **creare funzioni ad hoc per il salvataggio** che recupereranno certe informazioni (ad esempio, le **coordinate del personaggio**, il suo livello di **energia**, le posizioni dei nemici, le **armi raccolte**, eccetera) e le scriveranno o, di contro, **al caricamento di un livello**, **recupereranno** tali dati dalla memoria e **li imposteranno nelle variabili** di posizione, orientamento ed altri dati per **ripristinare il livello salvato** in precedenza.

* * *

In questa seconda parte del tutorial vedremo come creare l'**apk**, ovvero il **pacchetto applicazione per piattaforma Android**.

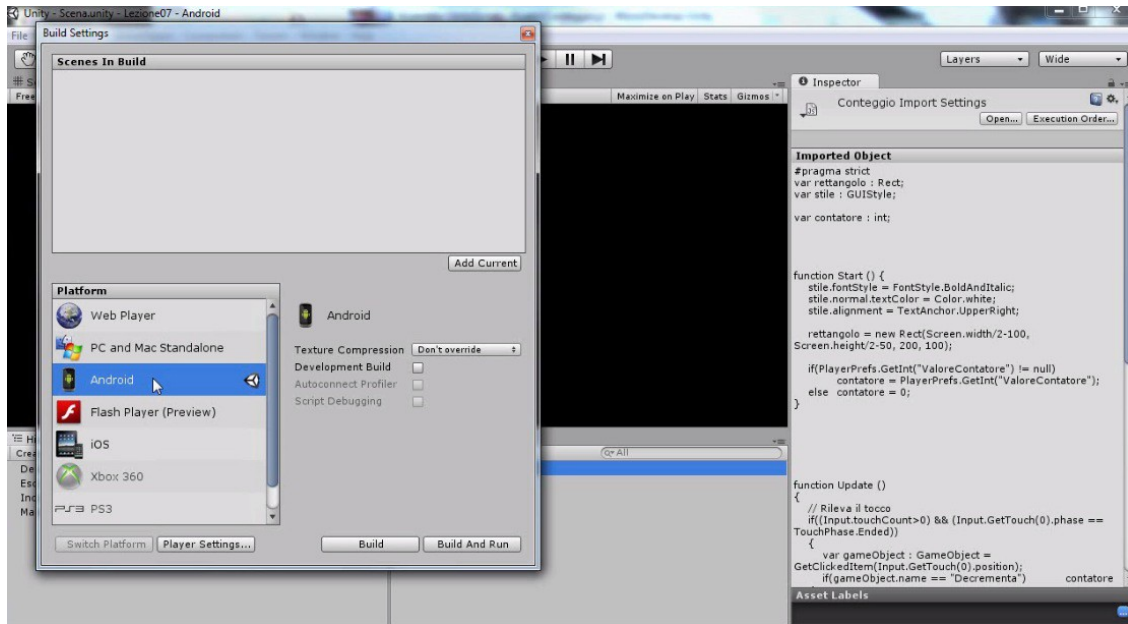
Per poter creare i **pacchetti eseguibili**, dobbiamo scegliere **File – Build Settings**, letteralmente *“impostazioni di costruzione”*, di realizzazione.



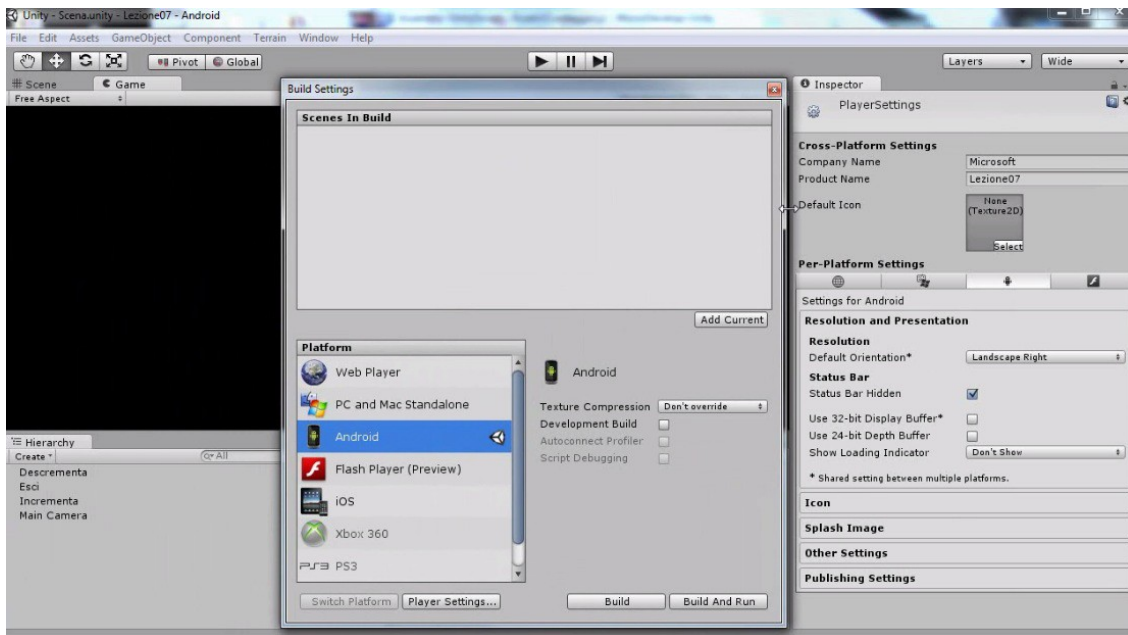
Nella scheda che apparirà, per prima cosa dovremo **scegliere il dispositivo di destinazione**; a seconda della **licenza** e del sistema utilizzati per lo sviluppo, certe voci saranno disponibili e certe altre no: ad esempio, nello screenshot nella pagina seguente si vede che la mia versione dispone **chiave di licenza per PC**, quindi anche **Flash** (per pubblicare le mie app su Internet), più **iOS e Android**, tuttavia sto usando **Unity su un sistema Win**, quindi, anche se **la chiave di licenza** me lo consente in via teorica, in pratica **NON posso compilare le App per iOS**.

Non posso fare nemmeno **compilazioni per PS3, Wii o XBOX**, questa volta perché **la mia chiave di licenza non prevedeva quei dispositivi di destinazione**. Le chiavi di licenza possono essere estese per includere altri dispositivi; per farlo, è necessario **contattare l'assistenza di Unity per modalità e, a volte, prezzi**.

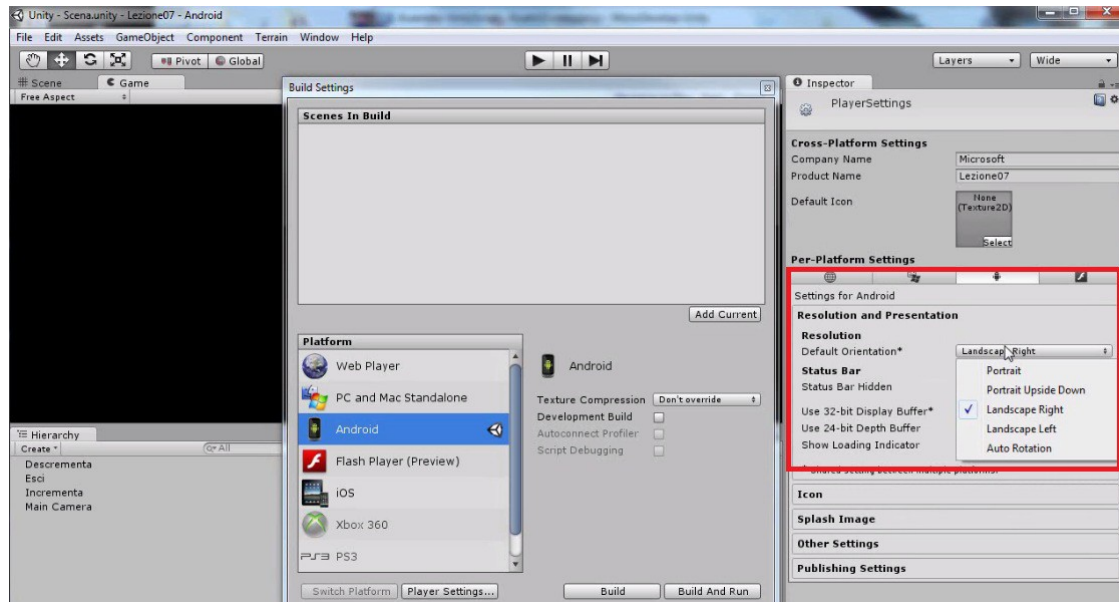
Seleziono quindi **Android**.



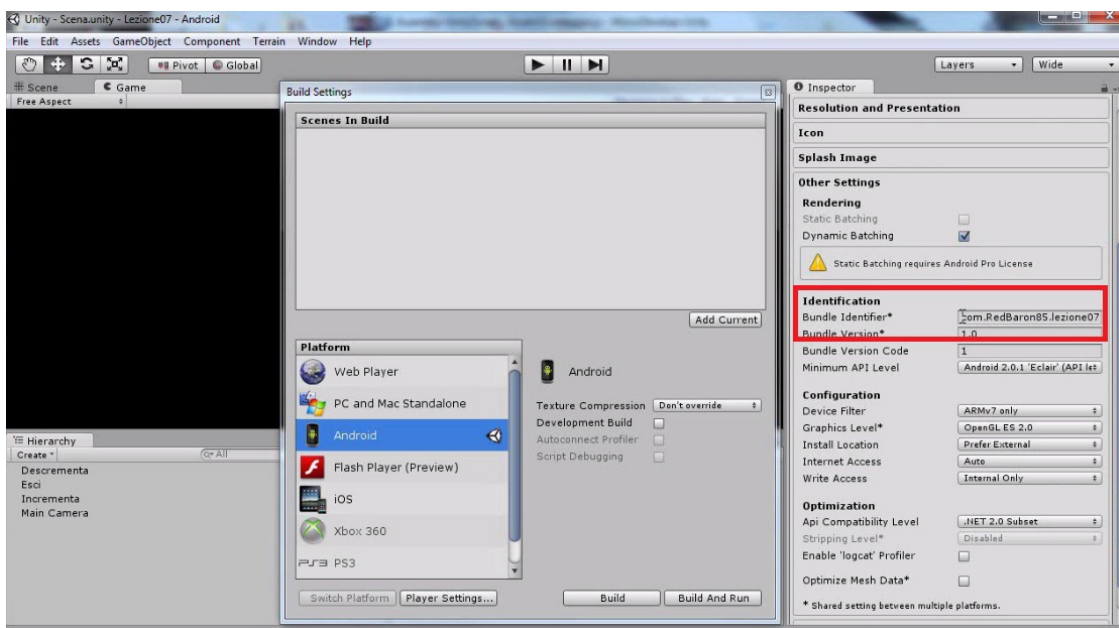
Cliccando su **Player Settings**, nell'**Inspector** della scena si apriranno dei **pannelli di personalizzazione** dell'esecuzione, dove poter mettere ad esempio **il livello di compatibilità minima dell'hardware**, le thumbnails, l'orientamento dello schermo ed altre impostazioni del genere...



Nel mio caso (in questi screenshot), sto lasciando le **impostazioni di default** (come **thumbnail**, ci sarà quella di **Unity**), tranne due: l'**orientamento Landscape Right** (per orientarlo in maniera orizzontale)



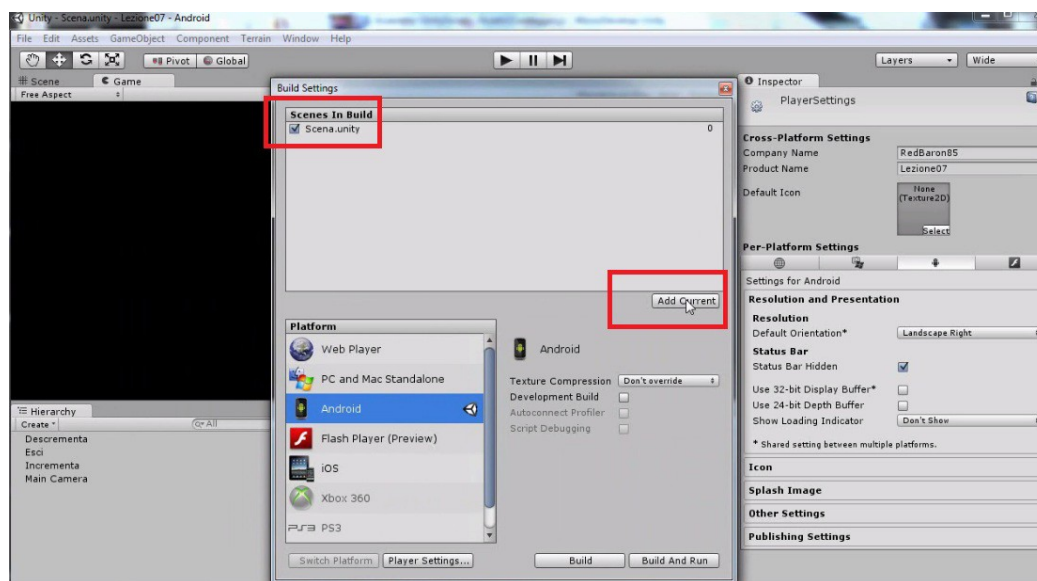
e, nella sezione **Other Settings**, il campo **Bundle Identifier**, che va riempito con una scritta nella forma **com.prodotto.reproduttore**; per questo esempio, ho digitato **com.RedBaron85.lezione07**.



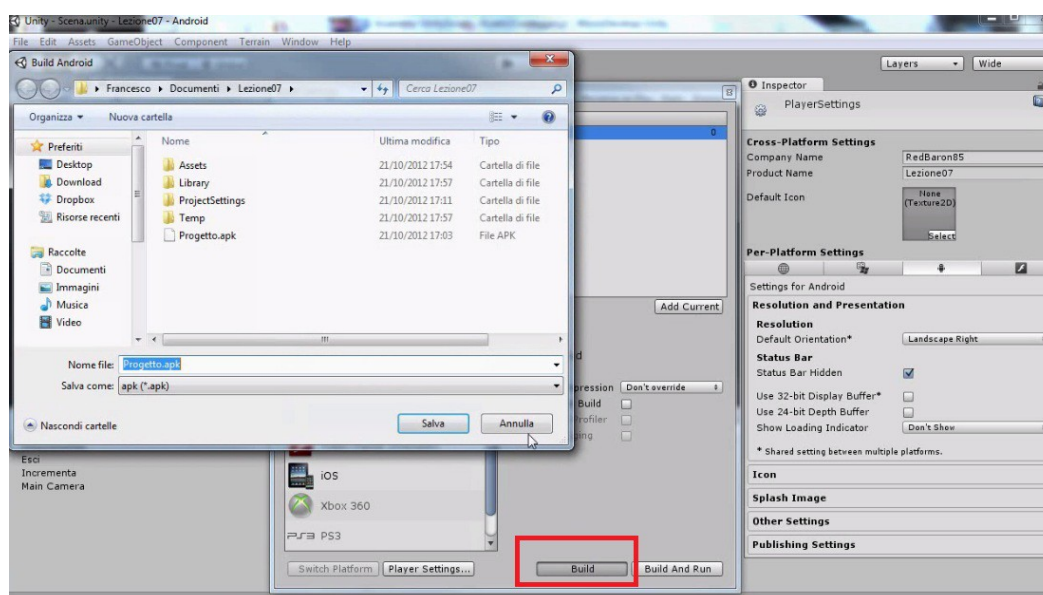
Attenzione, nota importante: è necessario inserire, nella parte alta di **Build Settings**, le **Scenes**, ovvero le scene o **livelli** che vogliamo includere nel gioco, facendo attenzione soprattutto **alla prima della lista**, che è quella che si avrà **all'avvio**.

Di default, **l'elenco è vuoto** e un'app così compilata non avrebbe un bel niente al suo interno.

Bisogna quindi **salvare almeno la scena corrente** come **file Scena negli Assets di gioco** (la scena apparirà nel pannello **Project**) con **File – Salva**, poi tornare qui in **Build Settings** e cliccare su **Add Current** per aggiungere la scena corrente al progetto.

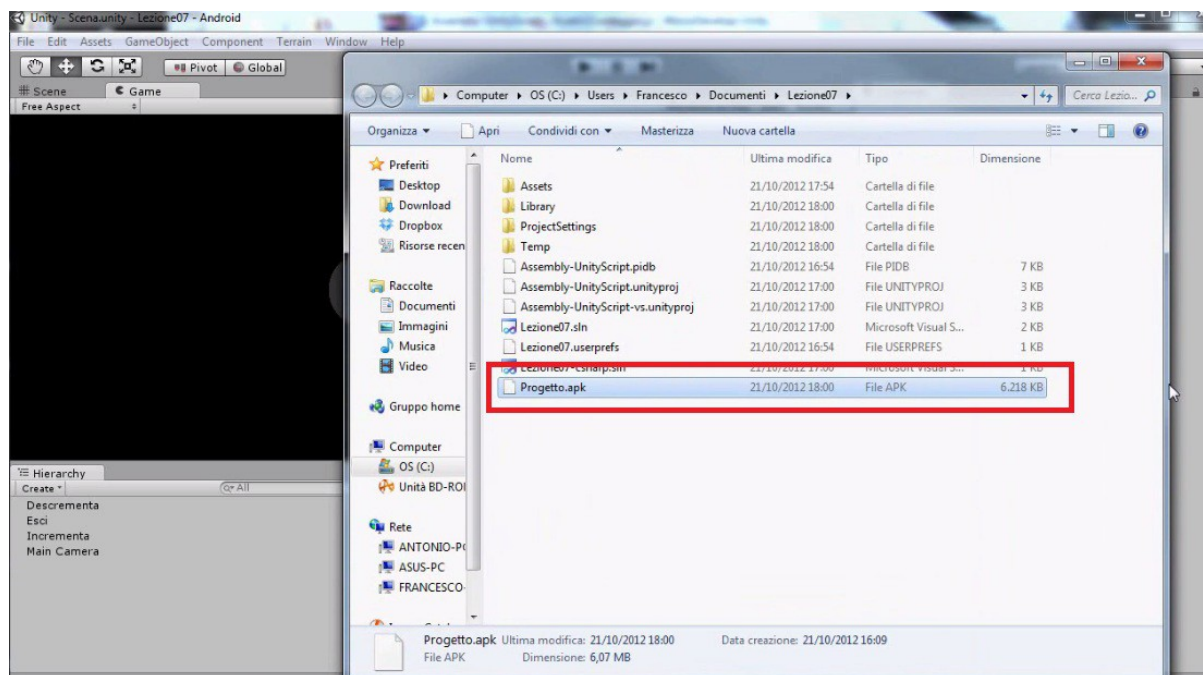


A questo punto possiamo **cliccare su Build**, scegliere **nome e percorso del file apk** nella schermata che apparirà e **confermare la compilazione**.



Nota a margine per chi ha un device **Android**: è necessario avere **installato**, sul pc, l'**Android SDK** (il **kit di sviluppo**, **System Development Kit**), per cui prima di fare tutto questo dovete **scaricare** l'**Android SDK** e installarlo; alla prima compilazione per **Android**, **Unity** vi chiederà il **percorso della cartella con l'SDK**, mentre ai successivi se lo ricorderà automaticamente.

Se tutto va bene, **al termine della “build”** avremo il file **apk** su disco.



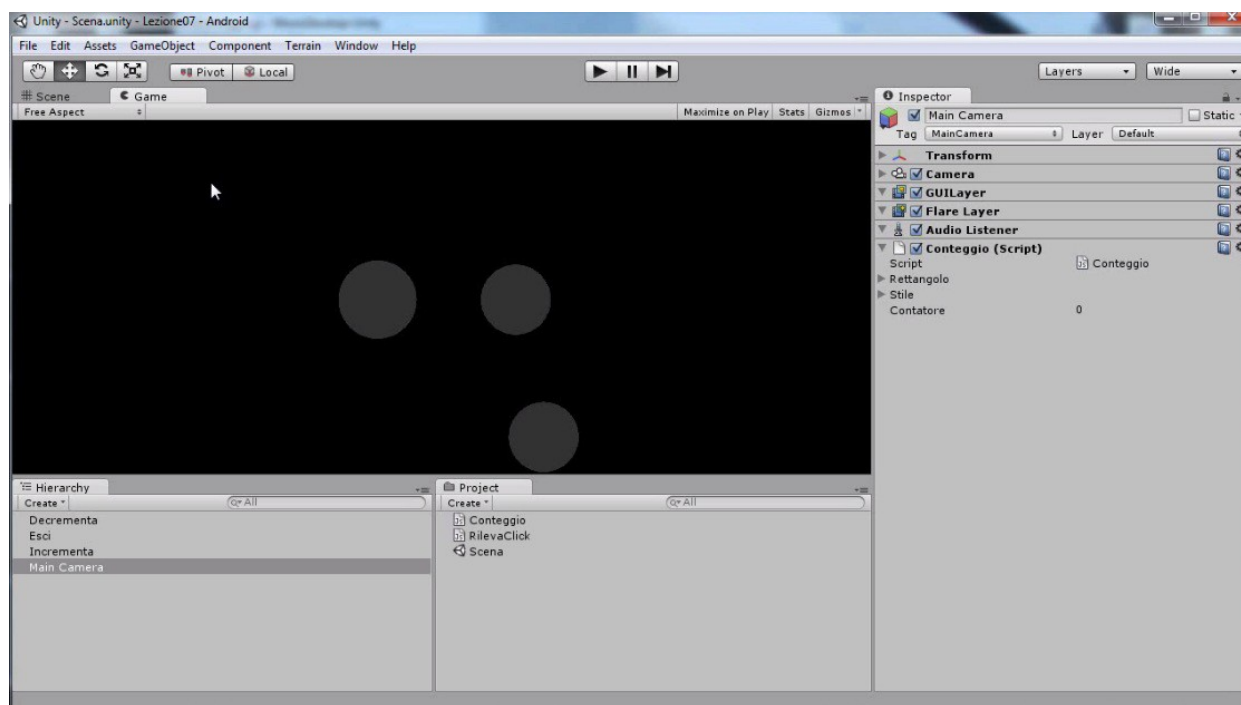
Dovete quindi **collegare lo smartphone o il tablet al pc** e **passare questo file in una cartella nel dispositivo**, ad esempio nella cartella **download**.

Da qui in poi le **operazioni variano in base al dispositivo**; nel **Galaxy S**, bisogna **aprire l'applicazione Archivio**, scegliere la **cartella** nella quale è stato depositato l'**apk**, **clickare** sulla relativa voce e **procedere con l'installazione**; al termine, l'applicazione sarà **disponibile come App**, con la sua icona, nella schermata **Home** del **Galaxy**, pronta per essere eseguita.

Le basi di Unity 3D 3.x e 4.x – 8: tocco delle dita su touch; click del mouse su pc

In questo ottavo capitolo della guida introduttiva su **Unity 3D** esamineremo le **funzioni** utilizzate nel **capitolo precedente** per recuperare il **tocco delle dita sul dispositivo**; inoltre, vedremo un paio di **funzioni per recuperare la posizione e i click del mouse**, da aggiungere a quanto fatto finora, perché che l'app definita fino a questo punto **non può essere eseguita su pc** (in quanto prevede l'interazione solo mediante **tocco sullo schermo**, non click del mouse o altro).

Riprendiamo quindi il progetto del capitolo precedente; come detto in quel tutorial, il cuore della gestione del tocco sullo schermo si trova nelle **funzioni *Update* e soprattutto *GetClickedItem***, letteralmente *“recupera oggetto cliccato”*; delle due, solo la prima è una funzione nativa di **Unity**, per cui il nome dev'essere esattamente quello, mentre per la seconda potete scegliere anche un nome diverso da quello che le ho dato io.



In realtà, la **porzione di script che riguarda il tocco**, in *GetClickedItem*, è veramente semplice, ma va discussa passo per passo per capire, la prima volta, come funziona, dopodiché risulta chiarissima e può essere utilizzata in altre occasioni.

Il sistema che vedremo qui si basa sulla **modalità Ortho della telecamera** e tiene conto anche dell'orientamento, nell'universo virtuale, di tale oggetto.

Iniziamo da **Update**, che in questo progetto contiene **solo un blocco if che rileva il tocco**, con una serie di **if-elseif annidati** che servono a determinare **qual è l'oggetto cliccato**.

```
function Update () {  
    // Rileva il tocco  
    if((Input.touchCount>0) && (Input.GetTouch(0).phase == TouchPhase.Ended))  
    {  
        var gameObject : GameObject =  
        GetClickedItem(Input.GetTouch(0).position);  
        if(gameObject.name == "Decrementa") contatore -= 1;  
        else if(gameObject.name == "Incrementa") contatore += 1;  
        else if(gameObject.name == "Esci") {  
            PlayerPrefs.SetInt("ValoreContatore", contatore);  
            Application.Quit();  
        }  
    }  
}
```

Come sappiamo, **Update** si aggiorna **automaticamente ad ogni frame** e ci consente, in questo modo, di verificare varie condizioni (abbiamo visto vari casi **nei tutorial precedenti**); l'**if** presente in questo blocco verifica, in particolare, **due avvenimenti**, riguardanti entrambi **Input** (un oggetto nativo di **Unity** che si riferisce ai **dispositivi di Input**): è l'interfaccia tra un mouse, una tastiera, un touchscreen o in generale qualsiasi controller e il nostro gioco.

L'attributo **touchCount** di **Input** conta il **numero di tocchi effettuati sul display** del dispositivo, se presente (notate che io sto sviluppando su pc, con mouse e tastiera, e in esecuzione l'utilizzo di **touch** non dà errore: semplicemente non fa nulla, perché si aspetta **un input di tipo touch** che, almeno sul mio pc, non c'è); con **GetTouch** recuperiamo il primo tocco, quello di **indice 0**

(**GetTouch(0)** vuol dire proprio questo), e questo perché c'è il **supporto per il multitouch** e, con **GetTouch**, possiamo anche determinare **l'ordine con il quale sono stati effettuati i tocchi**; infine, con **phase** del tocco possiamo capire se il tocco è stato **rilasciato (TouchPhase.Ended)**.

Fino a questo punto si tratta perlopiù di **oggetti e parametri di libreria standard di Unity**: l'**if** verifica la presenza di **tocchi** (con **touchCount > 0**, quindi più di un tocco) e, in **AND**, quindi entrambe le condizioni positive, che il **tocco sia finito**: se sì, esegue il codice nel blocco.

Parliamo proprio del blocco: **viene creato un oggetto al volo**, una variabile temporanea di tipo **GameObject** (anche qui, un **tipo nativo di Unity**) e viene assegnato a questo oggetto il risultato della funzione **GetClickedItem**, che vedremo tra pochissimo.

Un **GameObject** nel codice è **un oggetto di gioco a tutti gli effetti**, quindi ha anche i campi, le proprietà e i **Component** di un **Game Object** della scena, tutti accessibili via codice.

In questo particolare caso, subito dopo aver recuperato il **GameObject** cliccato, tramite **if** ed **elseif** ne controlliamo il nome, che ovviamente è il nome dato, nel nostro esempio, alle tre sfere (ecco perché nella puntata precedente ho dato nomi precisi e vi ho detto di stare attenti a maiuscole e minuscole) e, in base all'oggetto, effettua delle operazioni... questa porzione di codice è poco interessante (e qualcosina l'abbiamo già vista nel capitolo precedente), ma per completezza la riprendiamo: se l'oggetto è **Decrementa**, il **contatore** – variabile globale che teneva traccia dei tocchi – viene decrementato; viceversa con **Incrementa** viene incrementato; infine con **Esci** vengono effettuate **due azioni** (abbiamo infatti un blocco tra parentesi graffe): si aggiorna la variabile **valoreContatore** nella **memoria del dispositivo** con la funzione **SetInt** di **PlayerPrefs** e si esce dall'applicazione con **Application.Quit**.

Veniamo dunque all'argomento vero e proprio: la **funzione GetClickedItem**, che prende in input **un vettore di due coordinate**.

```
// Rileva l'elemento cliccato
function GetClickedItem(pos: Vector2) {
    var camera : Camera = Camera.main;
    var p : Vector3 = camera.ScreenToWorldPoint(Vector3 (pos[0], pos[1],
```

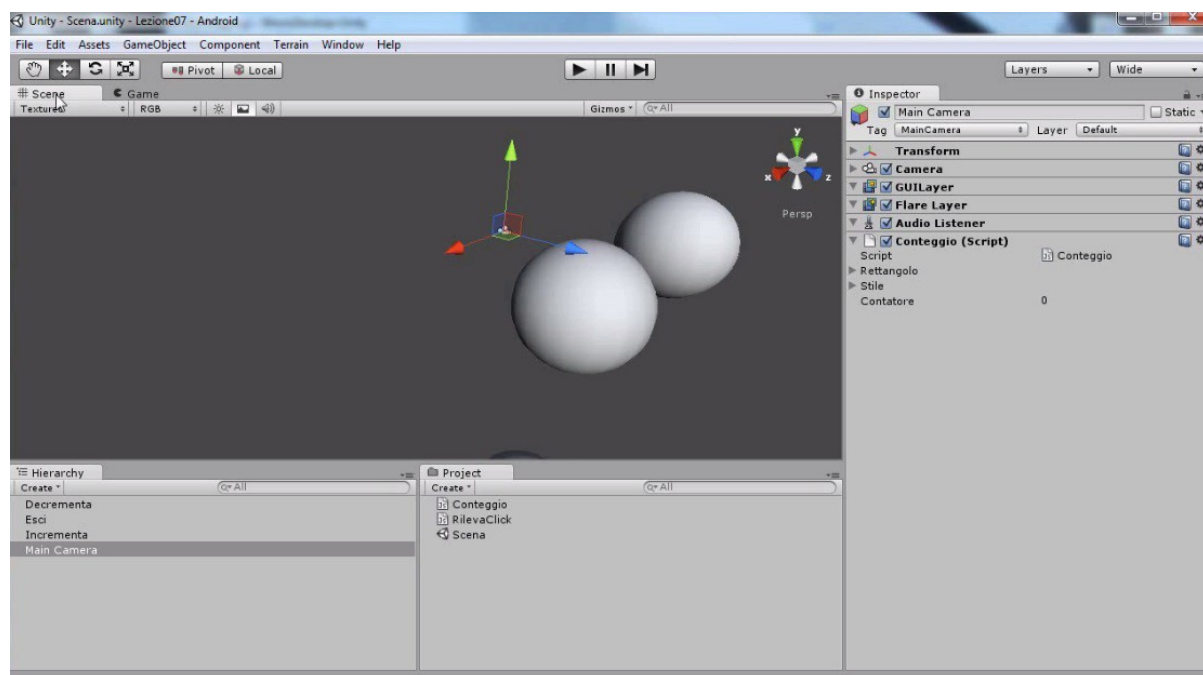
```
camera.nearClipPlane));  
var hit : RaycastHit;  
if(Physics.Raycast(p, Vector3(0.0, 0.0, 1.0), hit)) return  
hit.collider.gameObject;  
else return null;  
}
```

Tale funzione viene infatti chiamata, in **Update**, passando come parametro **Input.GetTouch(0).position**, ossia: del **primo tocco** sul dispositivo, **recupera la posizione**.

Questo campo **position**, nativo quindi di un **Touch** in **Unity**, è fatto di due valori: banalmente, le **coordinate x e y sul dispositivo**, inserite in un **unico oggetto di tipo Vector2**, un vettore di due elementi, che viene quindi passato a **GetClickedItem**.

Per spiegarvi cosa fa **GetClickedItem** non posso discutere il codice riga per riga: prima devo dirvi qual è il ragionamento che ci sta dietro, poi vediamo con quali oggetti e funzioni di libreria viene implementato.

Vediamo la **scena 3D**: il punto di vista del giocatore è dato dall'oggetto **Camera**, la **telecamera**.



Noi sappiamo che l'utente **ha toccato un punto dello schermo** e conosciamo, tramite **position**, le **coordinate XY** di tale tocco, ma queste **XY non** sono **coordinate DELLA SCENA** (dell'universo virtuale), ma dello **schermo del dispositivo**.

La prima cosa che ci serve è quindi **una funzione che “mappi” lo schermo** del dispositivo, e le sue coordinate, **sulla scena 3D** attualmente inquadrata dall'oggetto **MainCamera**, e per fortuna **Unity** gestisce tutto questo con una **semplicissima funzione**, come vedremo tra poco.

Ottenuta la mappatura, facciamo la seguente operazione: **proiettiamo un raggio virtuale**, invisibile, dal piano dell'inquadratura della telecamera in avanti, nella scena inquadrata, e **vediamo cosa colpisce**: SE colpisce qualcosa, ci facciamo dire qual è il **Game Object** (l'oggetto di scena colpito) e agiamo di conseguenza.

Tutto viene svolto **con funzioni native**, l'unico appunto da fare è che per dire alla telecamera di **proiettare il raggio “in avanti”** dobbiamo sapere qual è questo **“avanti”**, ossia se è sull'asse X, Y o Z, positivo o negativo; selezionando l'oggetto **Main Camera** nella scena, vediamo che **“in avanti” verso le sfere c'è l'asse Z positivo**, quindi il vettore del raggio sarà **0, 0, 1.0** (i primi due valori per X e Y, 1.0 positivo per dire Z positivo, secondo lo **schema XYZ**).

Vediamo quindi come tutto questo ragionamento viene implementato in termini di **codice** (riportato nelle pagine precedenti) all'interno della **funzione *GetClickedItem***.

La prima cosa da fare è creare una **variabile temporanea, d'appoggio**, per l'oggetto **telecamera**; magari la variabile in sé può essere creata a livello globale, fuori dalla funzione, ma visto che in un gioco possono esserci **più telecamere** (con la possibilità di passare da una telecamera all'altra), è consigliabile almeno **riassegnare *Camera.main*** o l'oggetto camera corrente alla variabile che, nel codice, deve rappresentare la telecamera attiva.

Come seconda variabile, creiamo un **vettore direzione tridimensionale**, chiamato qui **p** (per proiezione), realizzato mediante una **funzione nativa di Unity** che si occupa di mappare le coordinate XY del dispositivo toccato in coordinate XYZ dell'universo virtuale così come è inquadrato dalla telecamera virtuale.

Questa meravigliosa funzione (che è dell'oggetto camera, la telecamera virtuale, quindi qui va richiamata con **camera**.) si chiama **ScreenToWorldPoint**, letteralmente “**da schermo a mondo virtuale**”: da sistema di coordinate del dispositivo a universo, **scena 3D virtuale**.

I parametri sono tre ma in realtà noi ne forniamo due (le **coordinate X e Y sullo schermo**), le prime due con **pos[0]** e **pos[1]**, cioè i due valori (le coordinate X e Y) presenti nel vettore *position* fornito come parametro, e un **terzo parametro**, relativo alla distanza dalla telecamera, che è nativo di Unity: **camera.nearClipPlane**, cioè il punto dal quale **inizia la visualizzazione** della telecamera.

Faccio notare che il **vettore p** così ottenuto è solo la **DIREZIONE del raggio** che vogliamo proiettare dalla telecamera nella scena, per vedere se tocca qualcosa (ed eventualmente cosa), per cui ci mancano il **VERSO** del raggio e una **variabile-contenitore** per capire **qual è il Game Object eventualmente colpito**.

Creiamo quindi la terza e ultima variabile d'appoggio, **hit** (lett. “**colpito**”), di tipo **RaycastHit** (letteralmente “*oggetto colpito dal raggio proiettato*”, ennesimo oggetto **nativo di Unity**), quindi facciamo un controllo: “**if Physics.Raycast** [con gli argomenti che discuteremo, è vero” (ossia: **c'è un bersaglio sulla traiettoria del raggio**), “allora restituisci tale **GameObject**, passandolo quindi alla funzione che ha richiamato questo blocco in **Update**, altrimenti (se c'è stato un tocco ma non c'è alcun oggetto in quel punto dello schermo) **restituisci null, niente**”.

Physics.Raycast ha **tre parametri**: **p** (il vettore **DIREZIONE** del raggio); un vettore creato al volo che indica il **VERSO** di tale raggio (**0,0,1: Z positivo**); **hit** (la variabile-contenitore dell'oggetto **EVENTUALMENTE** colpito).

Il risultato (**l'oggetto colpito**, o **null**, in sua assenza) viene quindi **depositato proprio in hit**, che poi verrà restituito come **risultato della funzione**.

Il **return** fornisce quindi un **GameObject** alla chiamata in **Update** e poi, come abbiamo visto, **si verifica il nome di tale GameObject**, se presente, e si agisce di conseguenza. Tutto qui.

Il click sul tipo **null** non ha un **elseif** in **Update...** diciamo che viene **gestito in maniera implicita** non effettuando alcuna operazione: se la funzione restituisce **null**, gli **if-elseif** in **Update** vedono

che il suo **“nome”** non è né **Decrementa**, né **Incrementa** né **Esci** e... **non succede nulla**, visto che non esiste un **else finale generico**.

Purtroppo, non è tutto oro quel che luccica: la funzione che ho scritto **funziona bene per QUESTO scenario** in particolare, dove la **telecamera è fissa**, quindi la direzione inquadrata è sempre la stessa e la **modalità è Ortho**; in altre situazioni, potrebbero essere necessari degli **accorgimenti per proiettare bene il raggio**, comunque quanto detto è una **traccia di partenza** per i vostri esperimenti... questi problemi non riguardano invece la **gestione del click del mouse**, che è più semplice perché **Unity** rileva proprio **il click sul Game Object**, non il **position** sul dispositivo di tocco.

Ok, magari fate un paio di minuti di **pausa** prima di passare alla **seconda parte del tutorial**, che riguarda invece la **gestione del click col mouse**, per utilizzare il **gioco anche su pc**!

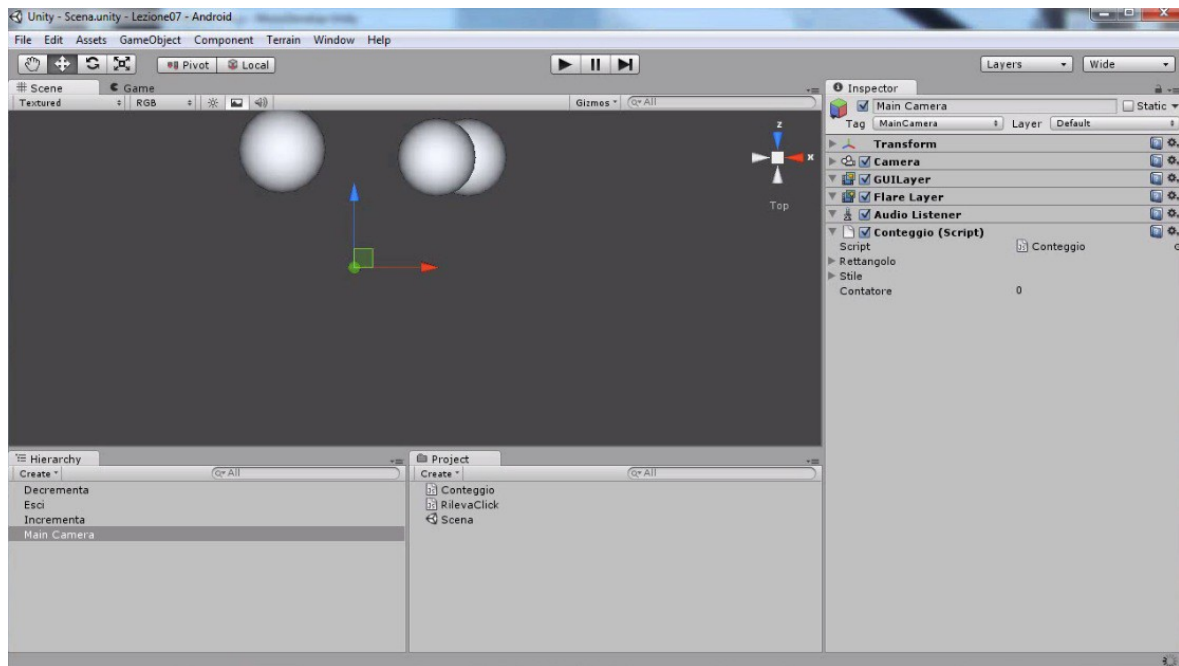
* * *

Vediamo quindi come effettuare la **gestione del click col mouse**.

In realtà, **rilevare il click del mouse** su un elemento è un'operazione di una banalità sconcertante, tuttavia l'approccio è completamente diverso da quello utilizzato finora; inoltre (e questo sì che è interessante), ci fa fare conoscenza con **GameObject.Find** e la possibilità di **recuperare e modificare campi e proprietà di ALTRI elementi**, diversi da quelli che possiedono lo script.

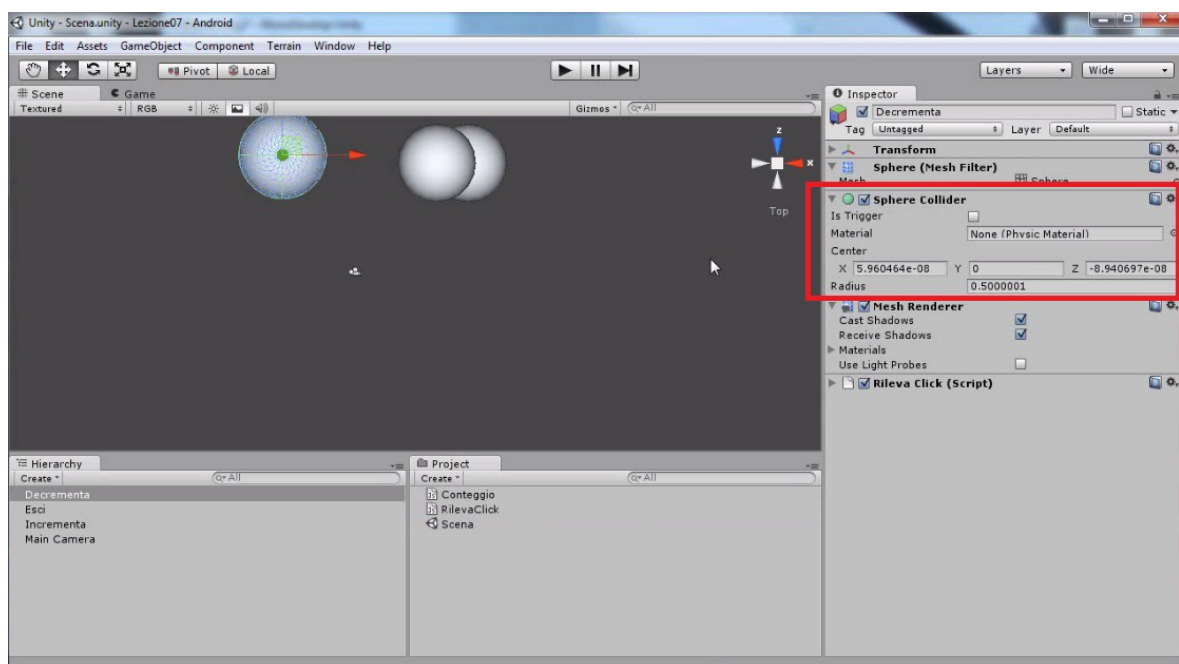
Lo **script Conteggio.js**, che contiene al suo interno **Update** col controllo sul **touch** e **GetClickedItem**, è infatti associato, nel nostro caso, alla **telecamera**; la funzione nativa che useremo per il **mouse**, invece, **rileva il click per l'oggetto che possiede lo script**.

Nel progetto, creiamo uno **script** chiamato ad esempio **“rilevaClick.js”**, quindi apriamo tale script in **MonoDevelop**.



Creiamo qui una **nuova funzione**, con un nome ben preciso: **OnMouseDown()**, senza parametri e con O, M e D maiuscole: si tratta, infatti, di una **funzione nativa di Unity**, quindi dobbiamo essere precisi.

NOTA: questa funzione... funziona solo se l'oggetto è un **COLLIDER**, come lo sono di default le **mesh**, incluse le tre sfere che abbiamo nella scena, come possiamo verificare **nell'Inspector**.



NOTA: **OnMouseDown** non funziona con **iPhone** o dispositivi **touch** (d'altro canto, **touch** non funziona col **mouse**).

Creiamo quindi la **funzione** **OnMouseDown()**, in spazio globale, nello script “**RilevaClick.js**”:

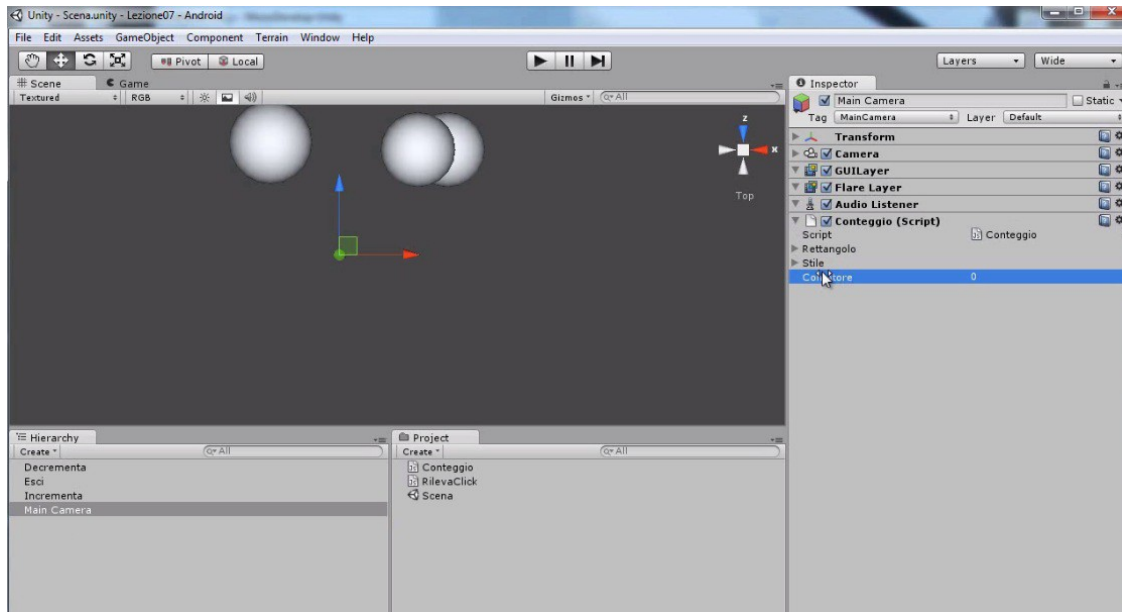
```
if((this.name == "Incrementa"))
(GameObject.Find("Main Camera").GetComponent(Conteggio)).contatore += 1;
else if((this.name == "Decrementa"))
(GameObject.Find("Main Camera").GetComponent(Conteggio)).contatore -= 1;
else if((this.name == "Esci")) {
    PlayerPrefs.SetInt("ValoreContatore", (GameObject.Find("Main
    Camera").GetComponent(Conteggio)).contatore);
    Application.Quit();
}
```

Per prima cosa, parliamo di **this**: è una **parola chiave**, che significa letteralmente “**questo**”, e che sta a indicare, appunto, **l'oggetto che ha chiamato lo script**.

La parola chiave **this** è di fondamentale importanza in caso di **script condivisi da più elementi**, come questo, perché ci aiuta a dire: “*se QUESTO oggetto, quello cliccato, si chiama Incrementa, allora fai questa operazione, altrimenti eccetera eccetera*”.

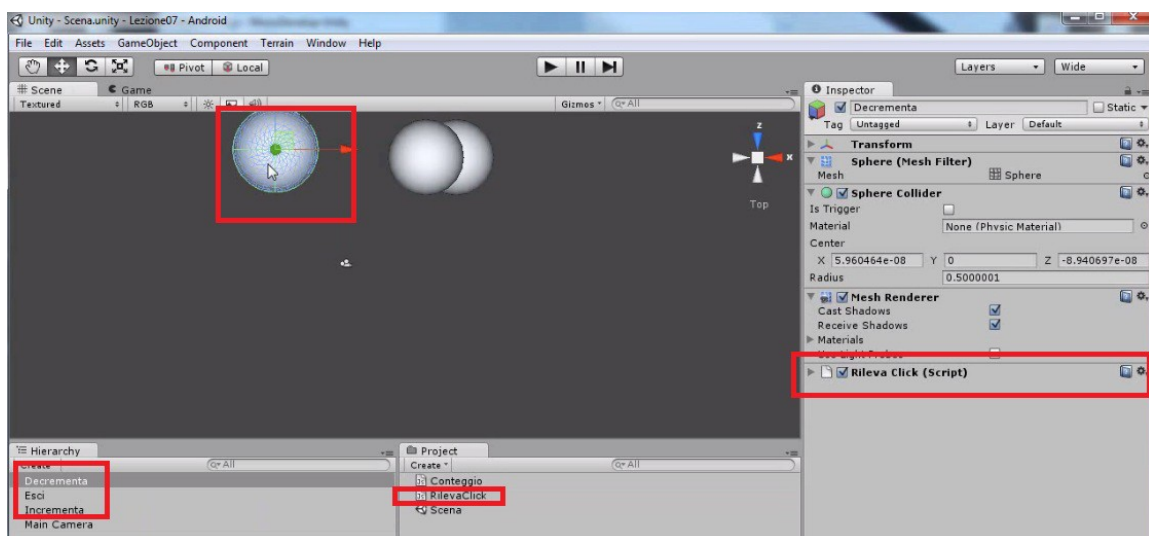
La parte interessante è data comunque da **GameObject.Find**: all'interno della funzione **Find**, scriviamo il nome dell'oggetto che vogliamo recuperare, in questo caso **Main Camera**; poi, su tale elemento, facciamo **.GetComponent** e, tra parentesi e senza doppi apici, scriviamo **il nome del Component associato all'oggetto recuperato**, nel nostro caso lo script **Conteggio**, associato a **Main Camera**; infine, di tale componente facciamo **.contatore** per accedere alla **variabile contatore dello script**.

Ricapitolando: stiamo prelevando, dalla scena, l'oggetto di nome **Main Camera**, con **Find**, quindi stiamo accedendo al suo componente **Conteggio** (uno **script**), e in questo componente stiamo accedendo addirittura a una **variabile**, **contatore**, che poi incrementeremo, decremteremo o semplicemente leggeremo, a seconda dei casi, per **Incrementa**, **Decrementa** ed **Esci**!

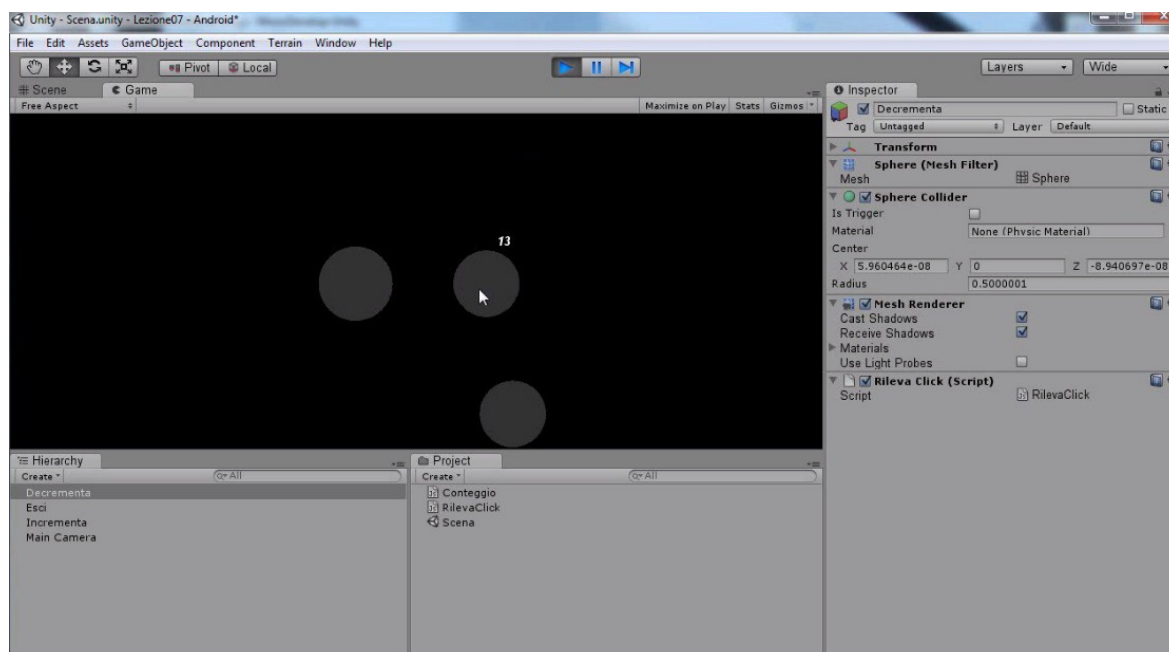


È possibile, quindi, **accedere agli elementi, alle componenti o addirittura alle variabili di qualsiasi oggetto di scena** da parte di uno script di un altro oggetto.

NOTA spiacevole numero 1: lo **script** appena realizzato **NON** va associato all'oggetto **Main Camera** ma va aggiunto, come **Component**, a **tutti gli oggetti cliccabili**; è vero che, tramite **this**, possiamo capire qual è l'oggetto cliccato, ma non possiamo fare come con **Conteggio**, che andava associato alla **Telecamera** e **rilevava il tocco per tutti** (perché **in realtà** non capiva qual era l'oggetto toccato, ma **proiettava un raggio nella scena**); qui, l'**oggetto cliccato** deve **possedere lo script** che rileva l'evento **OnMouseDown**, altrimenti **non se ne accorge**, quindi dobbiamo trascinare lo script **RilevaConteggio** sulle tre sfere nella **Gerarchia** o nella **scena 3D**.



NOTA spiacevole numero 2: Application.Quit non funzionerà in **modalità Demo**, dentro l'editor, come in questo caso, ma in caso di **Build dell'applicazione per pc o Mac** dovrebbe funzionare a dovere.



* * *

Tips & Tricks: Coroutine (script Javascript)

Questo **breve tutorial sulla programmazione in Unity 3D** nasce da una **richiesta di un utente del Forum** che, nella **sezione Unity 3D**, appunto, chiedeva come fare per **far apparire istanze di un oggetto a intervalli di 3 secondi**; tali oggetti **devono anche cadere verso il basso**, quindi devono essere provvisti di **Component Rigidbody**.

Ho deciso di fare il **tutorial** perché la soluzione che ho adottato prevede l'utilizzo di una **Coroutine**, argomento interessante perché si presta a **diverse applicazioni**.

Per questo esempio userò delle **semplici primitive Sphere** (mostrando tra l'altro **come crearle via codice**); a livello visivo, quindi, la scena è davvero triste, tuttavia in un **prototipo** – come questo – non è importante creare già i modelli finali, ma **utilizzare delle primitive come “segnaposto” e concentrarsi sulle tecniche**.

La soluzione prevista da *Chucky831* (l'utente del Forum *effe12.com*) faceva uso di alcune istruzioni inserite nel **metodo Update**; il problema è che questo metodo viene **aggiornato ad ogni frame** e mettere un **WaitForSeconds** non risolve il problema, in quanto si avrebbe comunque **una chiamata per frame**.

La soluzione sta nell'utilizzare una **Coroutine**.

“routine” è un sinonimo di procedura o **funzione**, quindi una sequenza di istruzioni; **“coroutine”** sta quindi per *routine da eseguire contemporaneamente* alle altre, in concomitanza.

Per chi ha già delle conoscenze di **informatica di base** dico solo una parola che dovrebbe rendere tutto chiaro: **multithread**; per tutti gli altri, ecco una spiegazione di quello che avviene: in ogni programma c'è un **“filo di esecuzione” principale**, ma generalmente ce ne sono altri che vengono eseguiti **in parallelo**: nei sistemi a **singolo processore**, si interrompe momentaneamente

l'esecuzione di un “**filo**” (in Inglese, **thread**) e si passa all'altro, e così via per tutti i **thread** previsti, solo che generalmente questa operazione viene eseguita molto velocemente e noi non ce ne accorgiamo, comunque questa è un'esecuzione in **PSEUDO parallelo** perché in realtà c'è un solo processore che esegue vari thread sempre in sequenza, ma intervallandoli, quindi **singolo processore – multithread**.

Nei sistemi **multiprocessore**, invece, c'è il **vero parallelismo** perché lì ogni processore può seguire **uno o più thread**, quindi abbiamo **multiprocessore single thread** o, ancora meglio, **multiprocessore multithread**.

Questo concetto del **multithread** è di fondamentale importanza nello sviluppo dei **videogiochi**: se ci pensate, tutti i **Game Object** provvisti di **script con Update** hanno delle **routine** da eseguire “**in parallelo**” **ad ogni frame**, inoltre il sistema deve gestire, ad esempio, gli effetti sonori, le comunicazioni con la scheda di rete, la memoria ram, l'input che gli date con mouse o touchscreen, e così via... il tutto frame per frame, **in parallelo** (... o quasi).

Anche lo **script che crea le sfere** in posizioni random ha un suo **thread** principale, dato da **Update** che si aggiorna automaticamente ad ogni frame; noi vogliamo creare **un secondo thread**, svincolato da **Update**, da gestire a piacere ma da **eseguire in parallelo ad Update**, ed ecco che entra in gioco la chiamata **StartCoroutine**, nativa di **Unity 3D**.

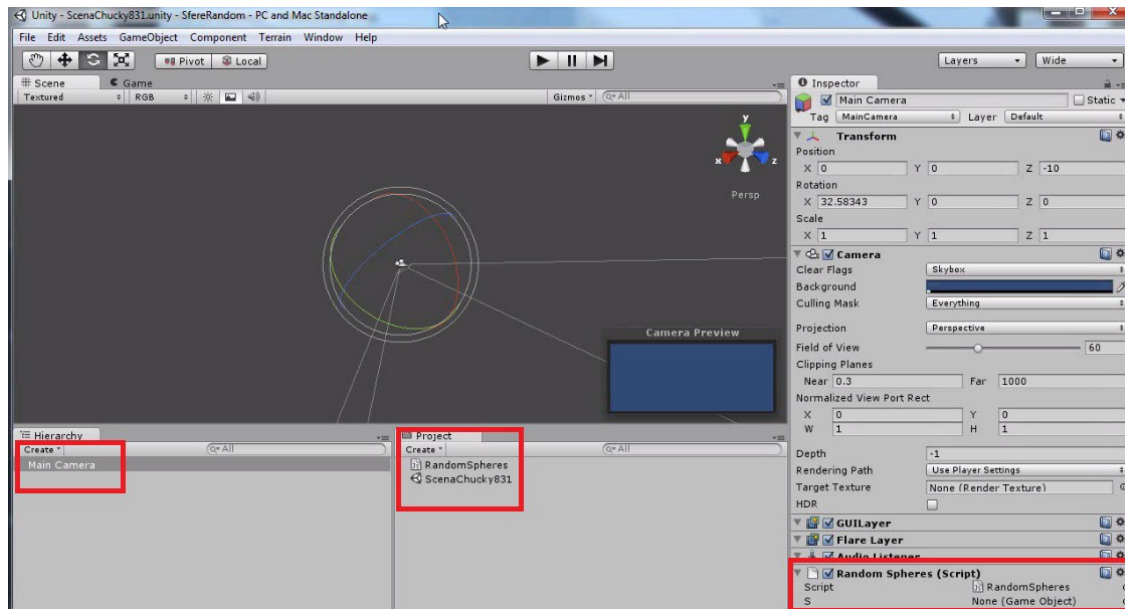
StartCoroutine prende come parametro **una funzione dello script** e la esegue in parallelo.

Esistono poi funzioni per **mettere in pausa o terminare le Coroutine**, comunque in questo tutorial non le esamineremo: vi sto dando l'input per approfondire con la meravigliosa **documentazione online di Unity 3D**, partendo da **Coroutine**.

Altro problema: il primo script preso in esame da *Chucky* spostava, al termine dell'animazione, la sfera nella **posizione random iniziale**, quindi la lasciava cadere di nuovo; la soluzione qui adottata crea invece **istanze di sfere random diverse**, mediante la creazione – via codice – di oggetti **Game Object**.

Passiamo alla **pratica** e all'esame del codice.

Creiamo uno **script JavaScript** da associare, ad esempio, all'oggetto **Main Camera**, quindi apriamo tale script in **MonoDevelop**.



All'inizio, nello spazio globale, creiamo una **variabile di tipo *GameObject***, oggetto di gioco, chiamata **s** (per sfera), con:

```
var s : GameObject;
```

Nella funzione **Start**, scriviamo:

```
StartCoroutine(creazione(3));
```

Attenzione: il parametro di **StartCoroutine** è un **nome di funzione** (che stiamo per definire) **senza doppi apici e con le parentesi tonde** (con, all'interno, eventuali parametri... qui, giusto per fare un esempio, sto mettendo il **numero di secondi** tra un'emissione e l'altra, 3).

Fuori da qualsiasi funzione, scriviamo:

```
function creazione(tempo) {  
    while(true) {  
        s = GameObject.CreatePrimitive(PrimitiveType.Sphere);  
        s.AddComponent(Rigidbody);  
        s.transform.position.y = 0;  
        s.transform.position.x = Random.Range(-10, 10);
```

```
        yield WaitForSeconds (tempo) ;  
    }  
}
```

La **funzione** prende quindi in input un **parametro**, da noi chiamato “**tempo**”.

Dentro la funzione, abbiamo un **blocco *while(true)***, che contiene le istruzioni da eseguire; col “**while(true)**” (letteralmente, “finché è vero”, ossia “**sempre**”, a meno di non interrompere il ciclo in qualche modo), eseguiremo sempre, in continuazione, le istruzioni contenute nel blocco... inclusa **l'istruzione di ritardo tra un ciclo e l'altro**.

La **prima riga** crea una **nuova istanza di una primitiva** di tipo sfera, assegnando il risultato di questa creazione alla **variabile *s*** (quindi ad ogni esecuzione noi ricicliamo la variabile **s**... la variabile, non le sfere: ogni sfera inserita - a meno di non effettuare altre operazioni non previste in questo tutorial - **resterà nella scena**, appesantendo la memoria... ma questo è un altro discorso e non lo tratteremo qui).

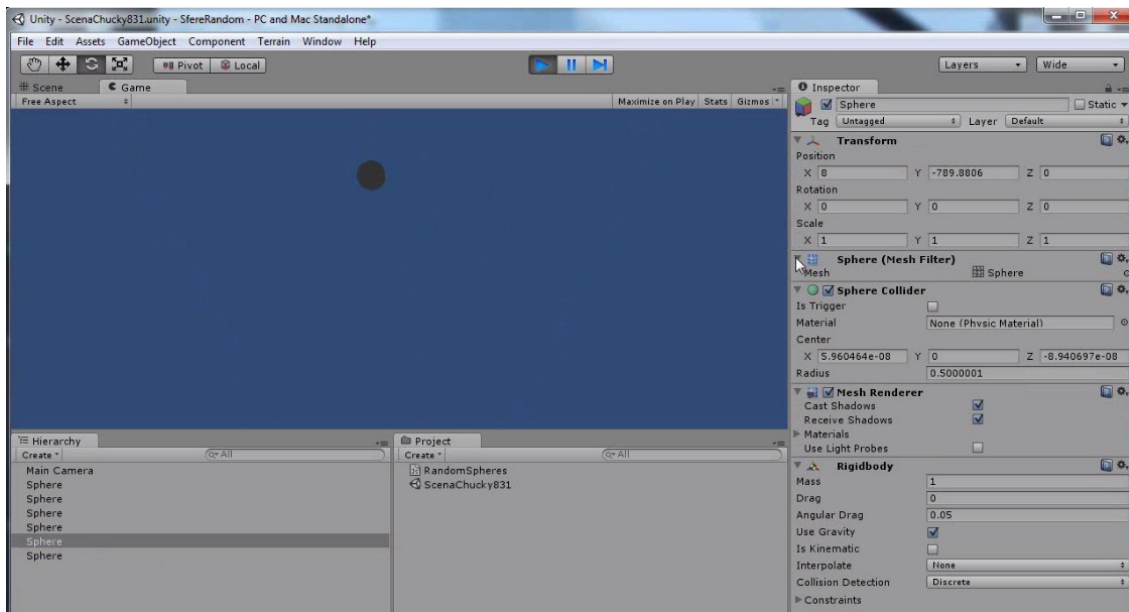
La **seconda riga** aggiunge un **Component** di tipo **RigidBody** all'oggetto appena istanziato; questo componente **serve a far cadere l'oggetto**, come visto nel **sesto capitolo del corso base su Unity 3D**.

La **terza e la quarta riga** posizionano l'oggetto: la **posizione verticale** è 0, la stessa della telecamera, ma quella orizzontale è determinata in maniera **casuale**, al volo, mediante una chiamata nativa **Random.Range**, che prende **due parametri** e che restituisce un **valore casuale (*Random*)** tra i due **estremi (*Range*)** passati come parametri, quindi **X** può assumere, in questo caso, un **valore tra -10 e 10**.

La **quinta istruzione** del blocco è **yield WaitForSeconds**, con parametro “**tempo**” - quello passato alla funzione **creazione** – che **ritarda l'esecuzione** del ciclo *while* successivo di un numero di **secondi** determinato appunto dal valore di “**tempo**”, che abbiamo deciso di definire esternamente, nella **funzione *Start*** (chiamata automaticamente da **Unity** all'avvio dello script, una volta sola).

Da notare che **non stiamo scrivendo alcunché in “Update”**, comunque lì possiamo metterci **quello che vogliamo** (ad esempio, rilevare la pressione di un tasto e **gestire l'interruzione della coroutine** della funzione **creazione**), oppure possiamo aggiungere **una funzione OnGUI** per gestire delle scritte sul display... in effetti, chi ha seguito i capitoli del corso sulle basi di Unity 3D ha già visto una **coroutine**, ossia **OnGUI**, che viene eseguita da **Unity in parallelo con Update**.

Mandiamo in **esecuzione** il gioco per osservare il risultato.



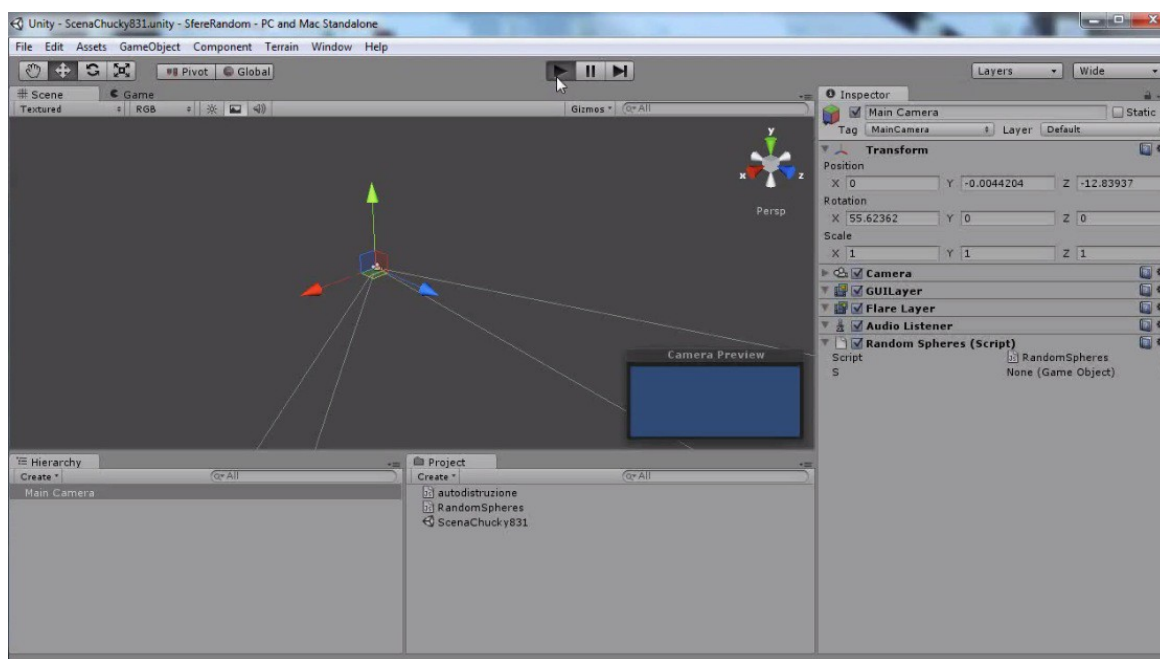
A questo punto, un **piccolo esercizio**: trovare un modo per **togliere dalla scena di gioco le sfere** create dopo, ad esempio, 2 secondi dalla loro creazione, **solo mediante script**, senza collisioni con oggetti o altro.

Le sfere create restano infatti nel gioco e **occupano memoria**, continuando a cadere all'infinito; è un problema che è stato notato **già nel sesto capitolo del corso** da alcuni utenti... vedremo comunque una possibile soluzione tra qualche capitolo.

Script JS: eliminare Game Objects dopo n secondi

Questa è, in un certo senso, la seconda parte del capitolo precedente, che riguardava l'inserimento di oggetti in posizioni quasi random nella scena di **Unity 3D**.

Lo **script** proposto come soluzione, in quel **tutorial**, istanziava **copie di un Prefab** e, dotandole di un **Component Rigidbody**, le lasciava cadere nell'universo virtuale.



C'era, però, un **problema**: gli oggetti così inseriti nella scena **vi restavano all'infinito** (come si poteva constatare guardando il **pannello Gerarchia** in fase di esecuzione del demo), **sprecando memoria e calcoli inutilmente**.

Nel **tips & tricks** precedente, quindi, vi ho lasciato come esercizio quello di **trovare una soluzione al problema**, ossia un modo per far sparire gli oggetti, in questo caso **dopo un certo numero di**

secondi, non in seguito a collisioni con altri elementi o posizioni nello spazio virtuale.

Una possibile soluzione consiste nel **creare un nuovo script**, ad esempio “**autodistruzione**”, con all'interno, nel metodo **Start** (che viene eseguito automaticamente **all'avvio dello script**, quando l'oggetto viene istanziato) **una sola riga di codice**:

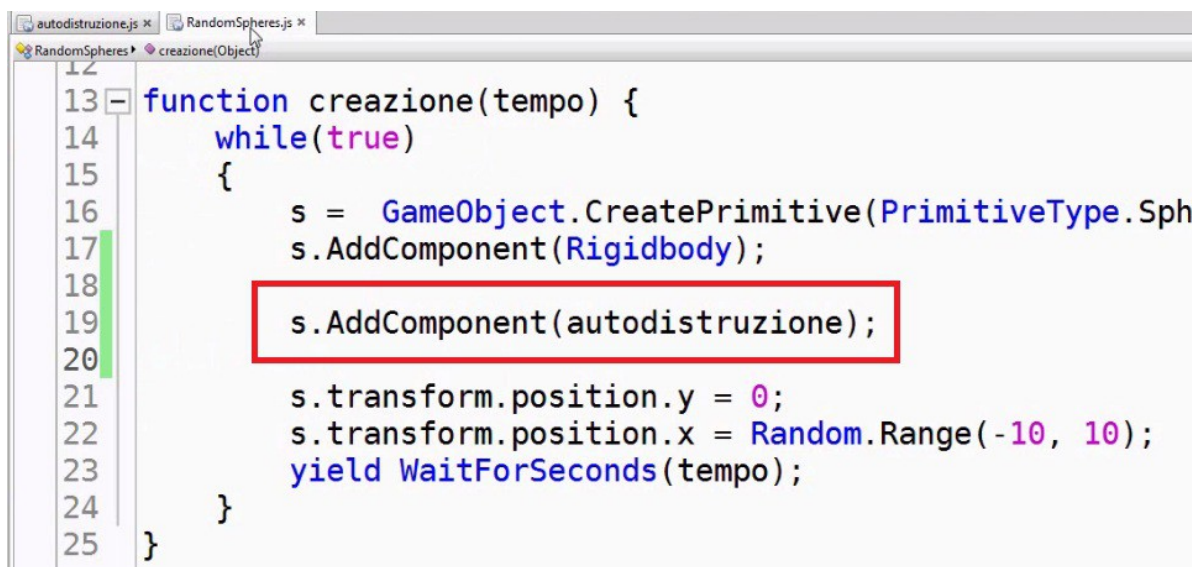
```
Destroy(gameObject, 5);
```

con la D di Destroy maiuscola.

Si tratta di una funzione nativa di **Unity** che prende in input **due parametri**: il primo, in questo caso, è “**gameObject**”, con g minuscolo e O maiuscolo, che indica che **l'elemento che vogliamo distruggere è proprio il *Game Object***, l'oggetto di scena, **che possiede lo script** (si può decidere, infatti, di distruggere elementi di altri tipi, ad esempio per rimuovere un **Component** da un **Game Object**), mentre il secondo parametro indica i **secondi** dopo i quali, partendo dalla chiamata alla funzione, **distruggere l'elemento**, in questo caso 5 secondi.

Questo è quindi uno **script** a parte, non legato (ancora) ad alcun elemento... per associarlo alle **sfe**re istanziate di volta in volta, dobbiamo tornare nello **script di creazione degli oggetti**, lì indicati con la variabile **s**, ed aggiungere la seguente istruzione subito dopo **s.AddComponent(Rigidbody)**:

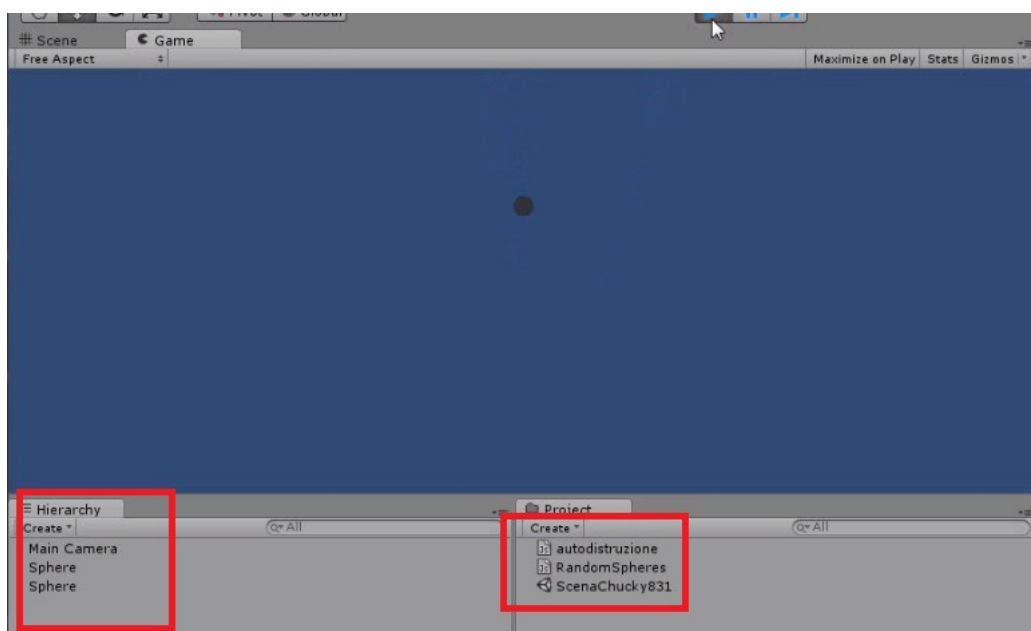
```
s.AddComponent(autodistruzione);
```



In questo modo, **quando creiamo un'istanza dell'oggetto**, oltre a darle il **componente *Rigid Body*** (come visto nel capitolo precedente), le associamo anche lo **script *autodistruzione*** e, tra l'altro, non appena effettuiamo questa operazione, **per ciascuna sfera viene avviato il metodo *Start* dello script “autodistruzione”** e, quindi, il timer per distruggere l'oggetto di gioco.

L'utilizzo di uno **script esterno** ha un vantaggio: è una soluzione **più modulare e riutilizzabile**, infatti se in un gioco potete inserire **più elementi** potete associare questo **script** a tutti questi elementi, senza preoccuparvi della loro natura e **senza andare a modificare i loro script**; inoltre, se in seguito volete aggiungere **eventi extra**, come ad esempio l'esecuzione di un **file audio** o la scrittura di qualcosa a video (ad esempio, **il conto alla rovescia**, ad esempio per un avviso di granata nemica e così via), vi basterà **modificare una volta lo script “autodistruzione”** per ritrovare tutte le modifiche in **tutti i *Game Object* che ne fanno uso** (di più: potete personalizzare comunque lo **script** in base all'oggetto che lo possiede mediante l'utilizzo di **this**, come visto all'interno della funzione **OnMouseDown** nell'ottavo capitolo del **corso di base su Unity**... come vedete, **tutto è collegato**, certi aspetti o funzioni visti in particolari ambiti possono tornare utili in momenti diversi).

Mandando in **esecuzione il gioco** (ovviamente dopo aver **salvato le modifiche agli script**, per renderle effettive), possiamo constatare, osservando il **pannello *Gerarchia***, che **gli oggetti creati spariscono dopo un certo numero di secondi** dalla scena, proprio come volevamo.



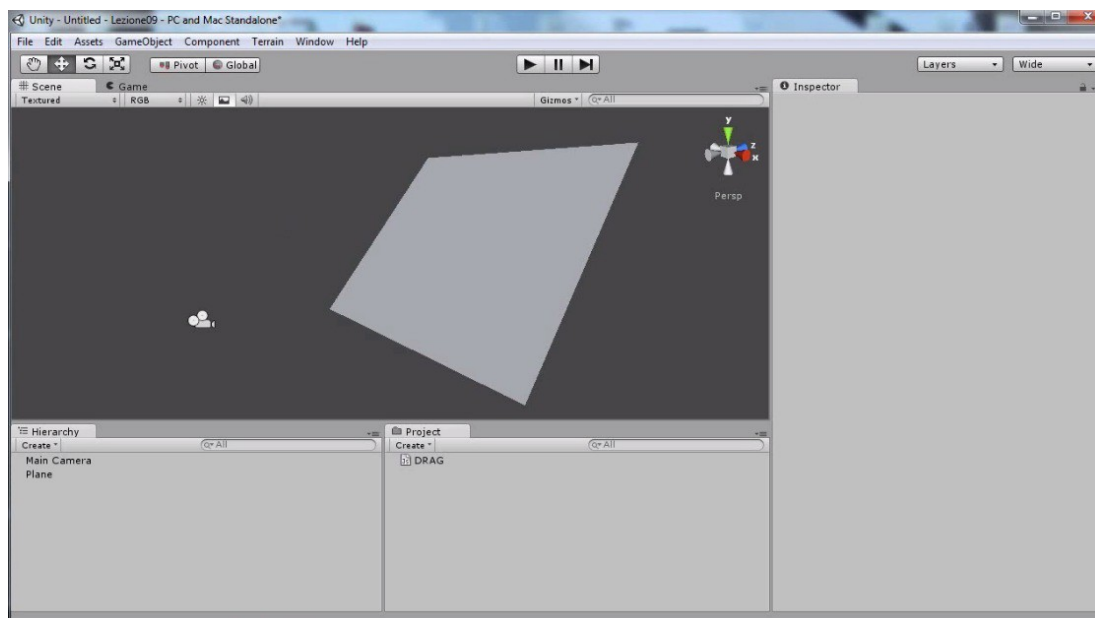
* * *

Le basi di Unity 3D 3.x e 4.x – 9: funzioni mouse buttons; Materials, Shaders, Textures

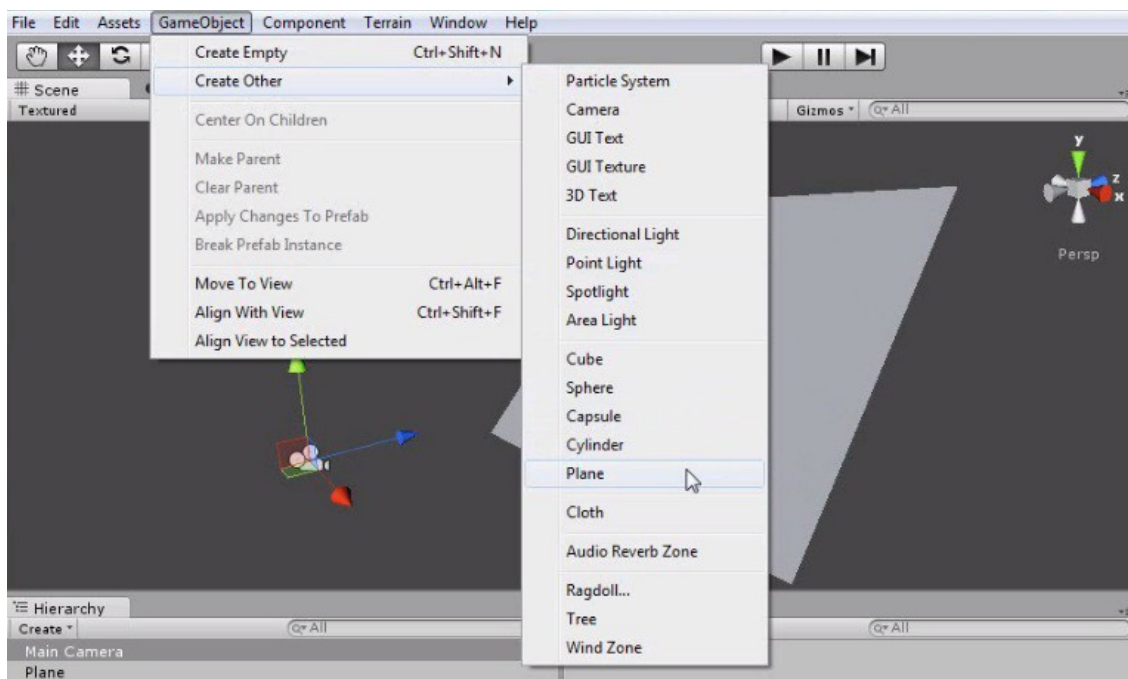
In questo nono capitolo di introduzione a **Unity 3D** vedremo un paio di funzioni riguardanti il **mouse**, come il **DRAG** (trascinamento), più qualche concetto su **Materials** e **Textures**, con un occhio all'ottimizzazione nell'uso delle risorse.

Iniziamo col primo argomento: altre funzioni, rispetto a quelle viste nel capitolo precedente, per **l'input del mouse**.

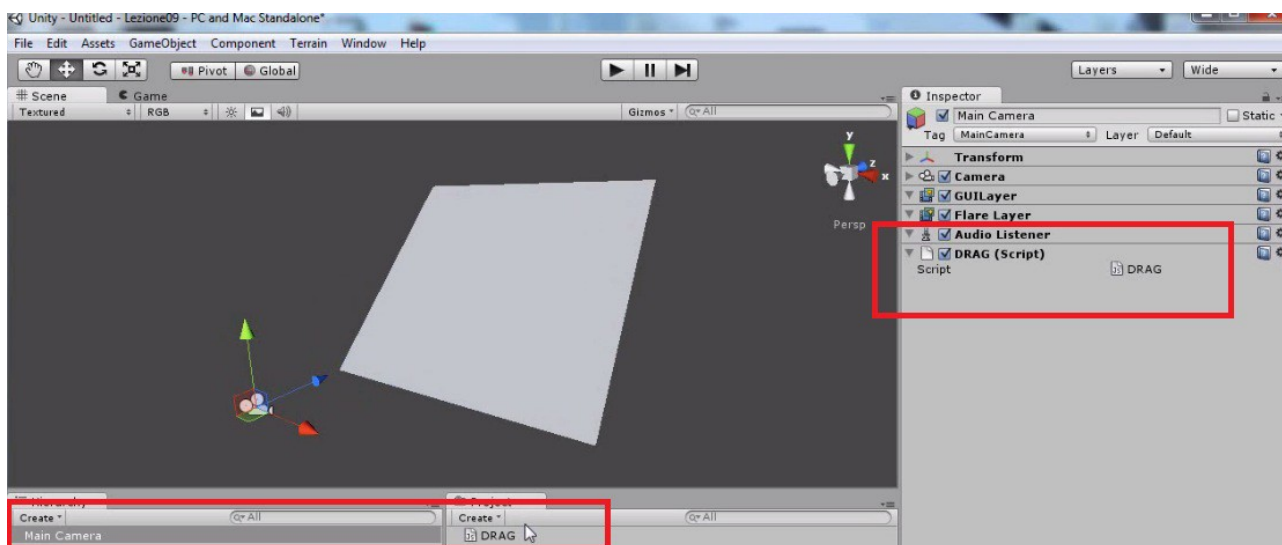
Nella scena utilizzata in questo esempio è presente un semplice **Plane**, una primitiva geometrica creata mediante **Game Object – Create Other – Plane**.



Creiamo uno **script Javascript**, chiamandolo ad esempio **DRAG**, quindi associamolo sin da subito **alla Main Camera** del progetto, non al **Plane**, poi apriamo **MonoDevelop** per inserire le istruzioni di codice.



Per prima cosa, vediamo come **proiettare un raggio ad un click del mouse** sulla scena e vedere se colpisce qualcosa (ed eventualmente cosa): stiamo quindi per realizzare un ibrido tra la **gestione del touch su dispositivo**, dove si proiettava un raggio dopo aver convertito le coordinate di schermo in **coordinate World**, e la **gestione del click del mouse**, implementata finora solo *per oggetto*, argomenti che abbiamo trattato nel capitolo precedente.



Con la **proiezione del raggio** anche col **click del mouse** non abbiamo più bisogno di dotare gli oggetti bersaglio di uno **script di ricezione** e possiamo selezionare e **manipolare qualsiasi oggetto** (purché dotato di **Component Collider**, come di default); ovviamente, non è sempre necessario selezionare qualsiasi oggetto, anzi a volte non è proprio desiderabile: ecco perché nel capitolo precedente abbiamo esaminato la soluzione con la **funzione OnMouseDown** da associare agli elementi cliccabili... comunque, procediamo con questa nuova soluzione.

All'interno della **funzione Update** scriviamo le seguenti istruzioni:

```
if (Input.GetButtonDown("Fire1")) {  
    var camera : Camera = Camera.main;  
    var p : Vector3 = camera.ScreenToWorldPoint(Vector3  
        (Input.mousePosition[0], Input.mousePosition[1],  
        camera.nearClipPlane));  
    var hit : RaycastHit;  
    if (Physics.Raycast(p, Vector3(0.0, 0.0, 1.0), hit))  
        hit.transform.position.x = p[0];  
}
```

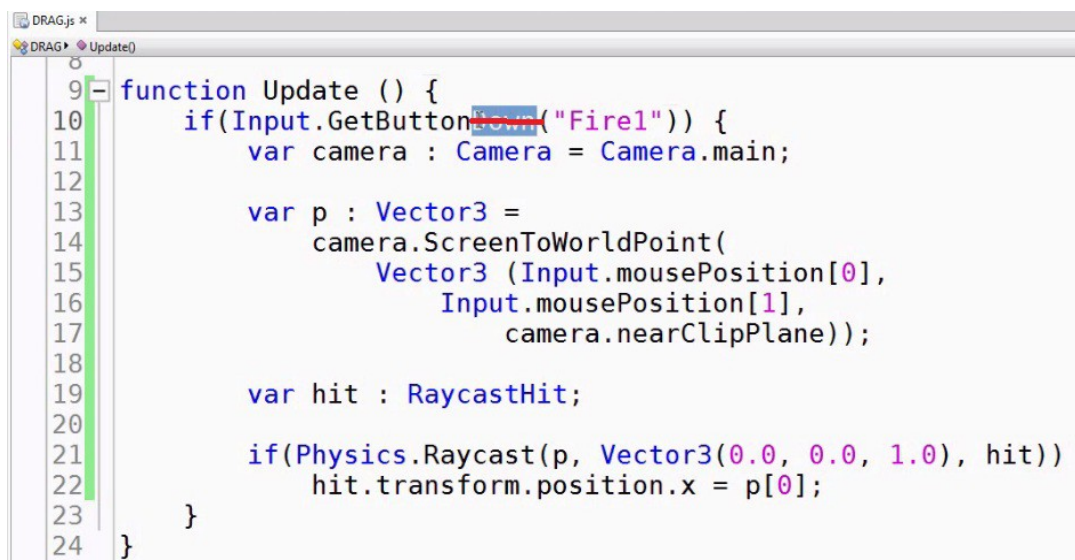
In pratica, tutto molto simile a quello visto nel capitolo (della guida di base) precedente, tranne per il fatto che, in **ScreenToWorldPoint**, anziché **pos[0]** e **pos[1]** rilevate dal tocco stiamo passando direttamente **Input.mousePosition[0]** e **[1]**, ossia le coordinate x e y del mouse sullo schermo, rilevate direttamente da **Input**; inoltre, come evento da effettuare una volta **rilevata la collisione** e recuperato l'**oggetto hit colpito**, stiamo cambiando le coordinate di tale oggetto (quindi **hit** si riferirà in generale all'oggetto colpito nella scena) con **hit.transform.position.x** assegnando a tale parametro (che riguarda il solo spostamento sull'asse x, cioè quello orizzontale) la posizione del mouse convertita in posizione nella scena (operazione già effettuata per noi da **ScreenToWorld**, per cui riusciamo la variabile **p**, prendendone il primo parametro, relativo all'**asse X**, con **p[0]**).

Mandando in esecuzione, notiamo comunque che il **Plane** si sposta quando facciamo click; tuttavia, si sposta solo per il click, senza trascinamento.

Ciò è dovuto al fatto che la **funzione GetButtonDown** restituisce, come dice la **documentazione online di Unity** per l'oggetto **Input**, valore **true** solo per il PRIMO frame della pressione del pulsante (NOTA a margine: “**Fire1**” è una **parola chiave di Unity** che identifica il **fuoco primario**

nei vari dispositivi come mouse o altri controller; nel mouse, è il pulsante sinistro); noi invece cerchiamo una condizione che sia **true** per tutto il tempo in cui il primo pulsante resta premuto, in modo da fare il trascinamento...

... e, sempre nella pagina **Input** della documentazione, proprio sopra **GetButtonDown**, troviamo **GetButton** che, cito testualmente, *“restituisce true mentre il pulsante identificato dalla stringa argomento resta premuto”*, quindi nello script sostituiamo **GetButtonDown** con **GetButton** nell'if e mandiamo in esecuzione: abbiamo realizzato il **trascinamento dell'oggetto sull'asse X** (ma passarlo anche sull'asse Y è facilissimo) **mediante click del mouse** con uno script associato alla telecamera, non al singolo oggetto cliccato, quindi **valido per qualsiasi oggetto di scena** senza dovergli associare alcuno **script**.



```
1  DRAG.js
2  DRAG Update()
3
4  9 function Update () {
5    10     if(Input.GetButton("Fire1")) {
6      11         var camera : Camera = Camera.main;
7       12
8       13         var p : Vector3 =
9         14             camera.ScreenToWorldPoint(
10          15                 Vector3 (Input.mousePosition[0],
11          16                     Input.mousePosition[1],
12          17                     camera.nearClipPlane));
13         18
14         19         var hit : RaycastHit;
15         20
16         21         if(Physics.Raycast(p, Vector3(0.0, 0.0, 1.0), hit))
17         22             hit.transform.position.x = p[0];
18         23     }
19         24 }
```

* * *

Per parlare, invece, dei **Materials** e delle **Textures** con **trasparenza**, dobbiamo fare prima un discorso teorico-introdotivo e discutere della distinzione tra **Materials (materiali)** e **Shaders (ombreggiatori)**.

In **Unity**, uno **Shader** è un blocco di **informazioni** e codice che definisce le proprietà e gli elementi da utilizzare per definire l'aspetto (illuminazione e ombreggiatura, caratteristiche "fisiche" dell'aspetto visivo di un elemento) di un oggetto.

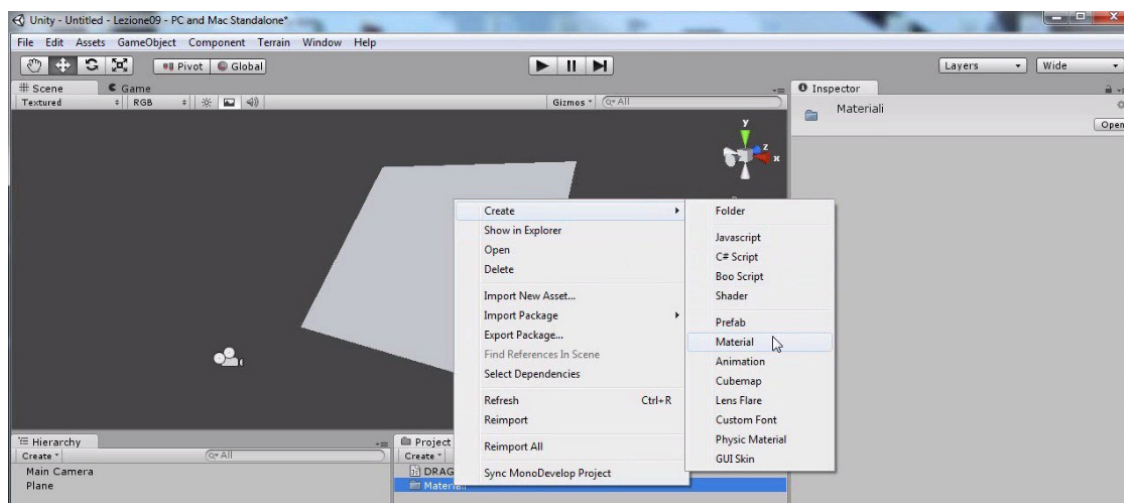
I **Materials** rappresentano, in un certo senso, l'**interfaccia** tra le proprietà e gli assets (elementi) da assegnare e i relativi campi e parametri dello **Shader**.

L'implementazione di **uno Shader per un oggetto di gioco** avviene, quindi, **mediante l'utilizzo di un Material**, oggetto del progetto che viene associato a **uno o più Game Objects**.

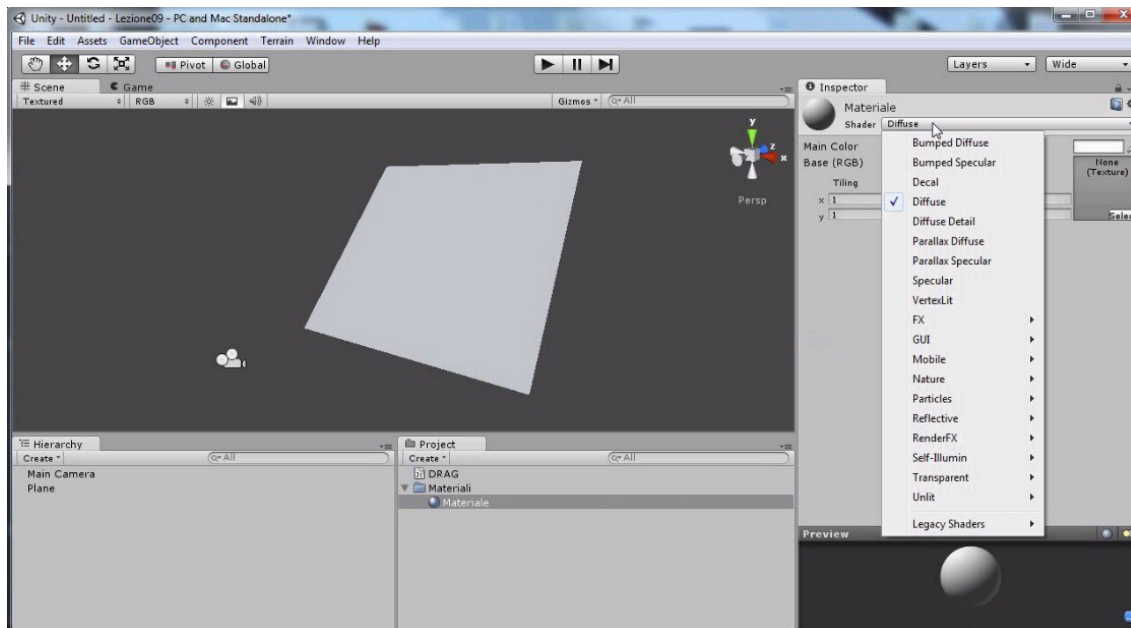
Facciamo subito un esempio pratico per capire meglio questa relazione tra **Shaders** e **Materials** e come legare il tutto a un **Game Object**.

Nella finestra **Project**, creiamo una cartella con nome **“Materiali”**, dentro la quale inserire appunto i **Materials** del nostro progetto.

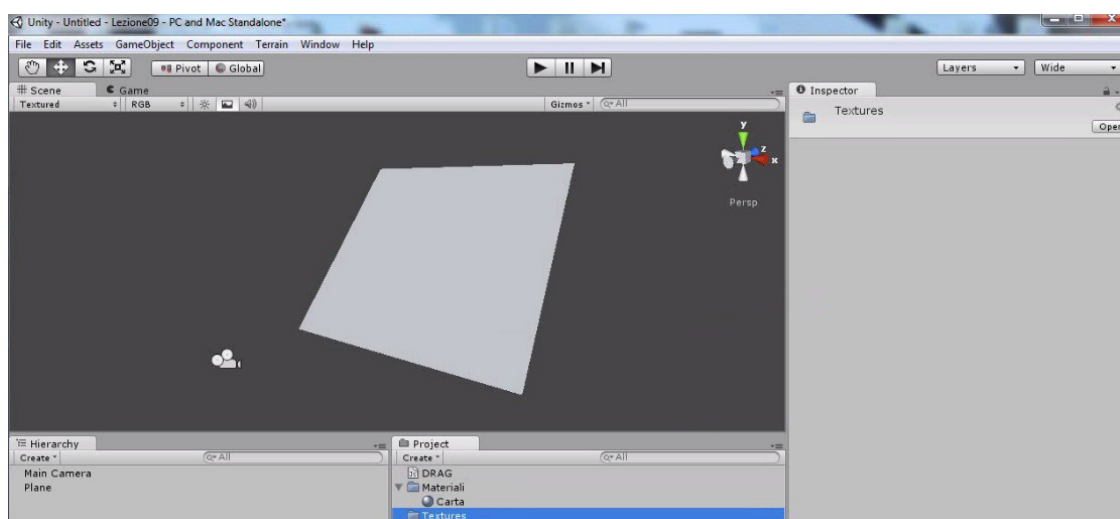
All'interno, creiamo un **Asset** di tipo **Material**, che quindi è un elemento del progetto a tutti gli effetti.



Esaminiamo l'**Inspector** per tale elemento, notando soprattutto la prima voce in alto: **Shader**, associata a un menù a tendina che ci mette a disposizione molte voci, alcune dai significati abbastanza intuitivi.

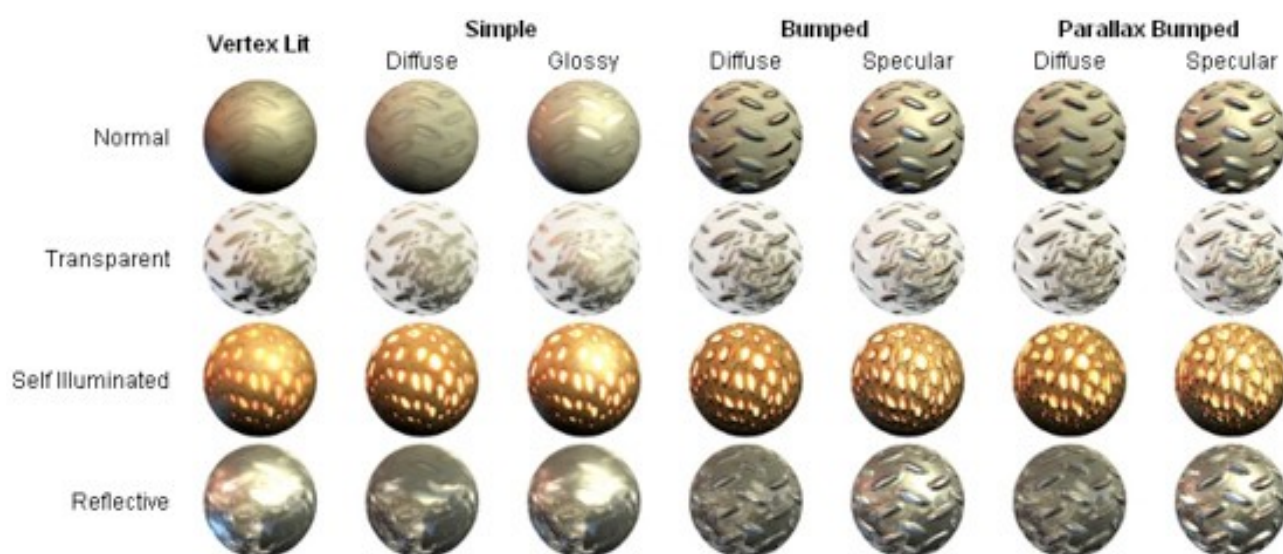


Ecco quindi l'**associazione tra informazioni e codice** (racchiuse in uno **Shader**) e l'elemento di gioco (il **Material**): quando associamo questo **Material** ad un oggetto di gioco, gli associamo anche la “natura fisica” rappresentata dallo **Shader** scelto... ma non solo: un **Material** può avere caratteristiche diverse, come ad esempio **un colore o una Texture**, comunque questi legami possono essere **cambiati via codice in esecuzione**, quindi un **Material** con **Textures** o colori diversi, ma “natura di fondo” - per così dire - identica, può essere condiviso da **più Game Objects**, con un notevole risparmio di risorse.



In questo **tutorial** non esamineremo in dettaglio TUTTE le voci presenti nel menù **Shader**, anche perché un elenco del genere risulterebbe noioso e poco utile a livello pratico; ci concentreremo, invece, sugli **esempi tipici** o **“famiglie di Shaders”** (che sono solo quattro), che poi si combinano con la natura **Diffuse**, cioè generalmente **opaca senza riflessi speculari** (come cemento o carta), e **Glossy/Specular**, che invece è **molto riflettente** (simile al metallo cromato, all'acciaio, eccetera).

L'immagine seguente è tratta dalla pagina **Materials/Shaders** della **documentazione ufficiale di Unity**; esaminiamola.



Lo **Shader Normal** è quello “classico”, il più comune, ossia quello da dare a oggetti e superfici che non presentano particolari effetti come riflessione o rifrazione; è uno **Shader di base** buono, salvo certe personalizzazioni, per legno, stucco, plastica, carta e tanti altri elementi... “*salvo certe personalizzazioni*”, perché chiaramente ci sono differenze nella modalità di dispersione della luce tra carta e legno o tra plastica e cemento, ma diciamo che *di base* si può partire da **Normal**, che è quindi uno **Shader per oggetti opachi, eventualmente texturizzati**.

La seconda famiglia è quella dei **Transparent**, per oggetti che presentano **parti trasparenti**, con queste parti definite dal **canale Alpha della Texture** utilizzata.

Certi formati di immagini, come il **PNG** o le **GIF**, mettono a disposizione anche un **canale Alpha** per definire il **grado di trasparenza dei vari pixel** dell'immagine; se questo canale è presente, potete associare un'immagine (ad esempio un logo) ad un oggetto, che sarà **trasparente nelle parti con Alpha semitrasparente nella Texture**. A breve, in questo stesso **tutorial**, farò un esempio pratico di questo particolare caso.

Il tipo **Self-Illuminated** è ottimo per gli oggetti che devono **emettere luce**; pensate, ad esempio, a insegne al neon, pannelli luminosi, strumenti fantascientifici di laboratorio e così via.

Il tipo **Reflective** è utile, infine, per gli **oggetti opachi che però possono riflettere**, parzialmente o completamente, l'ambiente.

Sulle **colonne** dell'immagine nella pagina precedente, detta *“builtin Unity Shaders Matrix”* (griglia degli Shaders nativi di Unity) troviamo le versioni **Semplice** e **Bumped** o **Parallax Bumped** per ottenere i **rilievi** su un oggetto mediante le **Textures**, senza andare a modellare quelle parti), più la sottodivisione **Diffuse** e **Specular/Glossy** di cui vi ho già parlato prima.

L'incrocio di riga per colonna, in base alle caratteristiche di **opacità, rilievi e tipo di riflessi (Diffuse / Specular)** vi dà lo **Shader** che vi serve, quindi se sapete che volete ottenere un certo tipo di materiale guardate questa immagine per capire **quale Shader scegliere**.

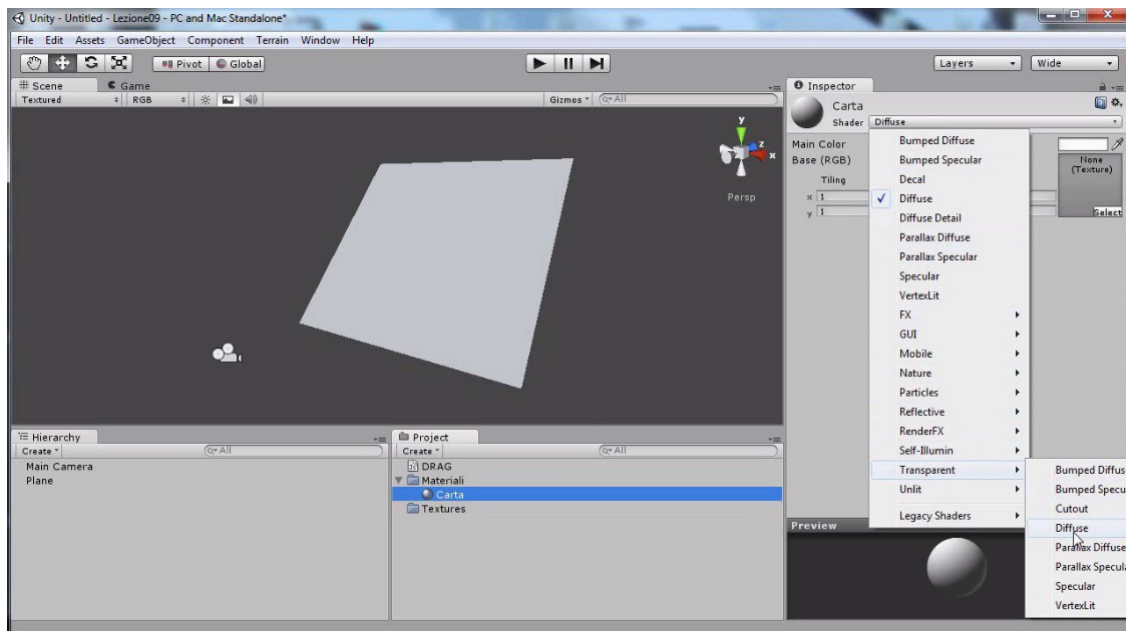
* * *

Torniamo alla scena di **Unity** e vediamo un esempio di **Alpha Texture** su un **Material Transparent**.

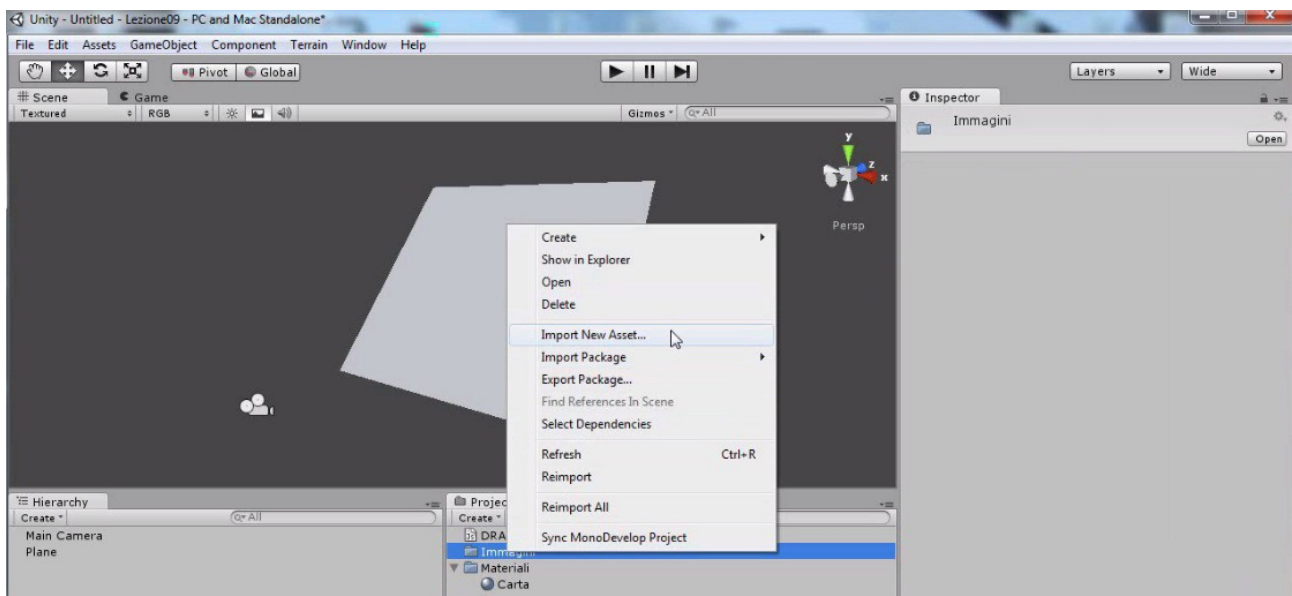
Nella scena abbiamo il **Plane** visto nella prima parte di questo **tutorial**.

Aggiungiamo, nel **Project**, un **Material** (dentro una **cartella “Materiali”**, per raggrupparli e fare ordine nel progetto) che chiameremo ad esempio **Carta**.

Vogliamo un **Material** che presenti **parti trasparenti** definite dal canale **Alpha** della **Texture** che caricheremo, comunque il **Material** deve avere un'illuminazione standard opaca, senza **effetti di Bumping** né **riflessi Glossy**, per cui, ricordando l'immagine vista poco fa, nell'**Inspector** cambiamo lo **shader** di questo **Material** da **Diffuse** a **Transparent – Diffuse**: un oggetto opaco diffuso con elementi di trasparenza, appunto.



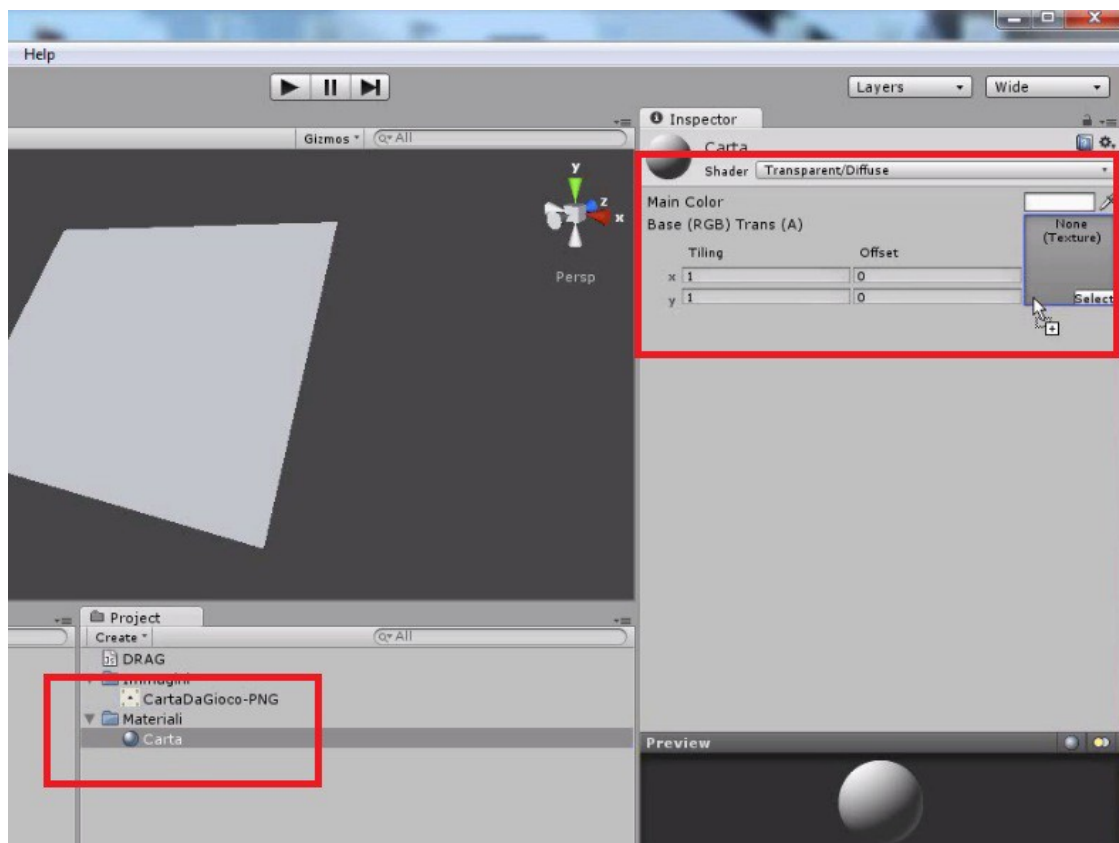
Per inserire **un'immagine Texture nel Project**, invece, creiamo una cartella chiamata ad esempio **Immagini** (anche qui, per questioni di ordine nel progetto) e, con un click destro, scegliamo **Import New Asset**, quindi scegliamo da disco l'immagine della carta.



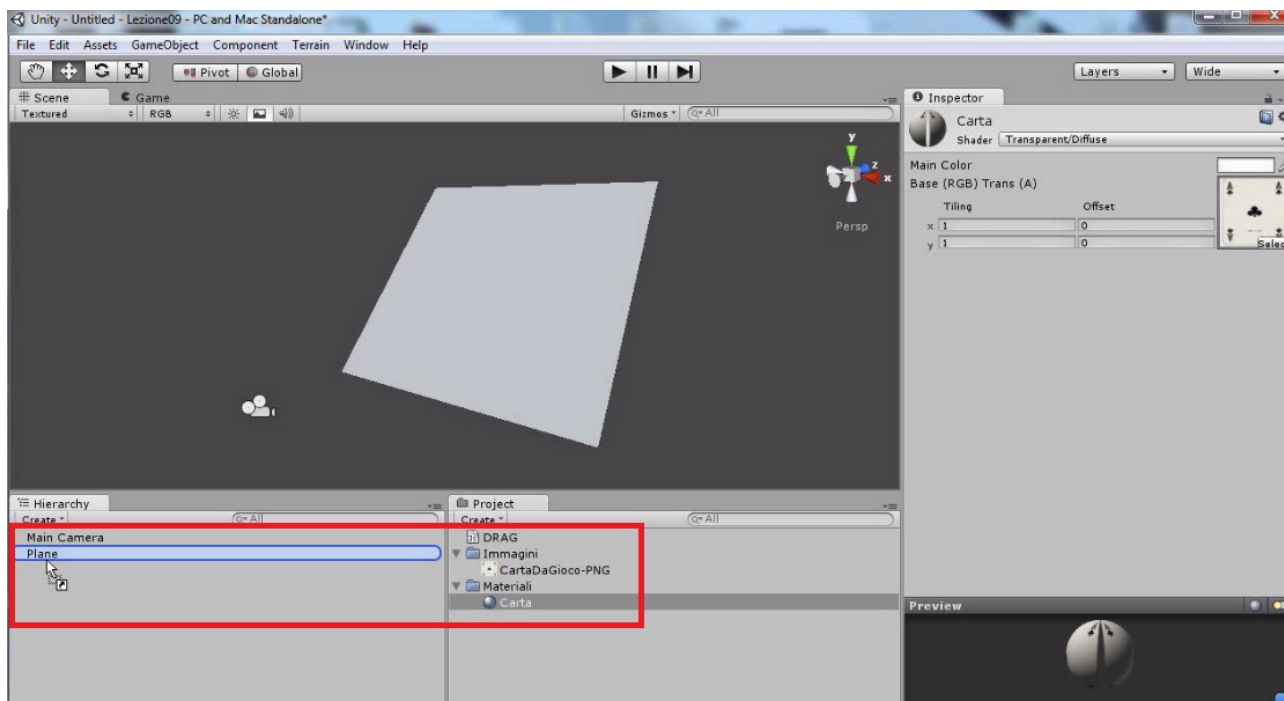
Il file che sto utilizzando è in formato **PNG** con canale **Alpha**, in particolare è **trasparente agli angoli**, dove il disegno della carta è arrotondato e non riempie il rettangolo dell'immagine.

A questo punto dobbiamo agganciare la **Texture** immagine al **Material** e, in seguito, il **Material** all'oggetto di gioco.

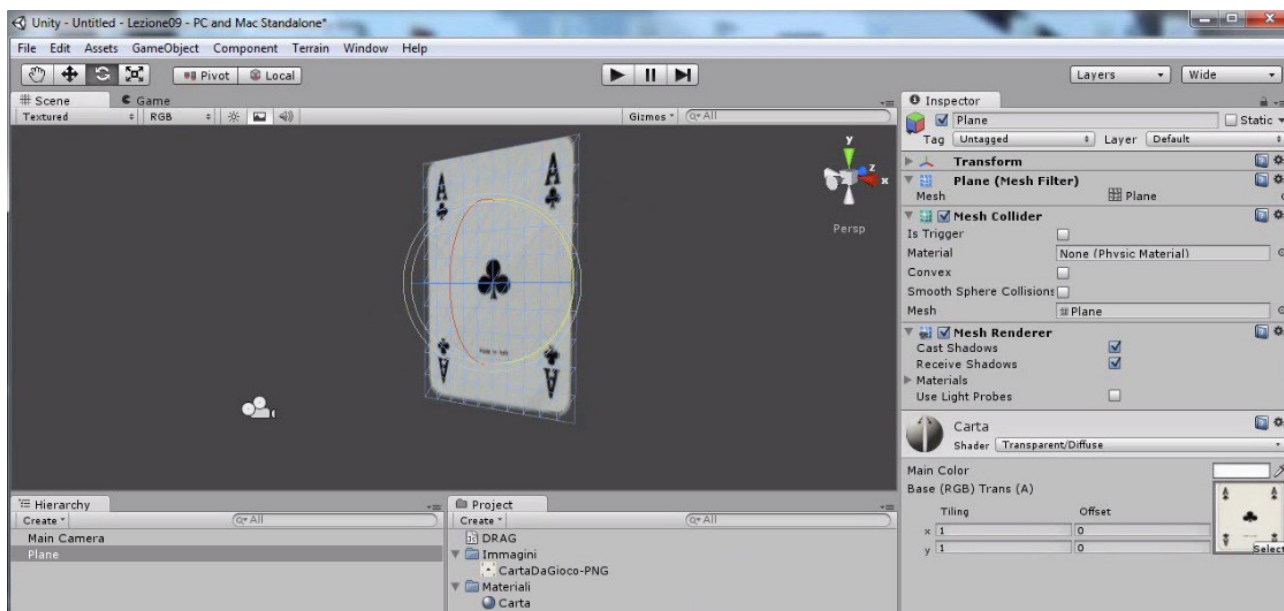
Selezioniamo il **Material**, quindi clicchiamo sull'immagine **Texture** e, tenendo premuto il mouse, trasciniamo tale elemento nello **slot Texture** del **Material**: abbiamo appena associato la **Texture** al **Material**.



Adesso, **trasciniamo il Material dal Project all'oggetto di gioco**, o nella **Gerarchia** o nella **scena 3D**, per associare il tutto al **Game Object** (immagine nella prossima pagina).



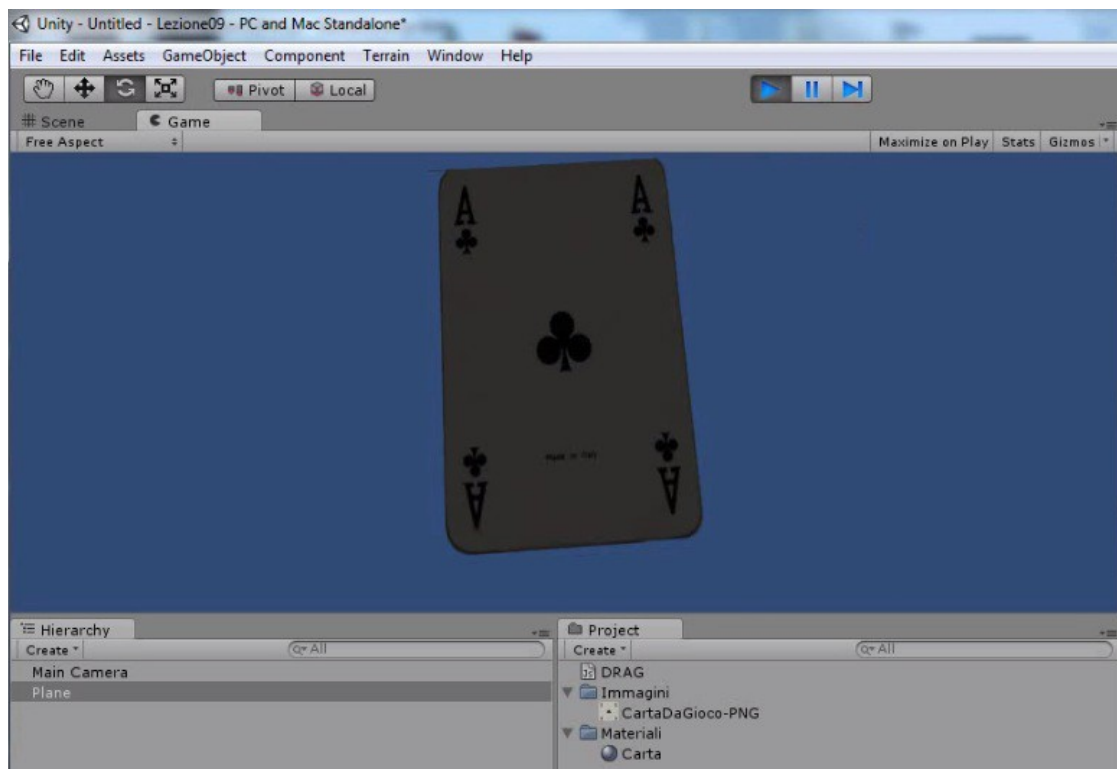
Nella scena è ora visibile il **Plane** texturizzato con le parti agli **angoli trasparenti**, seguendo il **canale Alpha** dell'immagine utilizzata come **Texture**.



Come vedremo **in un altro tutorial** (anche se potete già esplorare le funzioni di codice dalla

documentazione di Unity), potremo comunque **accedere al *Material*** utilizzato da un **Game Object** e, da qui, alle ***Textures* associate al *Material***, cambiandole al volo **durante l'esecuzione**.

NOTA conclusiva: è ovviamente possibile definire, mediante **script**, degli ***Shaders* personalizzati**, da utilizzare con i propri **Material** e, ovviamente, molti sviluppatori hanno creato **packages**, pacchetti di **Shaders** originali, programmandoli in proprio.



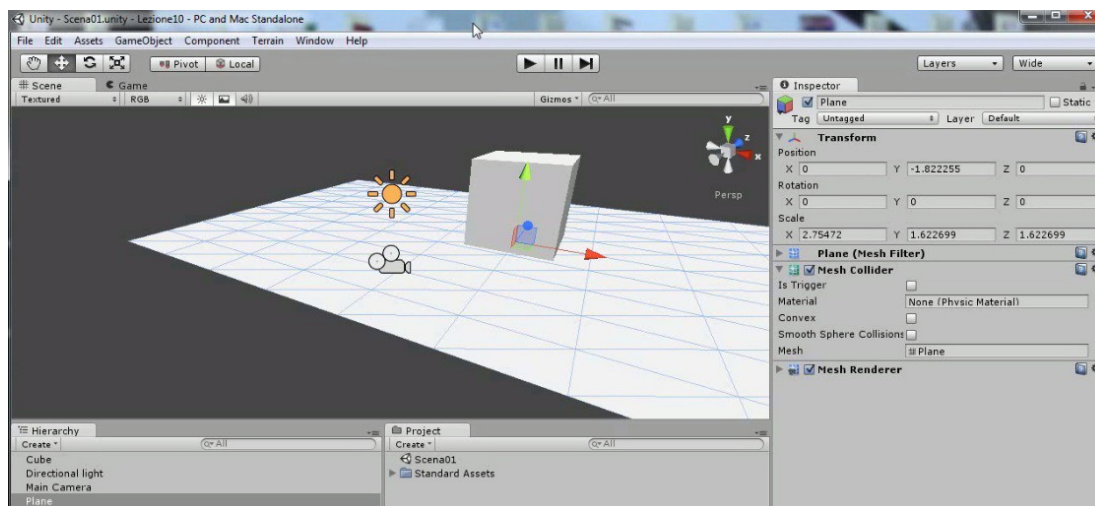
* * *

Le basi di Unity 3D 3.x e 4.x – 10: FPS Character Controller; rotazioni; cambiare scena / livello

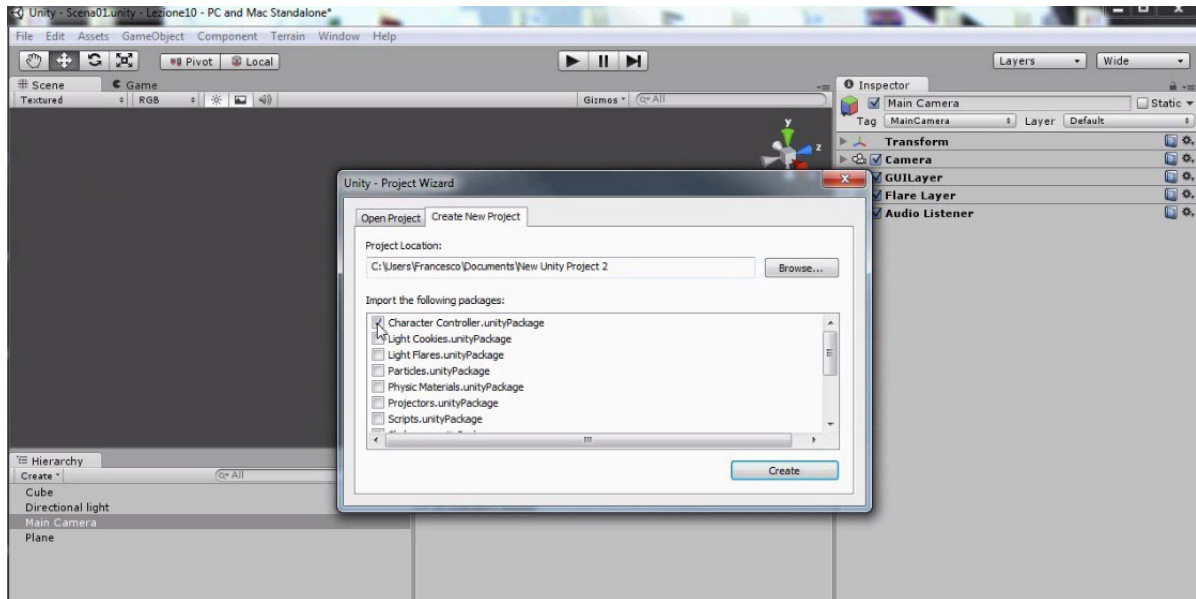
Eccoci al decimo e ultimo capitolo del corso introduttivo su **Unity 3D**; in questa sede tratteremo la **rotazione** degli oggetti, l'**FPS Controller** nativo e le **funzioni per cambiare Scene**, utili per passare **da un livello di gioco ad un altro**.

Partiamo dal secondo argomento perché, in realtà, è il più semplice: l'**FPS Controller** nativo di **Unity 3D**. Come suggerisce il nome, si tratta di un **Component** che incorpora la gestione dell'input dell'utente in riferimento allo “schema” tipico degli **FPS**.

Il vero nome è, in realtà, **Character Controller**, ossia sistema di controllo del “personaggio” (in generale, quindi, visto che potete utilizzarlo per pilotare qualsiasi cosa: da un calciatore a un cubo... come in questo esempio).

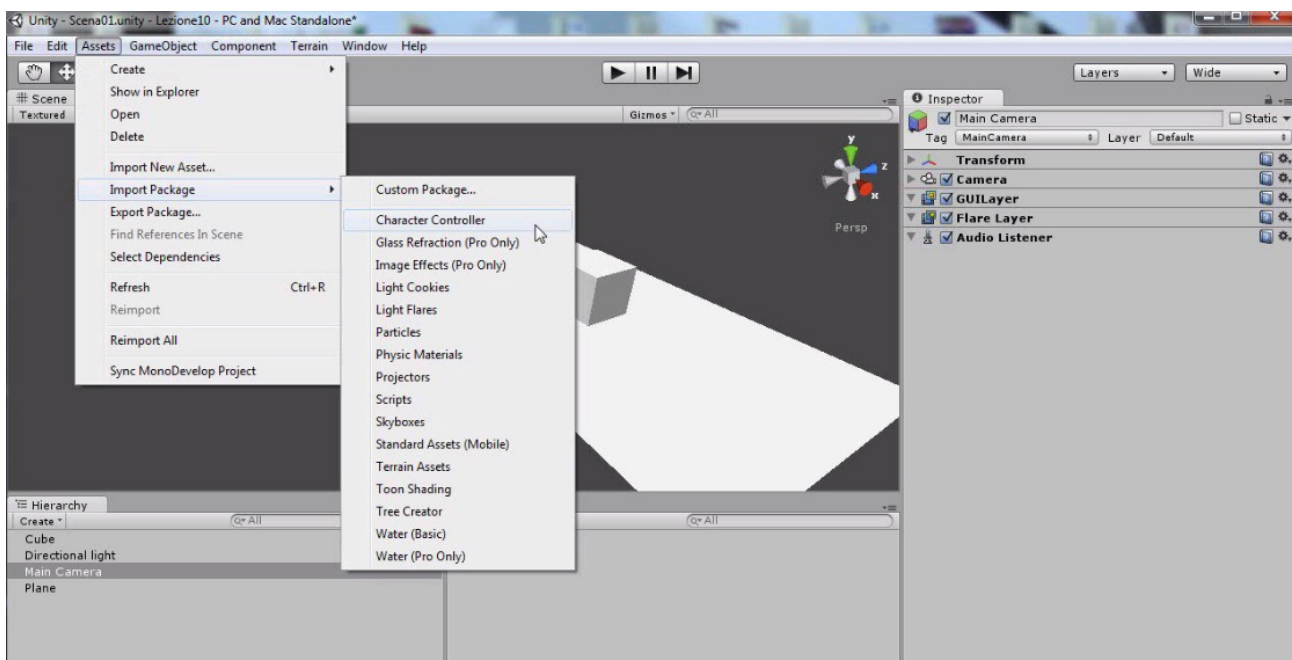


Il **Controller** va aggiunto, in genere, in fase di creazione di un nuovo progetto: facendo **File – New Project**, troveremo, nella schermata che apparirà, il campo ove inserire il nome del progetto e un **elenco di componenti aggiuntivi** che possiamo selezionare per inserirli da subito nel progetto.

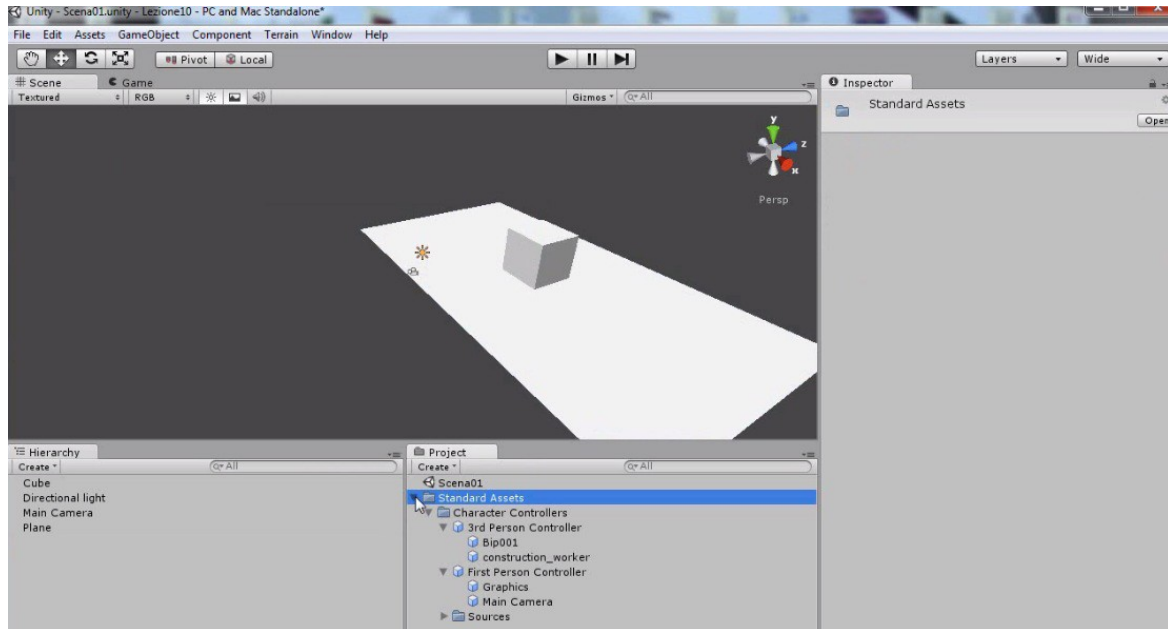


Il primo elemento della lista è proprio **Character Controller**, quindi se vogliamo inserire questo **package** sin da subito nel progetto selezioniamo la relativa casella e creiamo il progetto.

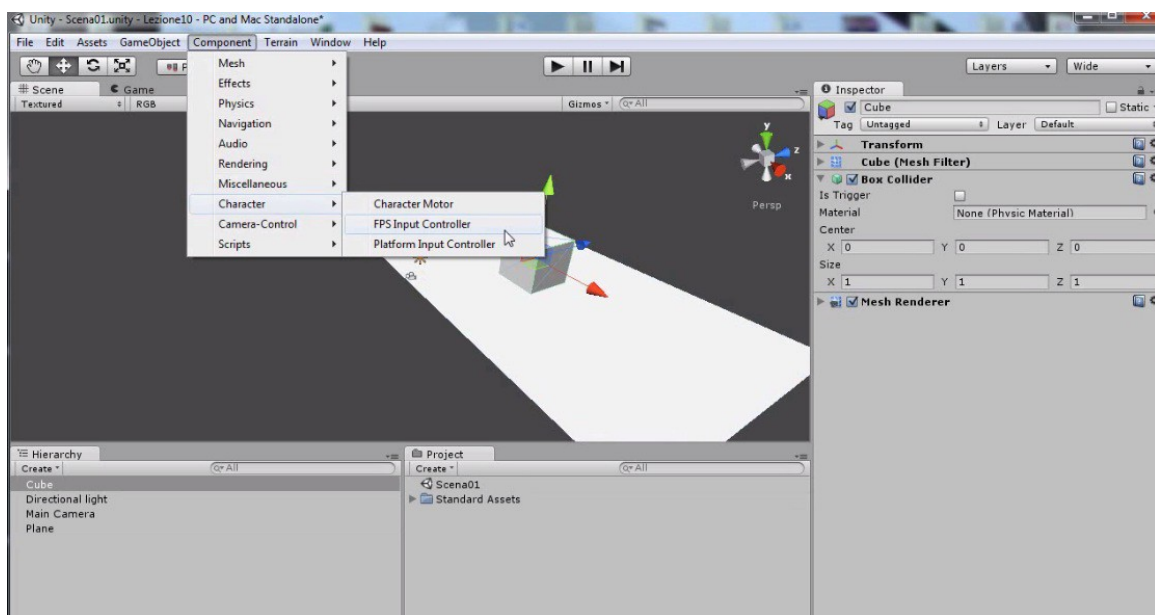
Inseriamo quindi questo **Controller** nella scena (è comunque aggiungere questo controller in un secondo momento, a progetto già creato: è sufficiente scegliere, dal **menù Assets**, la sezione **Import Package** e da qui, neanche a dirlo, **Character Controller**).



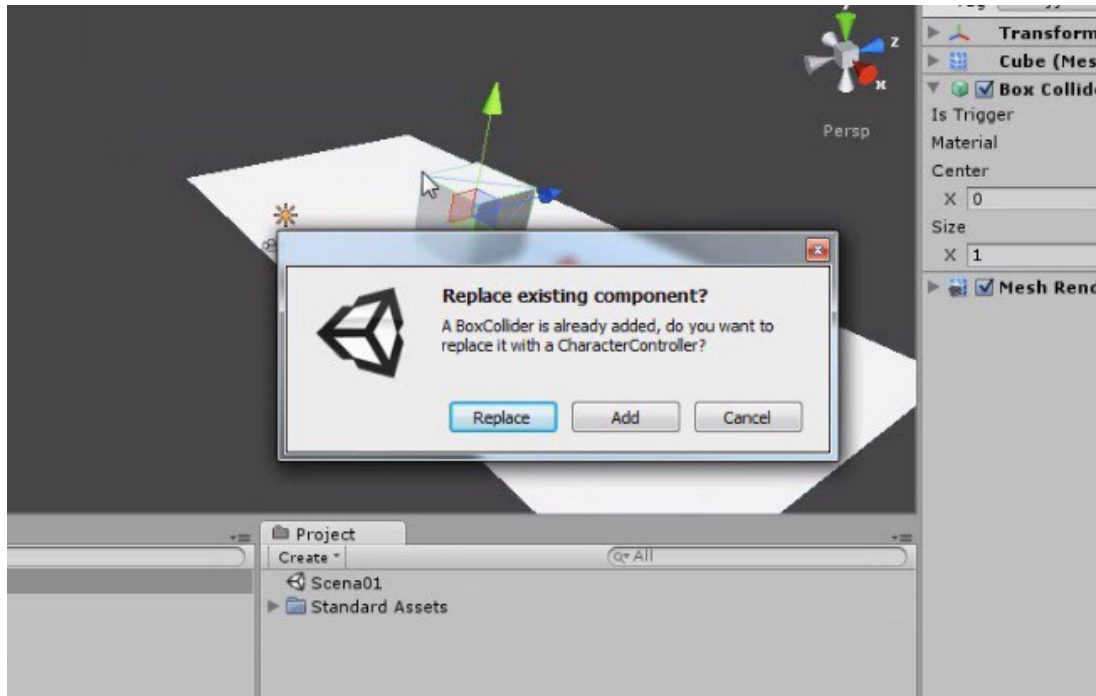
Effettuato l'Import del pacchetto (proprio come visto nel terzo capitolo di questo minicorso col **Terrain**), avremo una cartella di elementi nel **Project** e, soprattutto, nuove voci nel **menù Component**: la sezione **Character** e tre voci di menù in questa sottosezione.



Con l'oggetto da muovere selezionato (nel mio caso, il Cube posto sulla pedana) scegliamo, in particolare, **FPS Controller**: in pratica, come avviene con gli altri componenti (come i **Rigid Body** o gli **Script**) stiamo dotando l'oggetto di un **componente FPS Controller**.

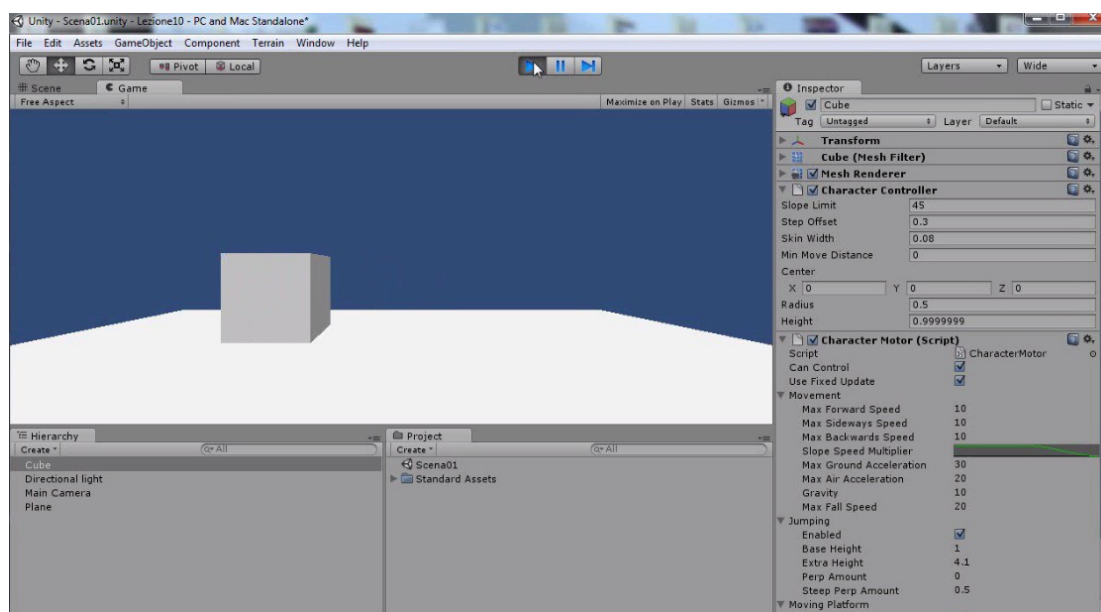


Unity ci chiederà se vogliamo sostituire il **Component Rigid Body** presente di default per il cubo con l'**FPS Controller**: i due elementi, quindi, sono esclusivi. Confermiamo la scelta.



In cosa si traduce tutto questo, all'atto pratico?

Mandiamo in esecuzione, cliccando su **Play**, quindi proviamo a premere i tasti freccia, WASD o la barra spazio: il cubo, dotato di **FPS Controller**, **si sposta e “salta”**, tra l'altro con un salto che avviene solo quando è effettivamente a contatto col terreno.



A questo punto sorge spontanea una domanda: ma se **Unity** ci dà già, di serie, questo **controller**, perché abbiamo visto come spostare e ruotare gli elementi via codice?

La risposta è semplice: per vedere come avviene l'elaborazione via codice e per capire come sono organizzati **i componenti e i moduli in Unity**, così sarà più facile, in futuro, capire **da cosa è composto un Package**, analizzarne i vari **Component** e **modificarli** a piacere.

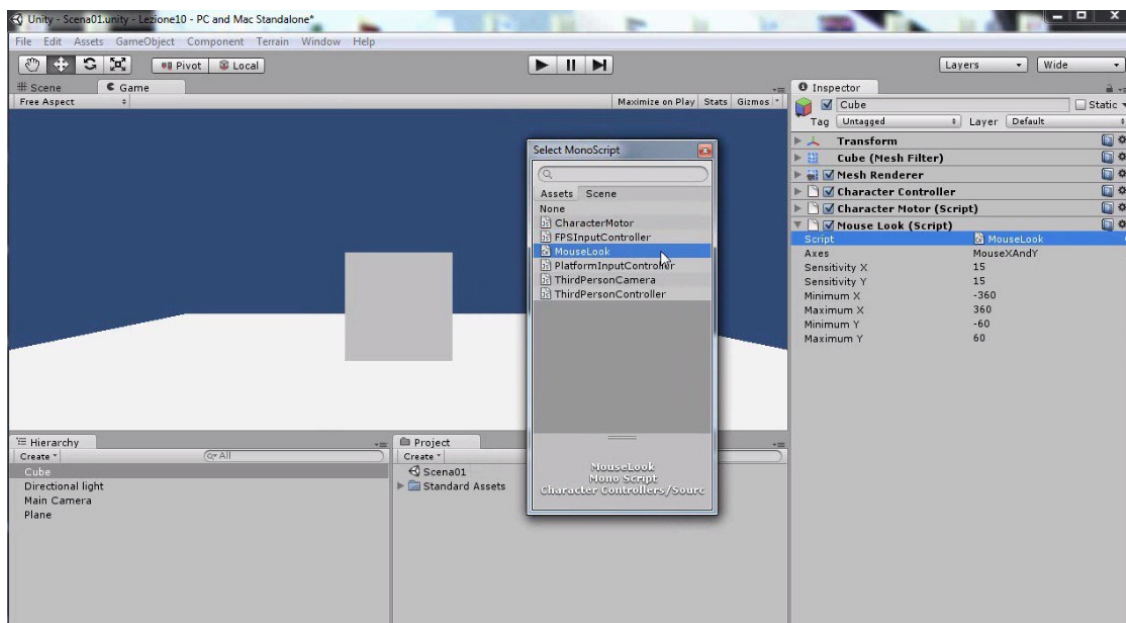
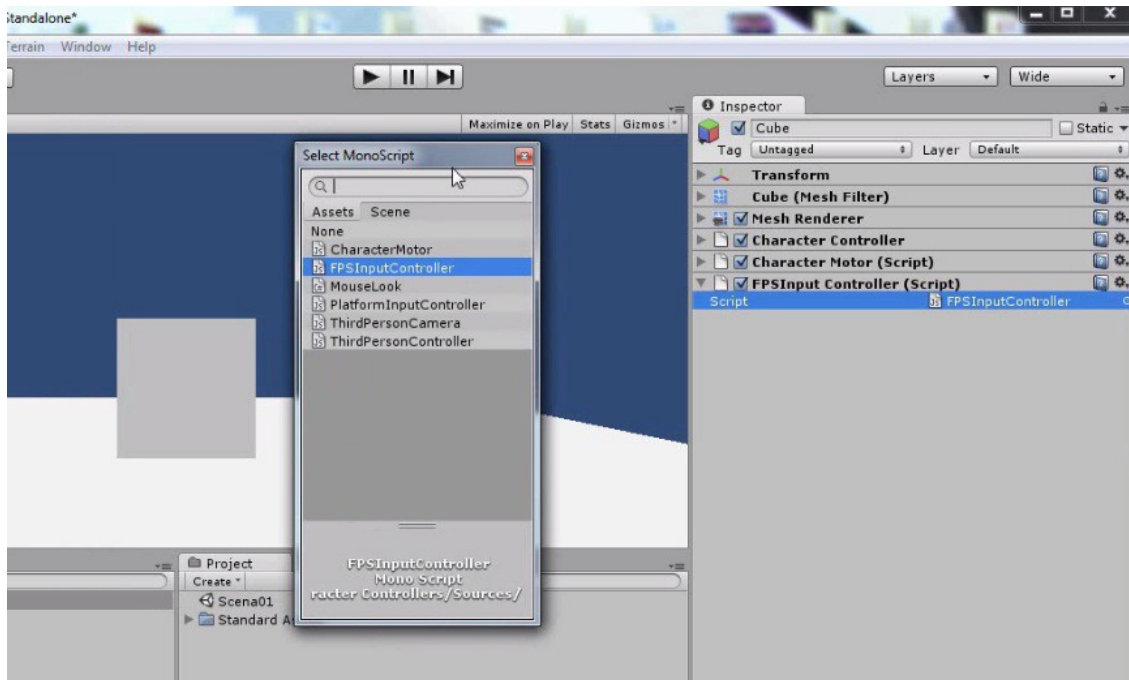
In un capitolo precedente abbiamo visto la procedura via codice, “sporvandoci le mani”; ora, invece, stiamo passando ad un **Controller** nativo perché... c'è già, quindi se il vostro scopo è quello di **spostare un personaggio con i tasti freccia** o – come vedremo tra pochissimo – **ruotarlo mediante il mouse**, usate direttamente l'**FPS Controller** ed eventualmente modificalo (ed è qui che entrano in gioco le competenze di scripting trattate nei capitoli precedenti: servono anche per modificare script altrui!)... in un certo senso, è inutile reinventare la ruota, ma è anche importante capire il funzionamento, gli elementi e le caratteristiche della ruota!

Parliamo quindi degli **elementi del Component FPS Controller**.

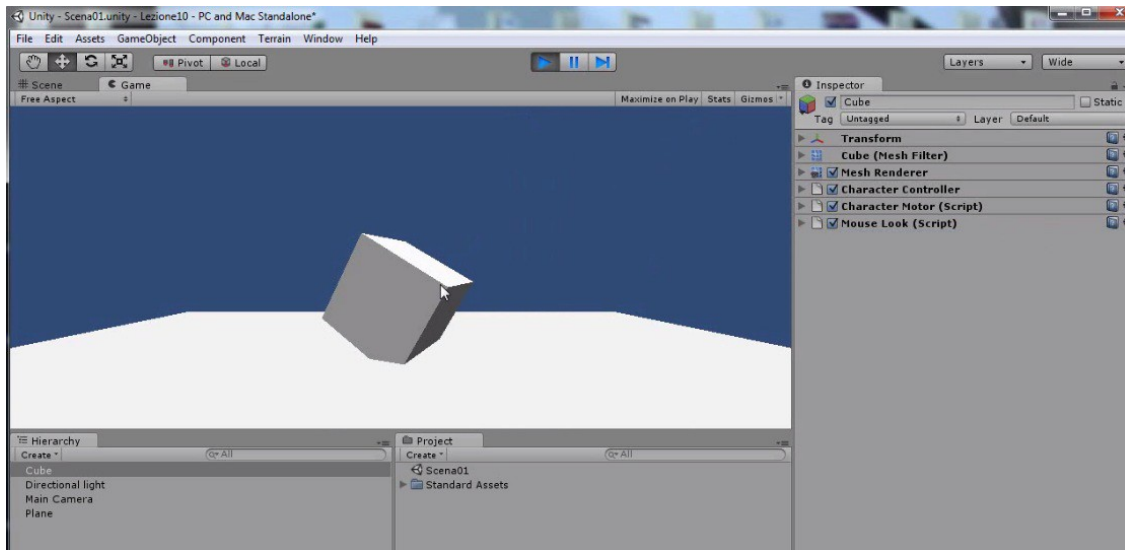
Selezionando il Cubo, nell'**Inspector** notiamo la presenza di alcuni nuovi **Components**, come **Character Controller** e **Character Motor Script**, ciascuno con le relative voci, la maggior parte delle quali dai significati intuitivi, per cui non mi dilungherò a descriverle ma vi invito a sperimentare direttamente; piuttosto, vorrei concentrarmi sul discorso del **Mouse Look**, ovvero come fare per **ruotare, cambiare l'orientamento dell'oggetto utilizzando il mouse**.

In questo momento, infatti, il cubo si muove seguendo lo schema frecce o WASD e la barra spazio, ma **non ruota mediante il mouse**, caratteristica tipica degli **FPS**.

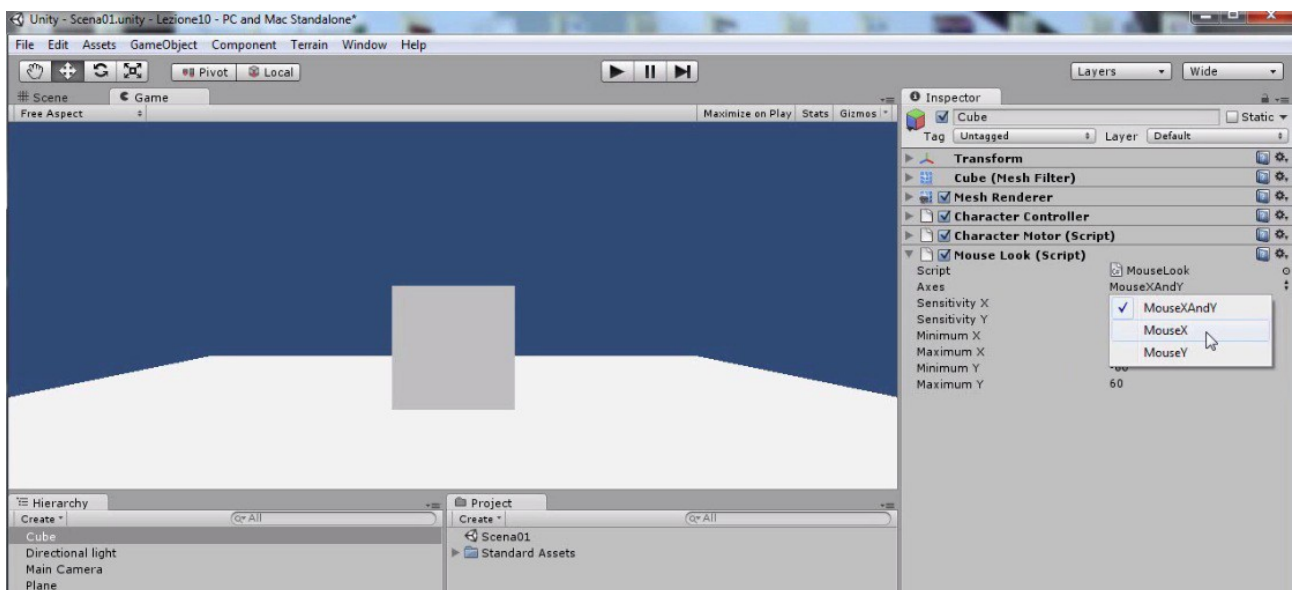
Guardando il **componente script FPS Input Controller**, nell'**Inspector**, notiamo che c'è un **cerchietto** accanto al nome dello script; se clicchiamo, appare una finestra di pop-up dalla quale **scegliere un altro script**, tra un insieme di script del package **Character Controller**, al posto di **FPS**... “al posto” perché la selezione di uno script diverso **sostituirà FPS**, né è possibile fare una selezione multipla.



Selezioniamo comunque **Mouse Look** e avviamo il gioco, per fare una prova (immagine nella prossima pagina): in effetti ora **con il mouse possiamo ruotare l'elemento** intorno ai due assi – direzioni del mouse, tuttavia abbiamo **perso il controllo con la tastiera**, per gli spostamenti... torniamo alla modalità **Edit**.



Per prima cosa, una nota sull'orientamento: come visto, l'oggetto si orienta sulle due direzioni del mouse, ma a volte è preferibile una **rotazione intorno al solo asse verticale**, non quello orizzontale; per cambiare questo aspetto, espandiamo lo **script *MouseLook*** e cambiamo, sempre dall'**Inspector**, la voce “**Axes**” da “**MouseXandY**”, di default, a “**MouseX**”, quindi riproviamo.

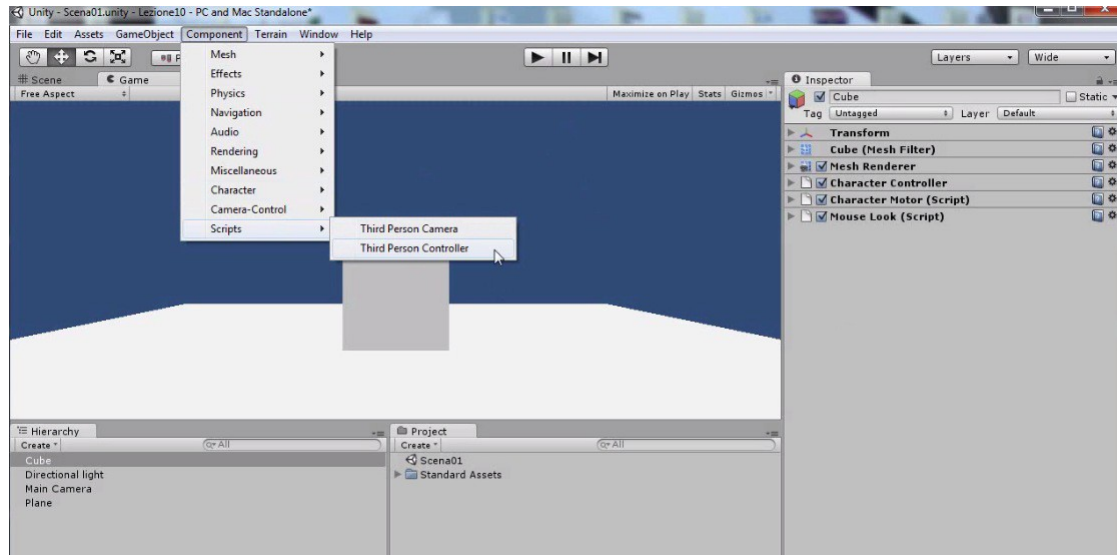


Con le altre voci potete cambiare altri aspetti, ma sono abbastanza intuitivi, quindi non perdiamoci troppo tempo e torniamo invece al discorso degli script “esclusivi”, per recuperare la componente di movimento da tastiera.

Riflettiamo: **cos'è lo script FPS Controller?**

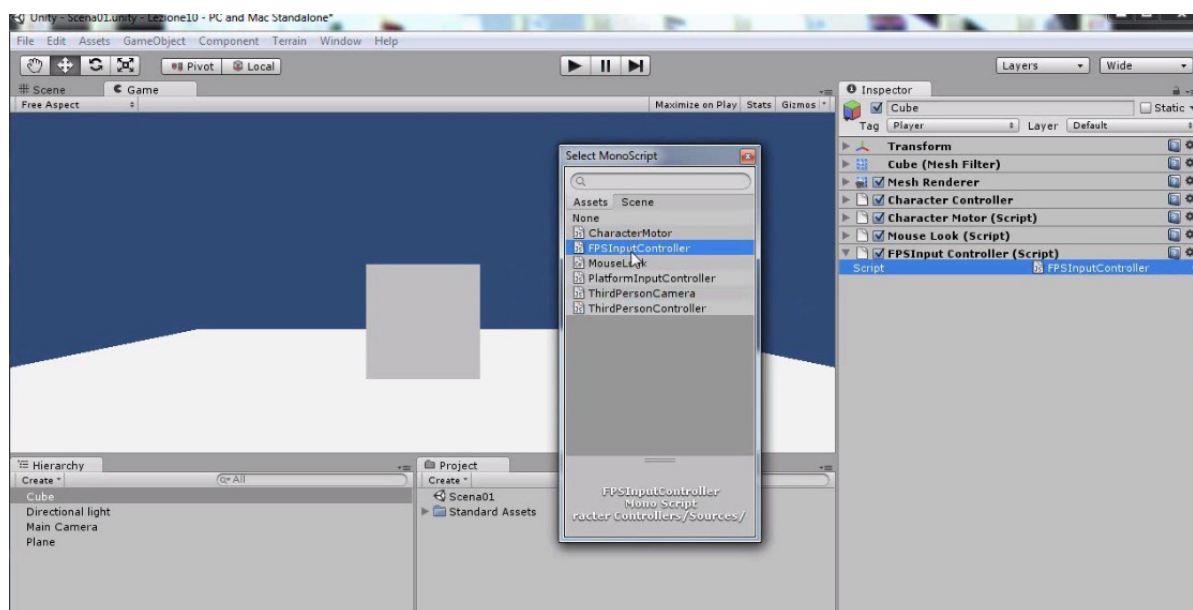
È uno script, come **MouseLook**, quindi è un **Component**...

Dal **menù Component**, quindi, scegliamo la sezione **Script** e, qui, **Third Person Controller**: un set di script importato mediante il **Package Character Controller**.

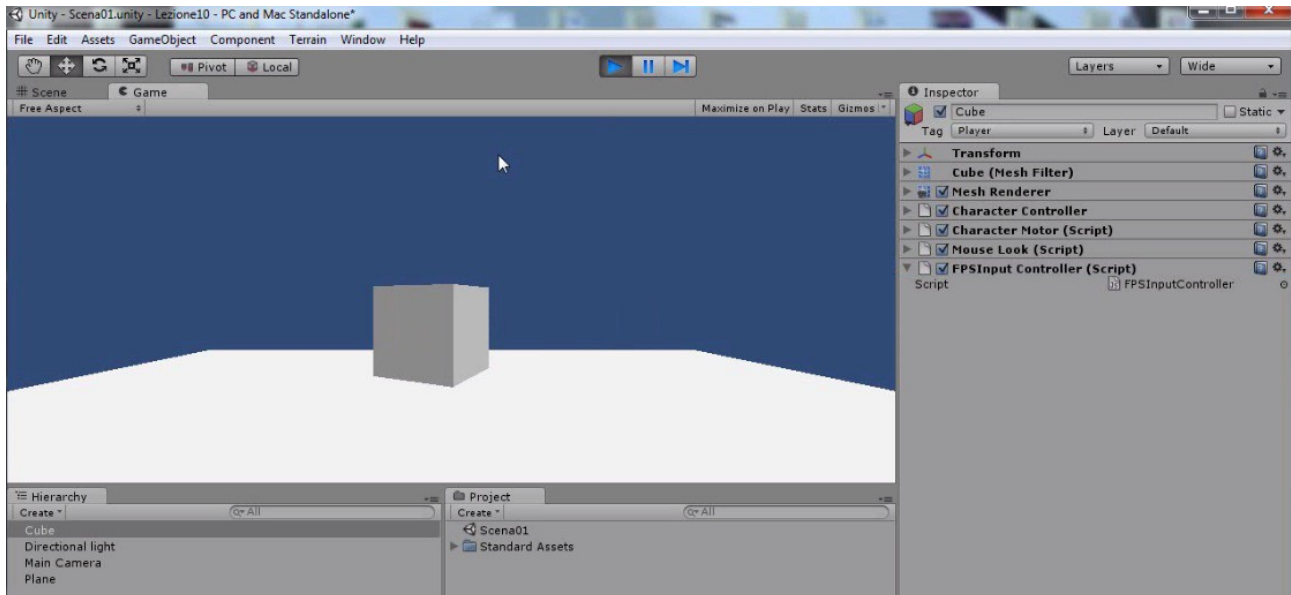


Nell'**Inspector**, per il cubo, apparirà un nuovo **Component** di tipo **Script**: **Third Person Controller**. Da notare che **MouseLook** è ancora lì, quindi i due **Component** non sono esclusivi, ma possono coesistere.

Questo script va già bene per il **movimento da tastiera** (provare per credere), ma se volete potete cliccare sul cerchietto e scegliere **FPS Controller** dal menù che apparirà per utilizzare un **Controller FPS vero e proprio**.



Una nota a margine (ma non troppo): nella scena c'è un **Plane** che fa da pavimento perché l'**FPS Controller** sostituisce, come detto, il **Rigid Body** di default, implementando quindi anche una **simulazione fisica**, con **gravità** ed altri elementi discussi nei tutorial precedenti, quindi senza Plane-pavimento il cubo con **FPS Controller** cadrebbe nel vuoto.



In questa prima parte del tutorial abbiamo visto (anche se in maniera molto superficiale, a livello di introduzione) l'**FPS Controller** e il **MouseLook**, per **ruotare gli elementi mediante il mouse**; vediamo ora **cambiare livello di gioco in Unity 3D**.

* * *

Come detto nel settimo capitolo (del corso base), in **Unity** gli ambienti o **livelli di gioco** sono le **“Scenes”**, le scene.

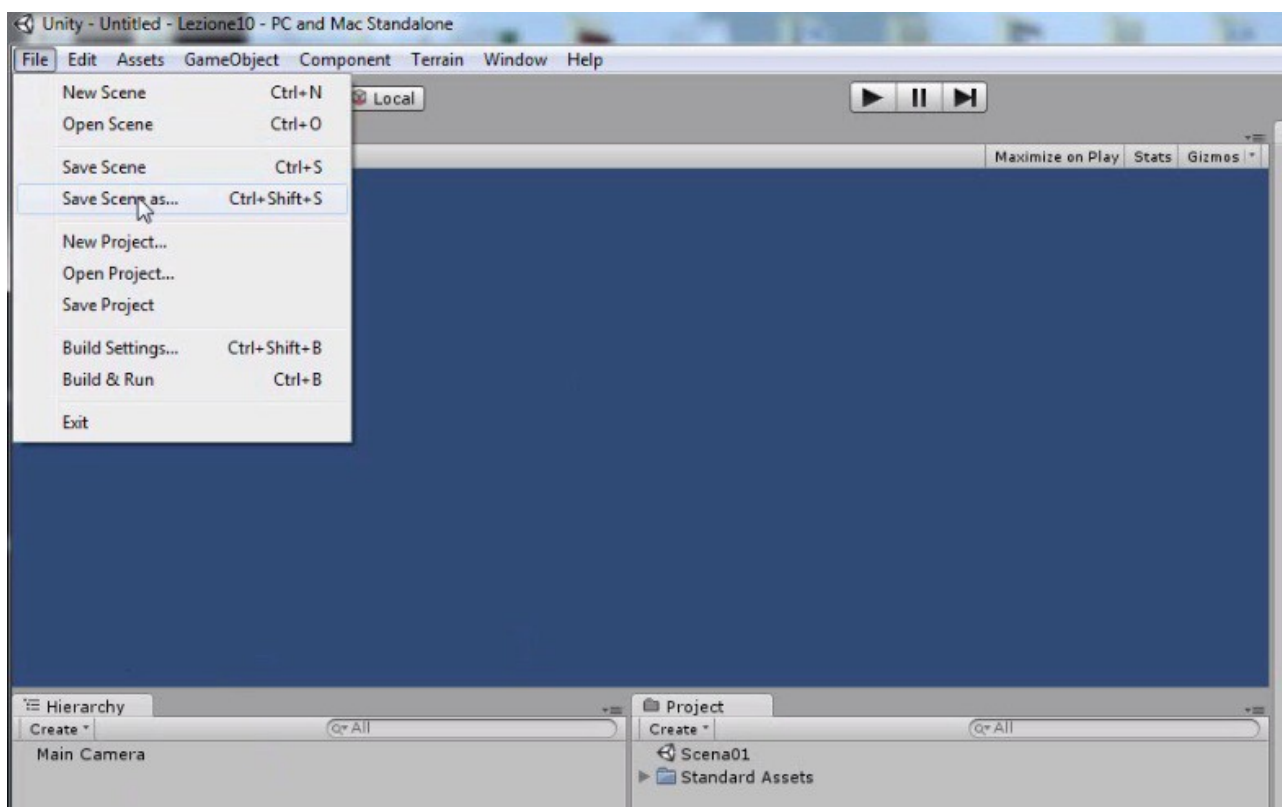
Abbiamo detto inoltre che, prima di fare il **Build** (la compilazione di un progetto), bisogna assicurarsi di avere **tutte le Scenes inserite nel pannello Build Settings**, con particolare riguardo alla prima scena della lista, che sarà l'ambiente o schermata di apertura del gioco.

Quanto al passaggio vero e proprio ad un'altra **Scene** (ovvero ad un altro ambiente o livello), questo avviene mediante una semplice istruzione di codice, mentre il **passaggio dei dati** può essere effettuato, ad esempio, mediante i **PlayerPrefs**, anche questi trattati in precedenza... come potete vedere, tanti piccoli elementi, trattati “a parte” in altri capitoli, tornano utili adesso.

Procediamo con ordine: attualmente, nel **Project** abbiamo una sola **Scene**, per cui per prima cosa **aggiungiamo un'altra Scene con File – New Scene** (Unity vi chiederà di salvare i cambiamenti apportati alla prima **Scene**; clicchiamo sul sì e andiamo avanti).

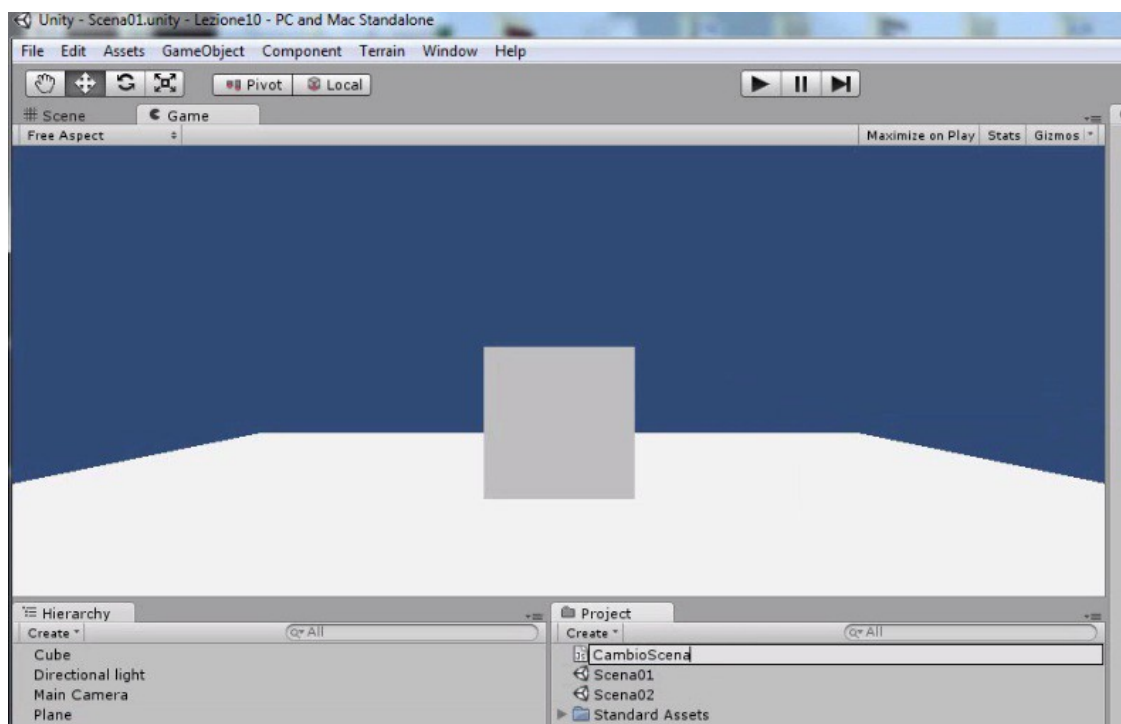
La nuova **Scene** è completamente **vuota**, ad eccezione della **Main Camera**, la telecamera principale.

NOTA: salviamo subito questa seconda scena con **File – Save Scene**, chiamandola ad esempio **Scene02**. In seguito, potremo passare, in fase di **Editing**, da una scena all'altra cliccando sul relativo elemento nel **Project**. Per progetti ricchi di scene, conviene creare una cartella ad hoc nel **Project**, per raggrupparle.



Torniamo quindi alla **Scena01**, quella originale, cliccando su tale elemento nel **Project**.

Selezioniamo la **telecamera** e aggiungiamo un nuovo **script JavaScript**, chiamandolo ad esempio **CambioScena**. Lo script che mostrerò è molto semplice, in quanto cambia scena in base alla **pressione dei tasti 1 (Scena01) e 2 (Scena02)**, ma ovviamente potete farlo **reagire ad altri eventi**; ad esempio, potrebbe cambiare scena se rileva che un personaggio si trova in un determinato ambiente (controllando le **coordinate Transform XYZ**) e preme un particolare tasto, in modo da simulare ad esempio una **cabina di teletrasporto** o la **fine di un livello**, giusto per fare due esempi.



Nello **script**, all'interno della funzione (predefinita) **Update** scriviamo le seguenti linee di codice:

```
if (Input.GetKey ("1"))           Application.LoadLevel ("Scena01") ;  
if (Input.GetKey ("2"))           Application.LoadLevel ("Scena02") ;
```

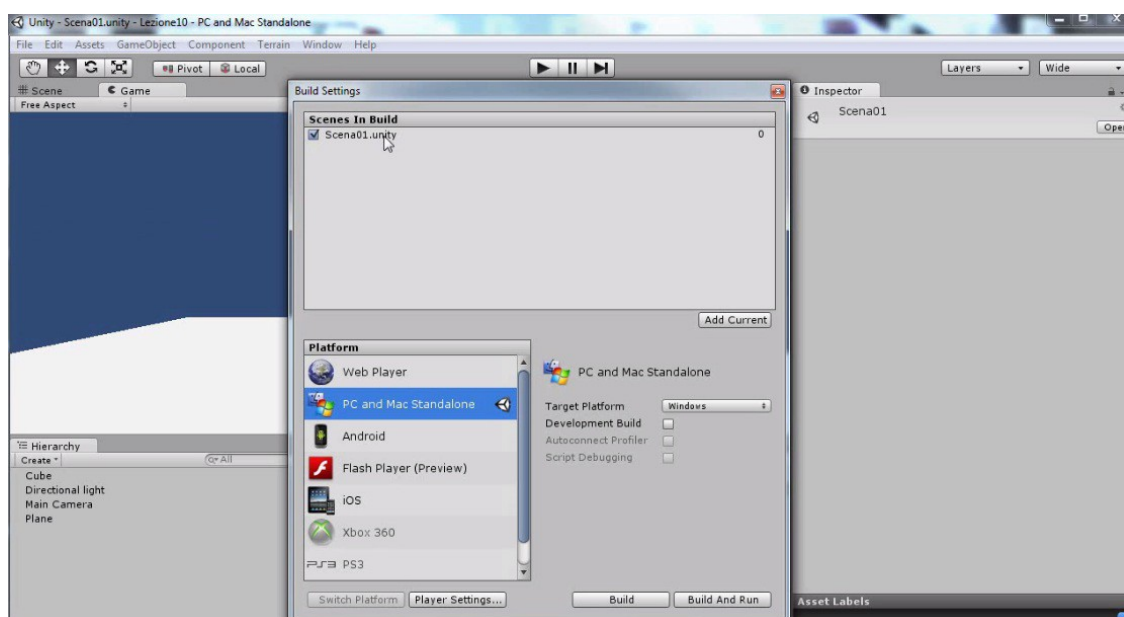
Salviamo lo script, torniamo al gioco e associamo questo script alla **Main Camera**... attenzione, però! Quella **Main Camera** *non* è lo stesso oggetto per le due scene!

Dobbiamo quindi: associare lo script alla **Main Camera** nella **Scena01**, salvare, aprire la **Scena02**, associare lo script alla **Main Camera** di tale scena e salvare anche questa scena.

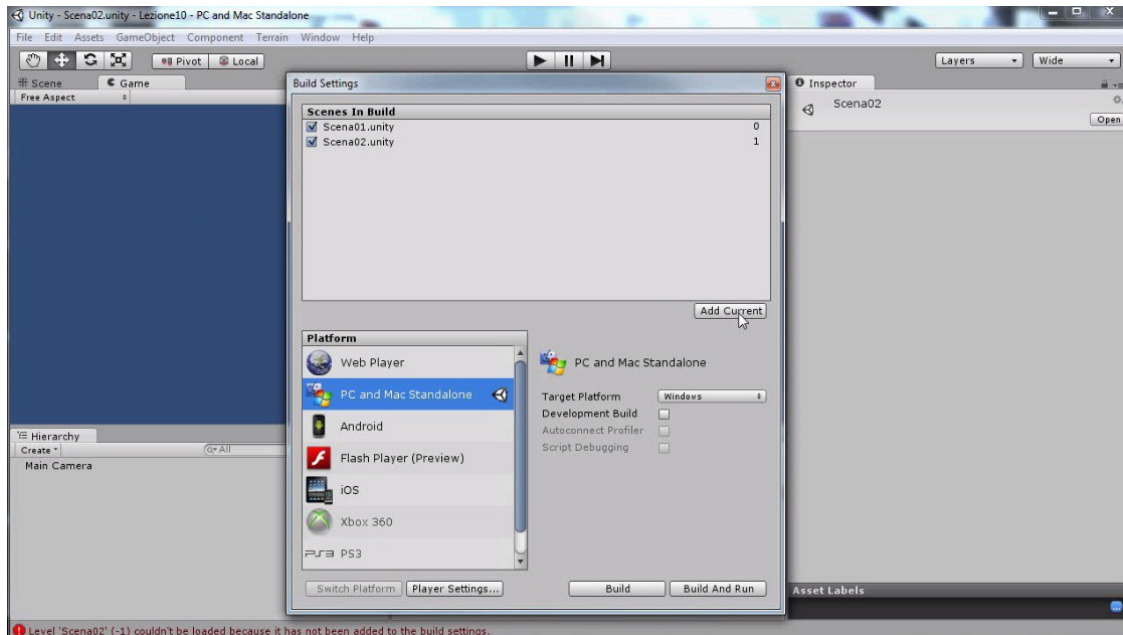
Assicuratevi di avere **uno script di cambio scena in tutte le scene del vostro progetto** o, per l'uscita, di qualche controllo e istruzione di **chiusura dell'applicazione**, per non rimanere “bloccati” da qualche parte durante l'esecuzione.

Mandiamo in esecuzione e premiamo **1** o **2** per passare da una scena all'altra: come potete vedere, l'unica istruzione di codice che ci serve per cambiare scena è **“Application.LoadLevel”** con il **nome della scena da caricare** come argomento.

In realtà, non abbiamo finito: mandando in esecuzione e premendo **2**, infatti, **Unity** ci dirà che non può farlo perché **Scena2 non è nell'elenco delle scene del gioco** per la release finale (nell'immagine sotto: notare l'assenza della seconda scena nell'elenco delle Scene da includere nella Build, la compilazione del progetto).



Procediamo in questo modo: in **Scena01**, scegliamo **File – Build Settings** e, nella finestra che apparirà, clicchiamo su **Add Current** per aggiungere **Scena01** all'elenco di scene del progetto; a questo punto, apriamo **Scena02** con un doppio click su tale voce nel **Project**, quindi di nuovo in **Build Settings** clicchiamo su **Add Current** per aggiungerla al progetto finale, ottenendo la situazione visibile nell'immagine nella pagina seguente (**NOTA**: come detto nel settimo capitolo della guida, la prima scena dell'elenco sarà quella di avvio del gioco).



Possiamo quindi tornare alla schermata principale del programma e mandarlo in esecuzione con **Play**, per poi provare il **cambio di livello** premendo **1** e **2**.

Anche la **Main Camera del secondo livello**, quindi, deve essere dotata di **script CambioScena**.

Un potenziale problema è quello del **passaggio dei dati da una Scena all'altra**; ad esempio, il giocatore potrebbe impostare il nome e i controlli nella schermata delle opzioni, quindi passare a un livello di gioco vero e proprio e, da lì, alla schermata dei punteggi.

Un metodo per passare tutti questi dati consiste nel **tramutarli in coppie nome-valore** e utilizzare i **PlayerPrefs**, come visto nel settimo tutorial del corso base, infatti i **PlayerPrefs** sono **globali**, del progetto, indipendentemente dalla **Scena** in uso, quindi potete **salvare un dato con un SET** nella **Scena01** e recuperarlo, mediante il nome della variabile, con **un GET nella Scena02**, e così via.

Anche questo aspetto, quindi, è molto facile da gestire (a patto ovviamente di accordarsi all'inizio sui **nomi dei PlayerPrefs**, soprattutto se lavorate in **team**).

* * *

Il Car Tutorial – Parte 1

Questo è il primo di tre tutorial basati sul “**Car Tutorial**” fornito gratuitamente sul sito ufficiale di Unity 3D Game Engine, che ho tradotto in italiano e portato a video per voi.

Come anticipato nel 10mo e ultimo tutorial sulle basi di **Unity 3D**, si tratta di un tutorial su un pacchetto zip scaricabile gratuitamente dal sito ufficiale che mette a disposizione uno scenario di corse automobilistiche: un modello 3D già realizzato, completo di auto e, soprattutto, molti script per gestire il comportamento del veicolo e di altri oggetti di scena.

Vedremo quindi quali sono gli elementi che compongono lo scenario e come personalizzarli per ottenere vari effetti.

Una nota a parte, prima di iniziare: nei tutorial – inclusi questi – ci sono dei concetti che riguardano non solo, ovviamente, il caso specifico in esame, ma che possono essere utilizzati in vari progetti (ad esempio, giusto per dirne due che trovate qui, l'utilizzo di un **Projector** per le ombre o l'offset tra mesh per evitare gli sfarfallii delle textures), quindi anche se l'argomento non vi interessa moltissimo vi consiglio, in generale, di seguire tutti i tutorials che riuscite a trovare.

In questo tutorial, in particolare, vedremo come creare un gioco di corse automobilistiche, **non da zero** ma trattando determinati argomenti come la creazione dell'auto partendo da modelli 3D, script e componenti.

Realizzeremo un progetto di guida completo, che potrete testare sul vostro pc (ma non su dispositivi mobile), dove potrete giocare in uno scenario e vedere come tutto è messo insieme.

Il pacchetto scaricabile dall'**Asset Store di Unity 3D**, quindi, è una scena completa, ad eccezione dell'auto, che personalizzeremo nel corso del Tutorial.

Iniziamo quindi parlando di cosa faremo. Nel corso di **tre tutorial**, vedremo:

- **come assemblare l'auto**, creando un Prefab partendo da modelli 3D, script e Components; è l'argomento di questo tutorial;
- **come modificare l'auto**, personalizzandone aspetti e parametri, per farla “comportare meglio”, in maniera fisicamente più accurata o, di contro, secondo i nostri gusti e desideri;
- **come modificare l'auto, di nuovo**, ma questa volta agendo direttamente sugli script di programmazione che gestiscono movimento, azioni, reazioni e in generale il comportamento dell'auto.

PREREQUISITI

Prima di iniziare, parliamo di cosa è necessario sapere, ossia i “prerequisiti”.

Questo non è un tutorial di base su Unity; in particolare, se non sapete già cos'è un **Game Object**, com'è organizzato **Unity**, cosa sono i **Components**, eccetera, vi consiglio di esaminare i miei 10 tutorials introduttivi sulle basi di **Unity 3D**, nei capitoli precedenti.

COSA NON FAREMO

Ecco cosa **non** faremo: non tratteremo gli script del pacchetto che non riguardano l'auto, non parleremo degli argomenti davvero di base (dati per scontati), non spiegheremo il flusso di lavoro e le componenti in dettaglio; per questi aspetti, si rimanda a tutorial di base (già citati) e alla documentazione online, che si consiglia di consultare per saperne di più su qualche **Component** o funzione nativa di **Unity 3D**.

L'approccio iniziale che seguiremo è quello di mostrare le istruzioni da fare, che dovrete eseguire per inserire l'auto e modificarne le variabili (e, quindi, il comportamento); all'inizio, quindi, dovrete seguire i passi, quasi “per fede”, poi quando avrete il quadro più ampio i vari elementi vi appariranno chiari. Ovviamente, siete liberi di “esplorare”, di cercare nella documentazione e sperimentare alternative.

Nel terzo e ultimo tutorial, il più vasto, esamineremo più in dettaglio il codice che “muove” (letteralmente) l'automobile. Non sarà una guida linea-per-linea, ma in generale copriremo tutti gli aspetti. Un modo per imparare a programmare o comunque migliorare le proprie capacità consiste nell'esaminare molto codice (oltre che, ovviamente, programmare per conto proprio, creando prodotti originali). In ogni caso, imparerete molto studiando il funzionamento di questo progetto.

Iniziamo quindi ad “assemblare” l'auto.

PARTE 1: ASSEMBLARE L'AUTO

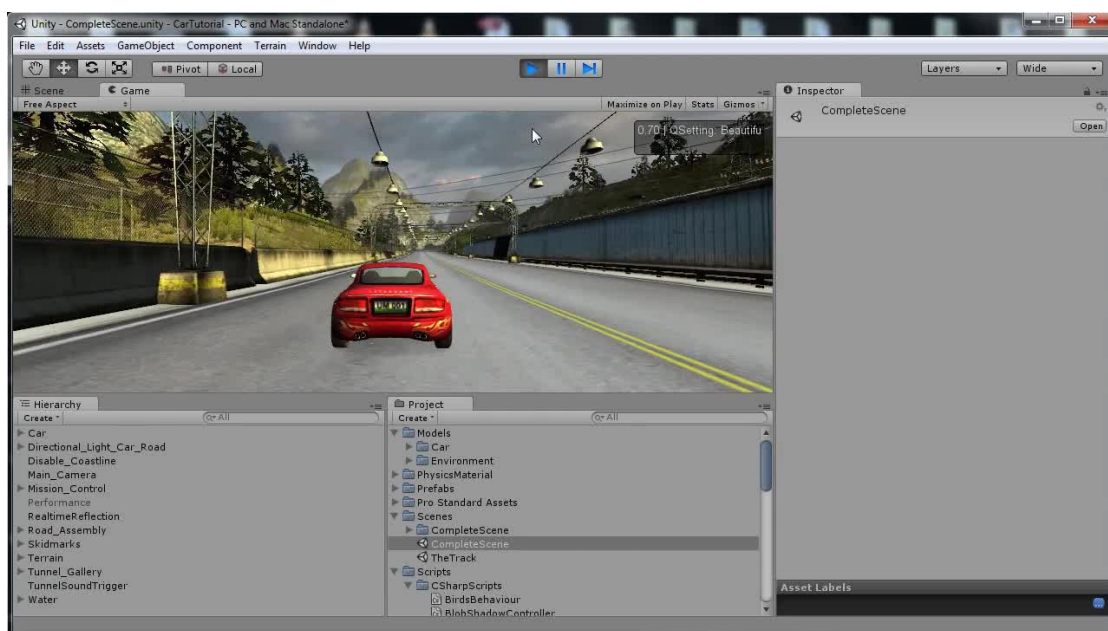
La prima cosa da fare è scaricare il pacchetto zip originale dall'**Asset Store di Unity 3D**; per mantenerne una copia “originale” sempre disponibile, l'ho caricato anche su un mio spazio web, quindi potete scaricarlo anche da www.redbaron85.com/unitycartutorial.zip.

Scompattate l'archivio e caricate il progetto con **Open Project** in **Unity 3D**.

Iniziamo aprendo la scena chiamata **Complete Scene**, scena completa.

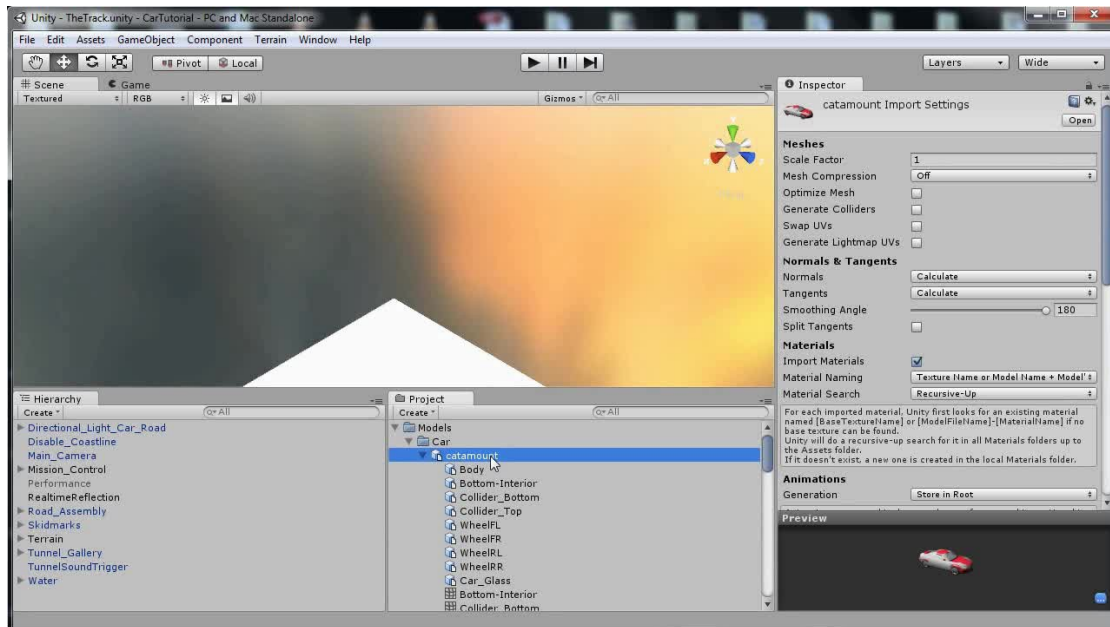
Qui è già tutto pronto, quindi possiamo provare subito a giocare all'interno dell'editor cliccando su **PLAY**, più che altro per vedere su cosa stiamo lavorando e cosa otterremo.

Terminata la prova, premiamo **Stop** e apriamo la **scena** chiamata **“The Track”**, il circuito.

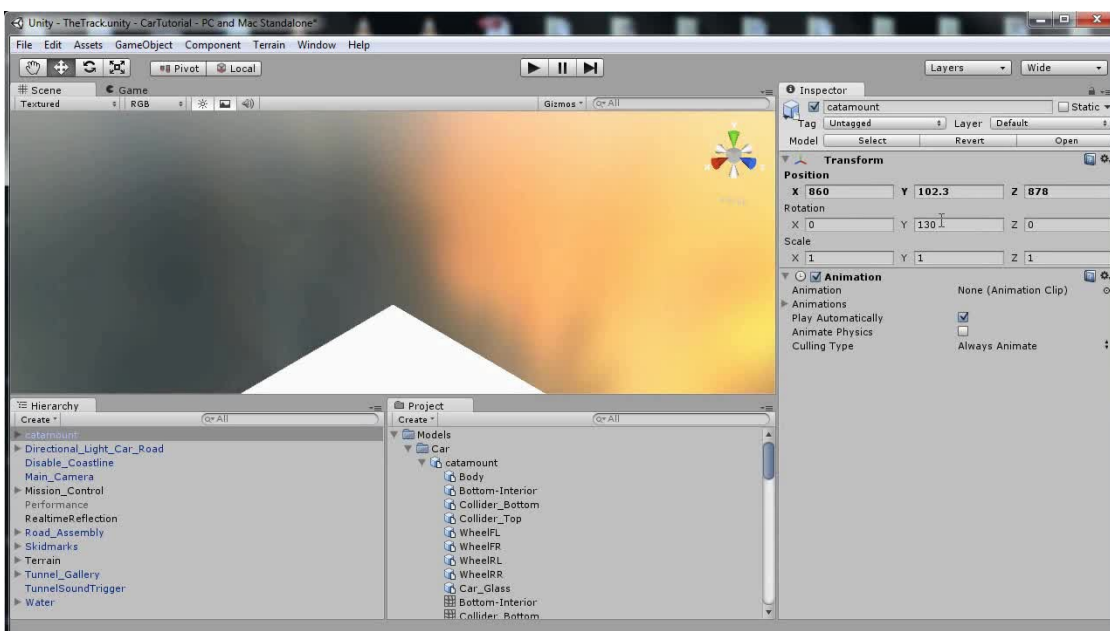


Questa scena contiene tutto quello che serve per una corsa automobilistica, tranne l'elemento più importante: **l'automobile!**

Trasciniamo il modello di automobile, che troviamo nella scheda **Project** al percorso **Models | Cars | catamount**, nella scena.

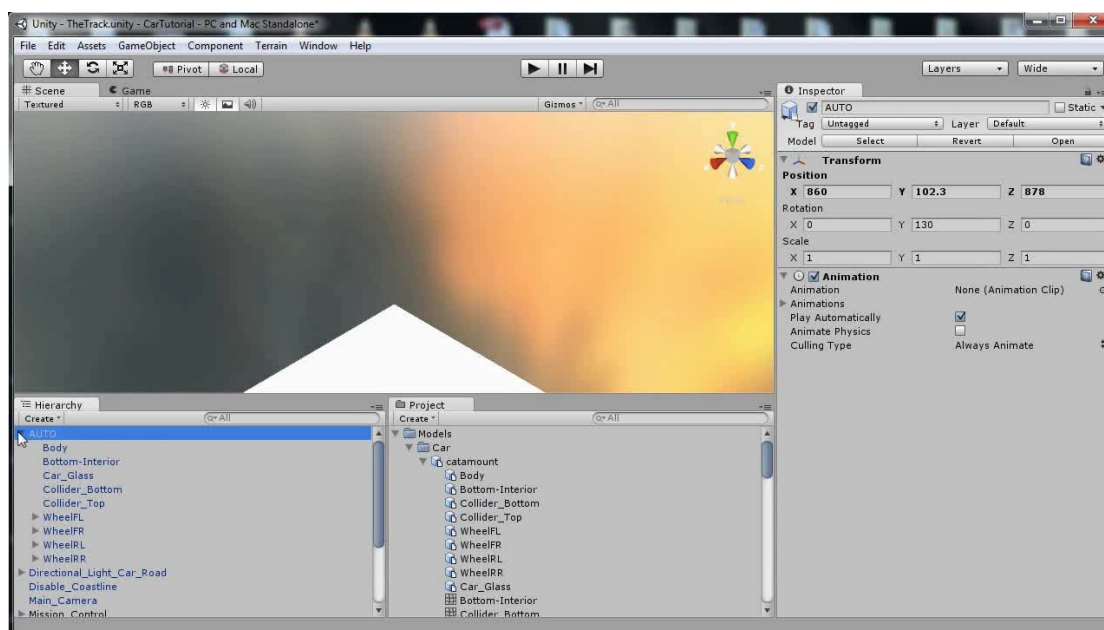


Avremo bisogno di spostare questo modello per posizionarlo correttamente, una volta importato nella scena; specifichiamo, nell'**Inspector**, le posizioni XYZ 860, 102.3, 878, quindi la rotazione Y a 130 gradi.



Per il resto del tutorial, ci riferiremo a questo modello chiamandolo **“AUTO”**, quindi rinominiamolo così nella scena. Questo, comunque, è solo il modello dell'automobile: guardando l'**Inspector**, notiamo che contiene molti elementi figli, come il telaio, i finestrini e le ruote. Contiene anche due semplici **mesh** che useremo come bounding box, “scatole di riferimento” per rilevare e gestire le collisioni (adesso non sono visibili, ma ce ne occuperemo a breve).

Date un'occhiata, anche velocemente, a quali sono le **parti** che compongono l'auto e come sono disposte nel pannello **Gerarchia**.

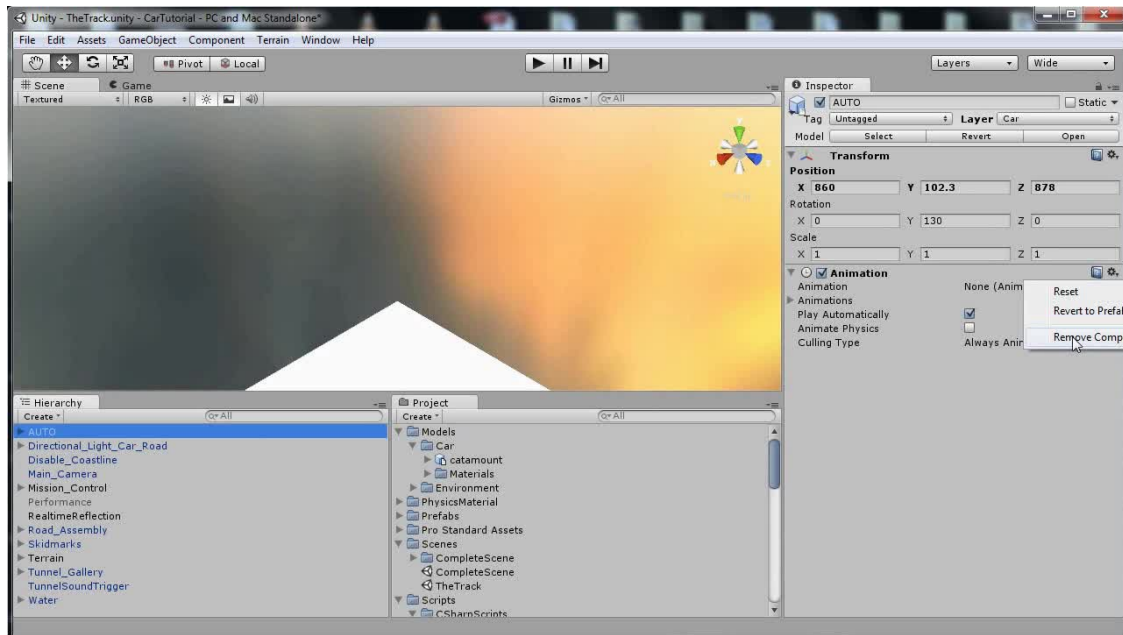


Per vari motivi, che discuteremo in seguito, vogliamo l'auto su un **Layer** tutto suo.

Con il **Game Object AUTO** selezionato, quindi, andiamo nell'**Inspector** e selezioniamo **“CAR”** nel menù **Layers**. Nella finestra di pop-up, scegliamo **“Yes, change children”** (lett.: “sì, cambia (anche) i figli”), per applicare le modifiche a tutti gli oggetti figli nella gerarchia dell'oggetto **AUTO**.

Il **Game Object AUTO** ha un **Component** di animazione.

Dal momento che non lo useremo, facciamo un po' di pulizia cliccando sull'icona della ruota dentata in alto a destra nella scheda del Component Animation e scegliamo **“Remove Component”**, confermando l'operazione (immagine nella pagina seguente).



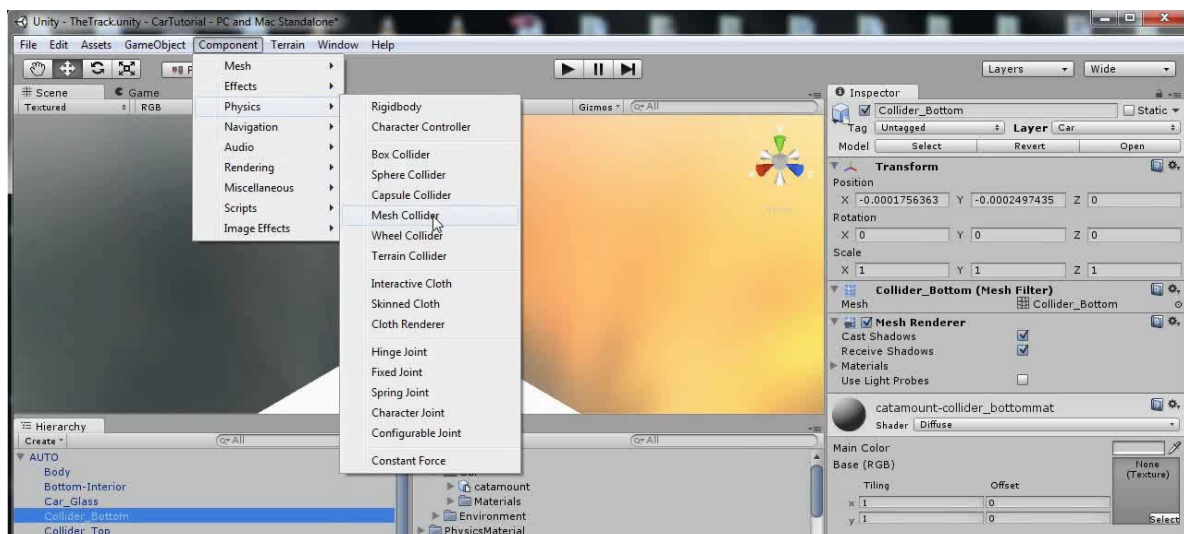
COLLISIONI

Parliamo un po' delle collisioni.

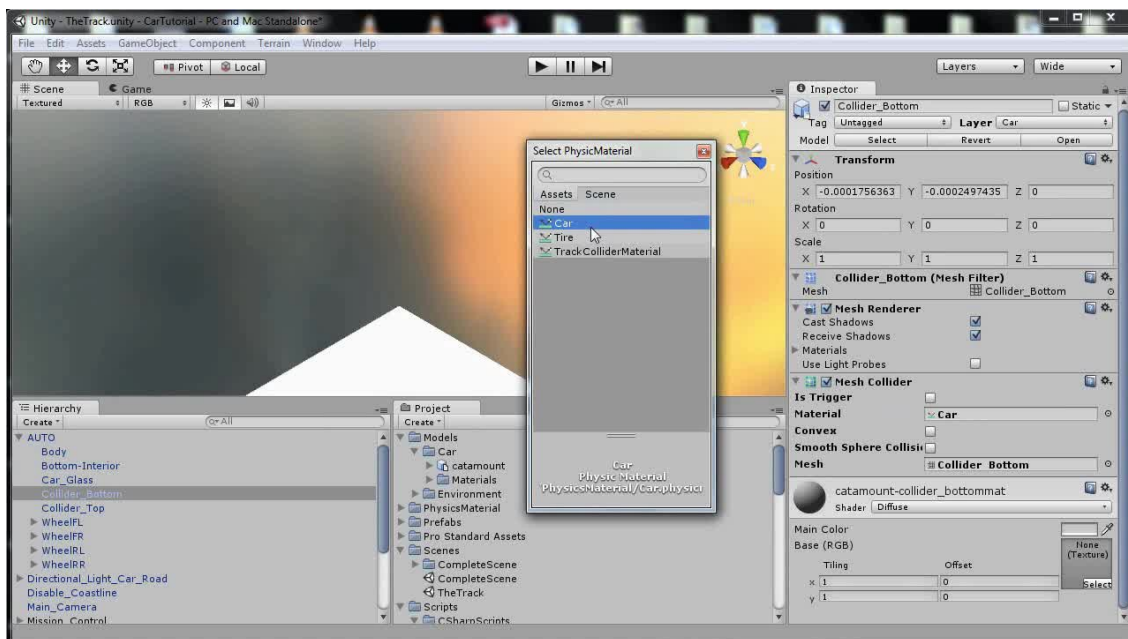
Impostiamo le componenti di collisione dell'auto per impedirle di oltrepassare il terreno durante il viaggio. Anziché usare bounding box complessi, basati sulla forma reale dell'auto, nel Tutorial troviamo, già impostate, **due mesh molto più semplici**, all'interno delle quali rientrano la parte superiore e la parte inferiore dell'auto. Useremo queste mesh come **bounds**, come “estremità” di collisione per migliorare le performance.

Clicchiamo sul **Game Object Collider_Bottom**, figlio dell'oggetto **AUTO**.

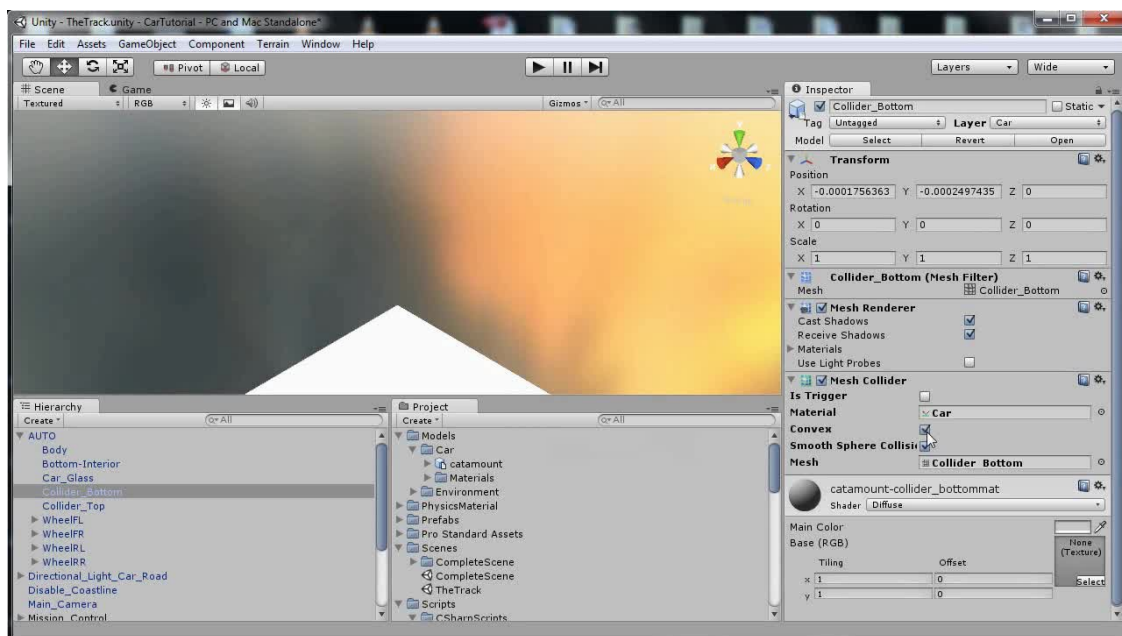
Nel menù **Components | Physics**, scegliamo **MeshCollider** per aggiungerne uno a **Collider_Bottom**.



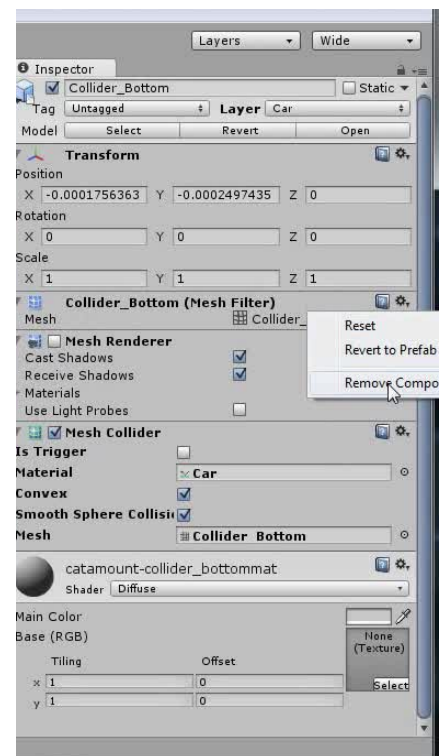
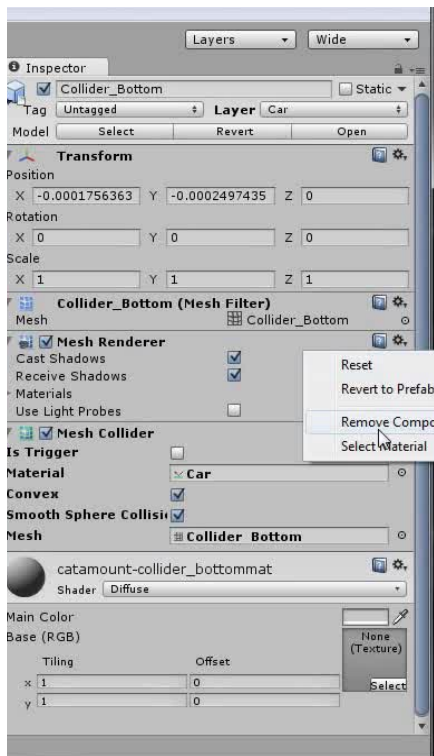
Nella scheda del **Mesh Collider**, scegliamo il materiale fisico dell'*AUTO* nel selettore **Material**.



Selezioniamo le due caselle “*Smooth Sphere Collisions*” e “*Convex*”.



Rimuoviamo il **Mesh Renderer** e il **Mesh Filter** dall'Inspector per il Game Object *Collider_Bottom*: ciò lo renderà invisibile, ma lo lascerà attivo (immagini nella pagina seguente).



Facciamo le stesse operazioni per l'oggetto **Collider_Top**, ossia: clicchiamo su tale oggetto, figlio dell'oggetto **AUTO**; in **Components | Physics**, scegliamo **MeshCollider**; nella scheda di questo componente, scegliamo il materiale fisico dell'**AUTO** nel selettore Material; selezioniamo le due caselle “**Smooth Sphere Collision**” e “**Convex**”; rimuoviamo il **MeshRenderer** e il **MeshFilter**, rendendo anche questo oggetto attivo ma invisibile.



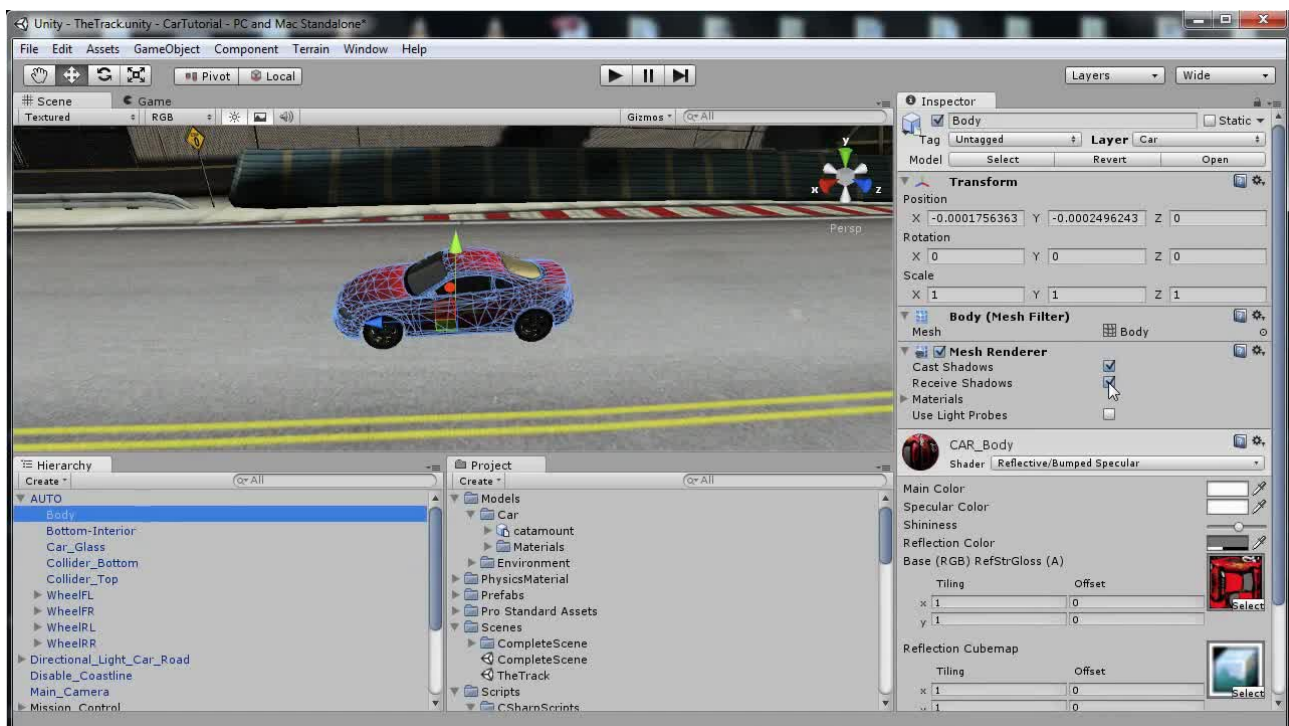
IMPOSTAZIONI OMBRE AUTO

Passiamo ora alle **ombre** dell'auto, che dobbiamo modificare per due motivi.

Per prima cosa, togliere la proiezione e la ricezione delle ombre per gli oggetti che non ne hanno bisogno è un buon metodo per **migliorare le prestazioni** di gioco.

In secondo luogo, è **meglio non proiettare ombre sull'auto**. Potrebbe essere più piacevole, alla vista, non avere ombre che appaiono e scompaiono continuamente sull'auto quando questa viaggia a tutta velocità (ovviamente, siete liberi di scegliere, in base ai vostri gusti; comunque, se volete ombre realistiche, potete ottenerle: dovete solo cambiare le impostazioni, a piacere).

Le impostazioni che useremo per le varie parti dell'auto sono **“Cast Shadows” ENABLED** (ossia: proietta ombre dell'auto su altri elementi, attivato) e **“Receive Shadows” DISABLED** (ossia: ricevi ombre da altri oggetti sull'auto, disattivato); in particolare, dobbiamo impostare così queste due voci per gli elementi: *BODY*, *BODY INTERIOR*, *CAR WINDOWS*, *DISC BRAKES* e *WHEELS* (sostanzialmente, telaio, interni, cristalli, freni e ruote).

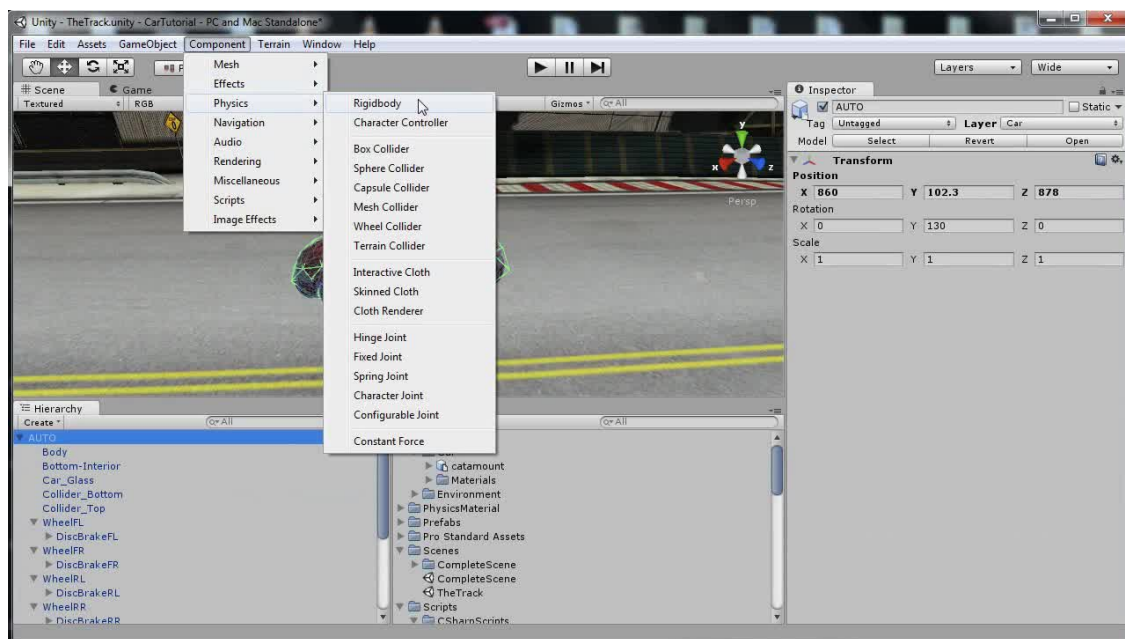


AGGIUNGERE I COMPONENTS DELL'AUTO

Vediamo ora quali **Components** aggiungere e come.

Per prima cosa, ci serve un modo per controllare la posizione dell'auto nel mondo virtuale mediante la simulazione fisica; il **Component Rigid Body**, fornito di serie, fa al caso nostro.

Con il **Game Object AUTO** selezionato, andiamo nel menù **COMPONENTS** e scegliamo **PHYSICS | RIGID BODY**.



Adesso, il **Game Object** avrà un componente **Rigid Body**.

L'auto ovviamente pesa più di 1 kg, quindi iniziamo impostando **MASS** nel **Rigid Body** a qualcosa di più realistico, tipo 1.500.



DRAG e **ANGULARDRAG** simulano le forze di attrito, per rallentare quindi lo spostamento e la rotazione del **Rigid Body**. Impostiamole entrambe a 0: le regoleremo mediante gli script.

Trasciniamo il file “**Car.js**”, dalla cartella “scripts – JavaScript”, sul **Game Object AUTO**.

È questo il “motore” dell'auto, che tratteremo in maniera approfondita nel terzo e ultimo tutorial di questa serie.

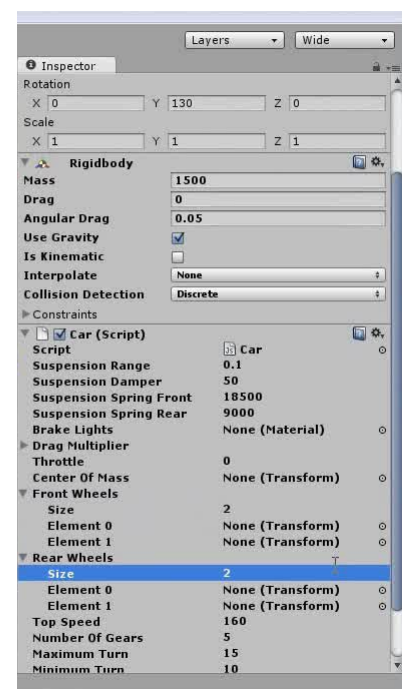


Il componente script dell'auto ha molte variabili, che esamineremo quando inizieremo a personalizzarla e, ancora meglio, quando daremo un'occhiata al codice. Per adesso, vogliamo semplicemente impostare quello che serve per far fare un veloce “giro di prova” alla nostra auto.

Per prima cosa, lo script deve sapere **quali sono le ruote** che l'auto ha a disposizione.

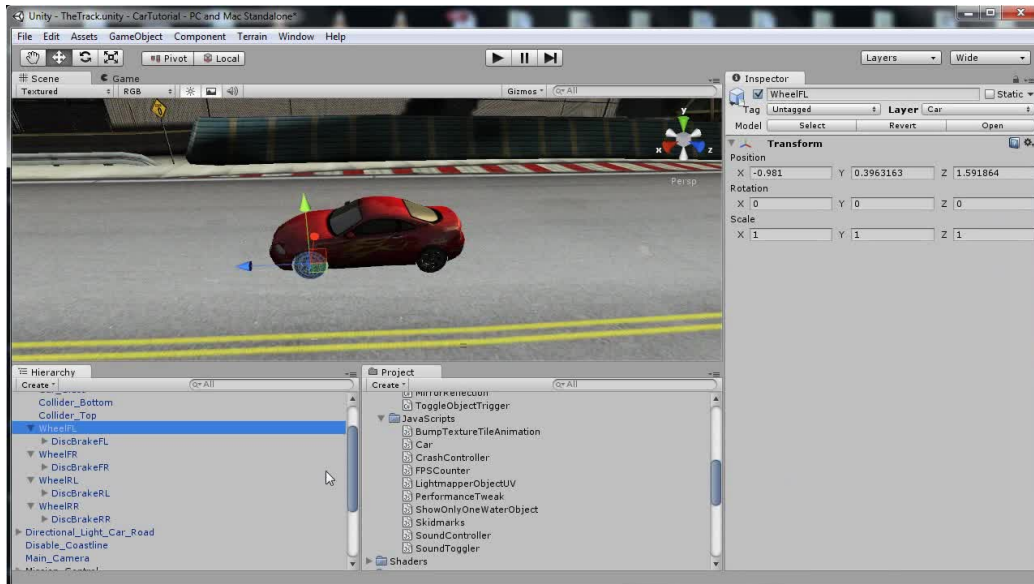
Nell'**Inspector** troviamo le sezioni **Front Wheels** e **Rear Wheels**; entrambe possono essere aperte con un click sulle frecce accanto ai nomi.

Impostiamo, per entrambe, **SIZE** a 2, per mettere una coppia di ruote avanti e una dietro.



Adesso apriamo le sezioni *WheelFL*, *WheelFR*, *WheelRL* e *WheelRR*, nell'Inspector.

Noteremo che ciascuna ha un **DiscBrake** come figlio. Per il momento, ci basta sapere solo che i **Game Objects** *Wheel* e *DiscBrake* sono le rappresentazioni grafiche della ruota nel gioco.



Trasciniamo **DiscBrakeFL** e **DiscBrakeFR** nei due slots *FrontWheels* nello script dell'auto e **DiscBrakeRL** e **DiscBrakeRR** in *RearWheels*.



Perché i **DiscBrakes** sono le ruote? Il motivo è semplice: i **DiscBrakes**, che sarebbero i freni, nella **Gerarchia** del nostro oggetto sono oggetti di gioco genitori degli oggetti (visivi) ruote (cioè i copertoni), quindi impostare i **DiscBrakes** come riferimenti per le ruote includerà anche i copertoni (che, visivamente, sono le ruote).

“A BLOB SHADOW”

Per quanto riguarda l'ombra dell'auto sul terreno, si tratterà più che altro di una “macchia” (un “blob”, come dice il titolo di questa sezione), non un'ombra precisa. Per questo motivo useremo un **Projector**, che... **proietta una macchia**, rappresentante la sagoma dell'auto sulla strada sotto il modello.

Un **Projector** è un **Component** fornito di serie e che si comporta proprio **come un proiettore reale**: si specifica la **Texture** da proiettare e, in base alle impostazioni del proiettore e alla distanza dei bersagli, l'immagine verrà proiettata sugli oggetti. Vediamo quindi quali sono i passi da seguire per mettere in pratica quanto detto.

Nel pannello **Gerarchia**, creiamo un **Game Object Empty** e trasciniamolo su **AUTO** per renderlo **figlio dell'automobile**.

Rinominiamolo “**Blob Shadow Projector**” (proiettore di ombra a macchia).

Aggiungiamo un **componente Projector** (*Component | Effects | Projector*) all'oggetto **Blob Shadow Projector**.



Impostiamo i campi del proiettore: **Near Clip Plane 0.1, Far Clip Plane 50, Field of View 30.**



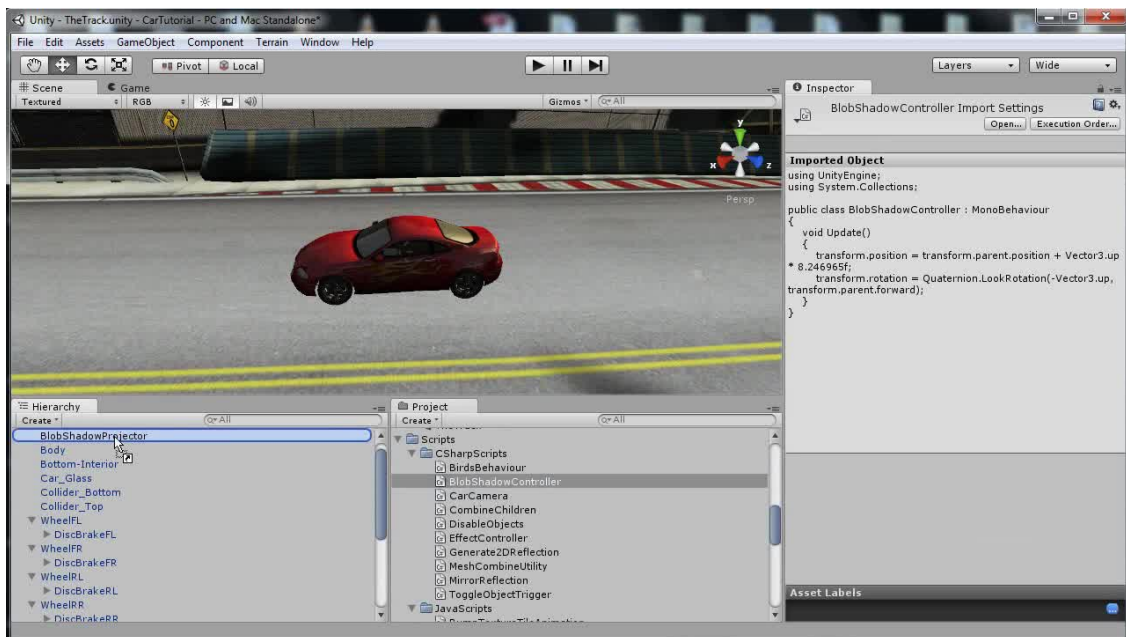
Assegniamo il **Material *Blob_shadow*** allo slot **Material** dell'oggetto.



Nella sezione “**Ignore Layers**”, selezionare “**Everything**” e deseleggiamo il layer “**Road**”; il proiettore, quindi, **proietterà l'ombra solo sul terreno** (immagine nella pagina seguente).



Aggiungiamo lo script *BlobShadowController.js* (cartella *Scripts / CSharpScripts* nella scheda *Project*).



La posizione e l'orientamento del componente **Projector** vengono aggiornate ad ogni frame nello script di **Blob Shadow Controller**, che è abbastanza semplice; in breve, è posto 10 metri sopra l'auto e basa la sua rotazione su quella del veicolo. È possibile dare un'occhiata allo script e provare a cambiare i valori se vogliamo far apparire l'ombra dell'auto sul terreno in maniera diversa: potete sperimentare a piacere.

STRISCE SULL'ASFALTO (“SKIDMARKS”)

Passiamo ora alle strisce delle frenate sull'asfalto, dette **“Skidmarks”**.

Una delle cose divertenti nel guidare quest'auto consiste nel farla **slittare in curva** o farle fare **un'inversione a U** ad alta velocità.

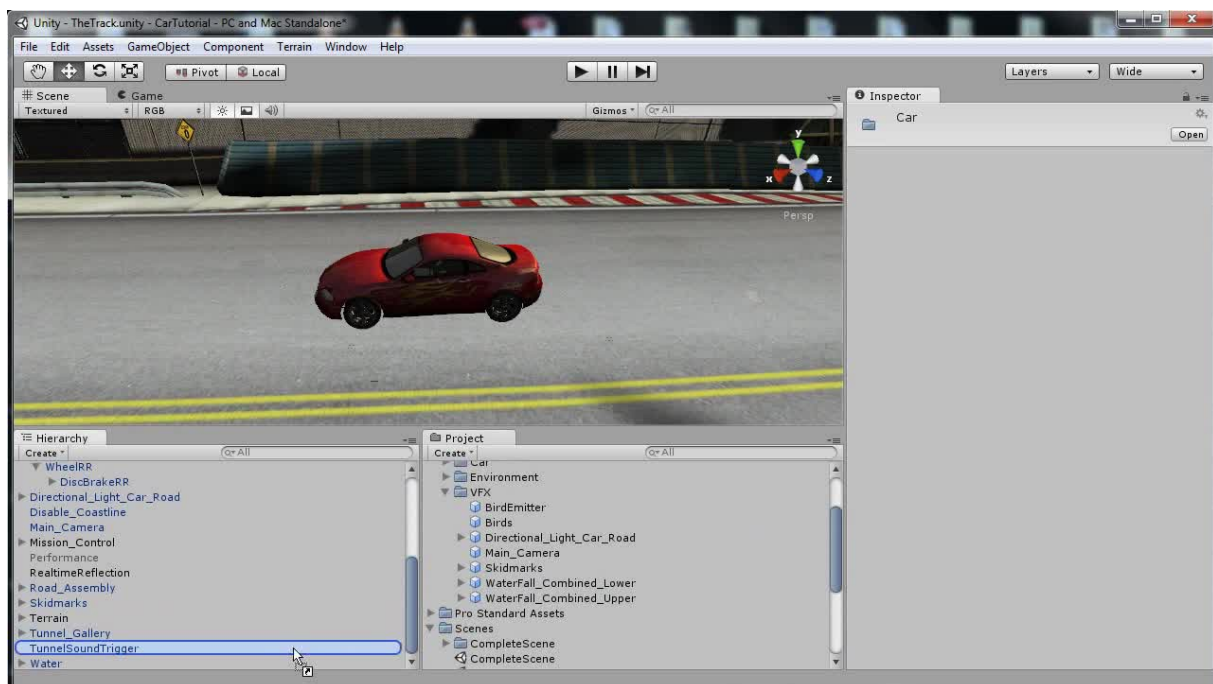
Per enfatizzare l'effetto delle ruote che slittano sulla strada e renderlo un po' più realistico, aggiungeremo delle **strisce sulla superficie** sulla quale si trova l'auto quando scivola.

Per impostare le strisce, **ci serve uno script** che controlli tutte le strisciate della scena. Questo script creerà e controllerà **le mesh che rappresentano le strisce**.

Ogni ruota conosce la sua posizione e “sa” se sta creando o meno delle strisce (ossia, c'è una **variabile-stato** che indica se in quel momento è in atto una “strisciata”). Useremo questa informazione per registrare una posizione nell'universo virtuale per sapere, in relazione al **controller delle strisce**, dove far apparire queste ultime.

Ogni ruota terrà traccia delle strisce precedenti, così le strisce di varie ruote non si mischieranno.

Nel pacchetto scaricato è già presente un **Prefab** per questo scopo, quindi bisogna solo trascinarlo nella scena: trasciniamo, quindi, il **Prefab Skidmarks da Prefabs | VFX | Skidmarks** all'interno della scena 3D.



Imposteremo l'intensità delle strisce cambiando i colori dei vertici creati. L'intensità è data dall'entità dello slittamento della ruota. Per metterlo in pratica, useremo uno **shader personalizzato** che userà il valore **Alpha (trasparenza)** dei vari vertici in fase di rendering.

Come per l'auto, ci sono molti parametri che possono essere modificati per migliorare la resa delle strisce. Nella sezione **Skidmarks Component** del **Game Object Skidmarks** troviamo alcune variabili pubbliche personalizzabili: vediamone alcune.



MAX MARKS

Questo parametro determina il numero massimo di posizioni (e, quindi, di strisce) che possono esserci all'interno della scena. Tutte le ruote usano lo stesso **Skidmarks**, quindi il valore vale per tutte le ruote.

Quando si raggiunge il numero massimo, le posizioni (ossia, le strisce) vengono sovrascritte in memoria, a partire dalle prime, che vengono quindi eliminate dalla scena.

MARK WIDTH

Questo parametro determina invece **la larghezza delle strisce** (uguali per tutte le strisciate).

Va impostato in base alle dimensioni delle ruote del veicolo, in maniera credibile: per un monster truck metteremo un valore alto, mentre lo abbasseremo sensibilmente per un'auto-razzo che deve infrangere il record di velocità su terra!

GROUND OFFSET

Quando la mesh delle strisce viene creata, ha le **coordinate** calcolate partendo dalla **posizione della ruota**. Questi punti saranno, quindi, sulla superficie sulla quale si trova la ruota al momento della strisciata. Quando due mesh sono una sull'altra, in fase di **rendering** potrebbero esserci degli **sfarfallii**.

Per evitare questo problema, abbiamo la variabile **GroundOffset**. La mesh creata verrà quindi **“alzata”** nella misura data dall'*Offset* nella direzione della normale della superficie sulla quale è creata; quindi, se ci sono degli sfarfallii o le mesh finiscono una dentro l'altra, provate a incrementare questo valore.

Per essere ancora più sicuri di renderizzare le strisce ad una certa distanza dal terreno, anche lo **Shader** che le rappresenta ha un **offset**, e fa riferimento ad un'altra pipeline di rendering, diversa da quella standard.

MIN DISTANCE

Quando un nuovo punto viene aggiunto al **controller Skidmark**, deve trovarsi ad una distanza minima dall'ultima striscia creata dalla ruota. In questo modo non c'è il rischio di sprecare punti (ossia strisce) creando molte strisce in uno spazio ristretto.

Il valore, comunque, **non deve essere troppo alto**, perché finirebbe per creare strisce troppo lunghe; inoltre, la distanza potrebbe essere percepita come un **lag**, un ritardo, in quanto le strisce apparirebbero molto più in là rispetto alle ruote (cioè, appunto, in ritardo). Abbassare il valore avvicinerà le strisce al punto di inizio frenata (o slittamento).

TEXTURE

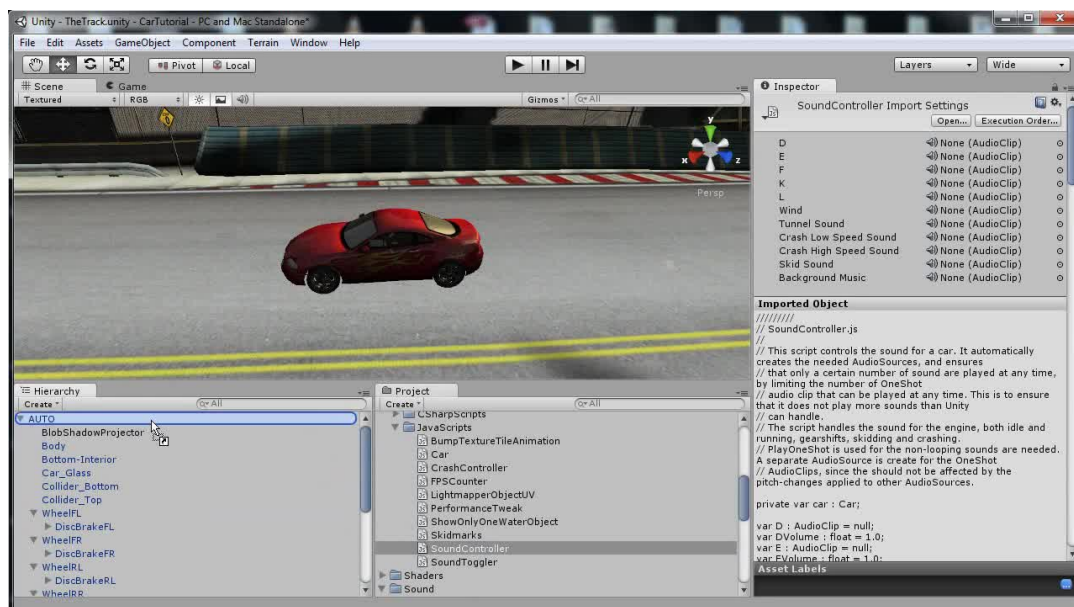
Il **pattern dei copertoni** di un monster truck è diverso da quello di un'auto di formula 1; la **texture** usata per le strisce, quindi, deve rifarsi all'aspetto dei copertoni che producono le strisce stesse.

Se creiamo un altro tipo di auto, con ruote molto diverse, **dovremo cambiare questa texture** per rendere le strisce coerenti con i copertoni delle ruote.

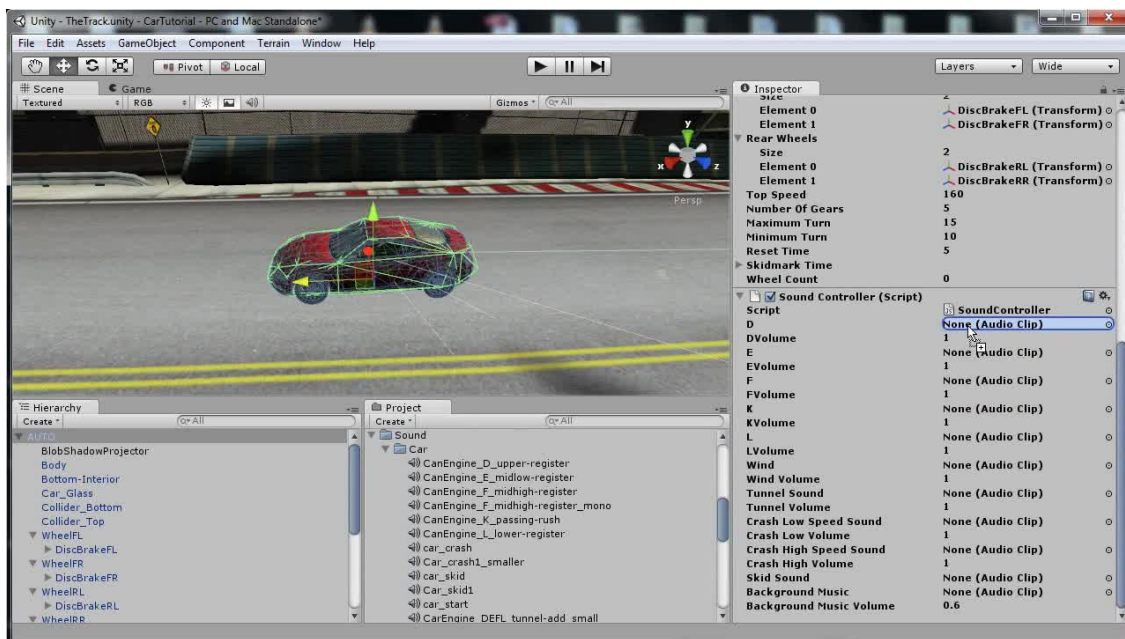


AGGIUNGERE I SUONI

Adesso, vediamo come vengono aggiunti e gestiti i file audio. Aggiungiamo, per prima cosa, lo script **SoundController.js** al Game Object **AUTO** (trascinandolo da *Scripts / JavaScript*).



Questo componente ha diversi slot per i suoni da riprodurre, da associare ad altrettanti file audio. Questi file si trovano nel **Project**, all'interno della cartella **Sound / Car**.



Trasciniamo e impostiamo, in particolare:

- *CarEngine_D_Upper-Register* nello slot *D* con volume 0.565;
- *CarEngine_E_Midlow-Register* nello slot *E* con volume 0.8;
- *CarEngine_F_Midhigh-Register* nello slot *F* con volume 0.78;
- *CarEngine_K_Passing-Rush* nello slot *K* con volume 0.565;
- *CarEngine_L_Lower-Register* nello slot *L* con volume 0.71;
- *Wind-Loop_Stereo_22KHz_16bit* nello slot *WIND* con volume 0.8;
- *CarEngine_DEFL_Tunnel-Add_Small* nello slot *TunnelSound* con volume 0.8;
- *Collision1* nello slot *CrashLowSpeedSound* con volume 0.8;
- *Car_Crash1_Smaller* nello slot *CrashHighSpeedSound* con volume 0.5;
- *Car_Skid1* nello slot *SkidSound*
- *CarTutorialSound* nello slot *BackgroundMusic* con volume 1.

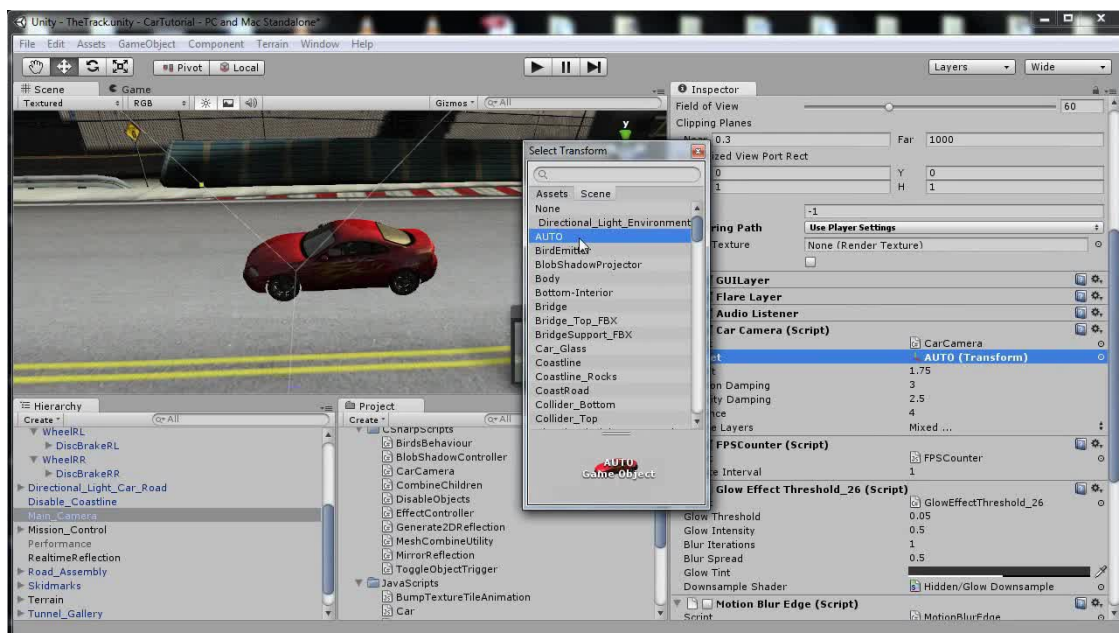
I valori del **Volume** sono solo **suggerimenti**, potete cambiarli per ottenere lo schema sonoro che desiderate o spingervi oltre e creare un menù per le impostazioni auto interattive per l'utente, da utilizzare cioè durante il gioco per variare i parametri durante l'esecuzione; **sui file audio**, invece, torneremo in futuro, spiegando tra l'altro perché ce ne sono **così tanti solo per il motore**.

FINALIZZAZIONE

Per finire, trasciniamo i seguenti script sul **Game Object AUTO**: *“LightmapperObjectUV.js”*, *“CrashController.js”* e *“Generate2DReflection.cs”*.



L'ultima cosa da fare è **impostare la telecamera**, che ancora non sa **cosa inquadrare e seguire**: nella sezione **script CarCamera** del **Game Object Main Camera**, impostiamo come **TARGET** il **Game Object AUTO**, concludendo qui il primo tutorial sul *Car Project*.



* * *

Il Car Tutorial – Parte 2

Se avete seguito il primo tutorial e **assemblato l'auto**, quest'ultima dovrebbe mostrare già delle **caratteristiche interessanti**, in fase di guida; tuttavia, avrete anche notato che c'è molto spazio per personalizzazioni e migliorie.

Eccoci quindi al “**tweaking**”: una fase cruciale dello sviluppo di videogiochi per renderli divertenti, avvincenti o comunque “**così come li vogliamo**”.

Il **tweaking** entra in scena dopo aver messo insieme i pezzi (non solo gli oggetti fisici, ma anche script e logica di gioco) che “grosso modo” costituiscono il gioco, passando ad **una fase di rifinitura**, ad esempio perché ci si accorge che la manovrabilità dell'auto, nel nostro esempio, va migliorata, il parametro della velocità massima va cambiato, l'illuminazione generale della scena deve essere modificata, e così via.

La “**tweakability**” è uno dei punti di forza di **Unity 3D**: come sappiamo, tutte le **variabili pubbliche** degli script sono visibili nell'**Inspector**, per cui è possibile cambiare i valori senza mettere mano al codice; inoltre, fatto un cambiamento, basta premere **PLAY** per vedere gli effetti nell'editor, quindi non c'è bisogno di aspettare una ricompilazione della scena o del codice.

CENTRO DI MASSA

Guidando, ci accorgiamo che l'auto viene **facilmente sballottata a destra e a sinistra** ad ogni sterzata. Il tutto nasce dal non aver ancora impostato il **centro di massa dell'auto**, quindi è questa la prima cosa che dobbiamo fare.

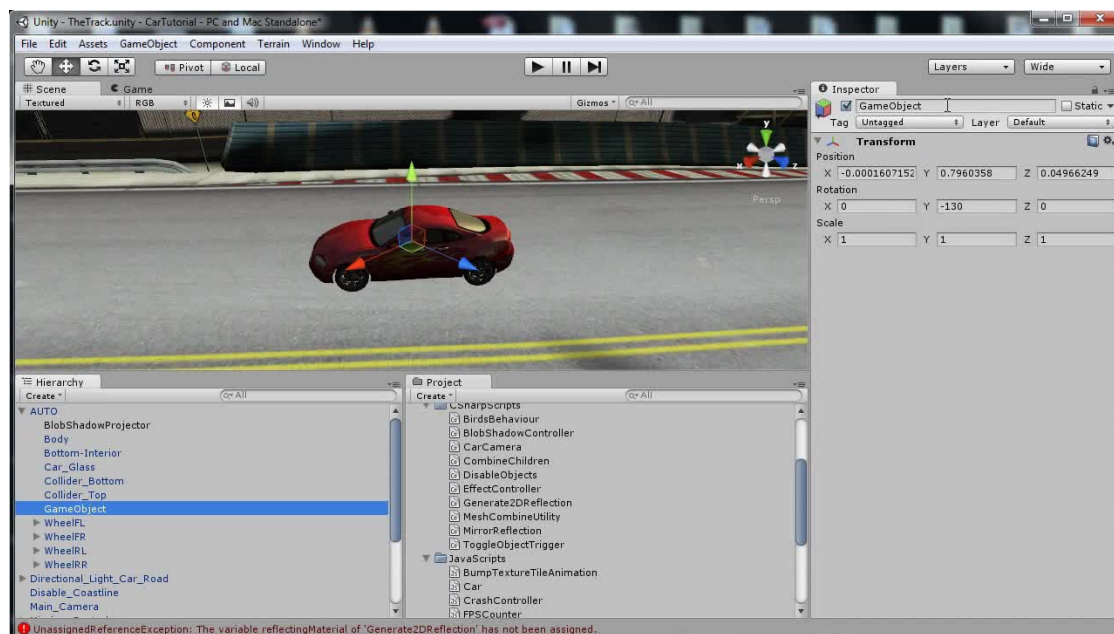
Tutte le **forze applicate al Rigid Body** dell'auto vengono applicate al centro di massa di tale elemento; tale centro, inoltre, viene calcolato e definito da **Unity** in base ai **Colliders** agganciati al **Rigid Body**, sia sullo stesso **Game Object** che sugli oggetti figli.

Dal momento che, in genere, **il centro di massa di un'automobile non è il baricentro dell'auto** (e probabilmente non lo è nemmeno il centro di massa calcolato automaticamente da **Unity** sulla base della mesh), vogliamo poterlo definire noi.

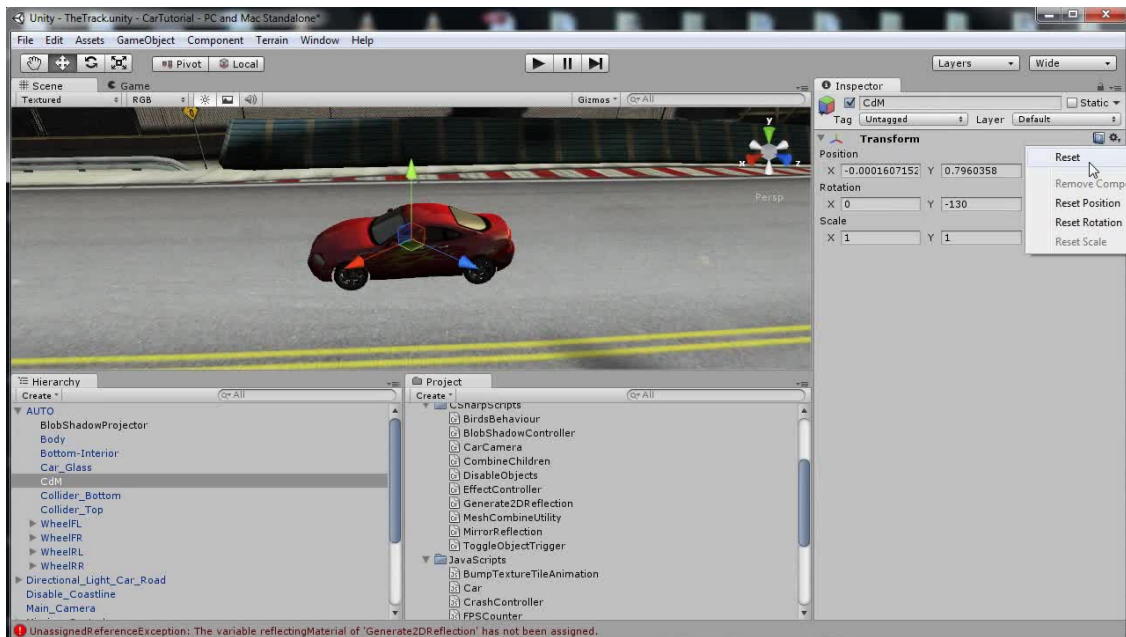
La posizione del centro di massa in un'auto **dipende dalla posizione del motore e di altri elementi** e può variare parecchio da modello a modello. Per questo tutorial, posizioneremo il centro di massa **poco dietro al motore**, leggermente sopra la superficie del terreno.

Creiamo un nuovo **Game Object Empty** e rendiamolo **figlio** del **Game Object AUTO**, trascinandolo su tale elemento nella **Gerarchia**.

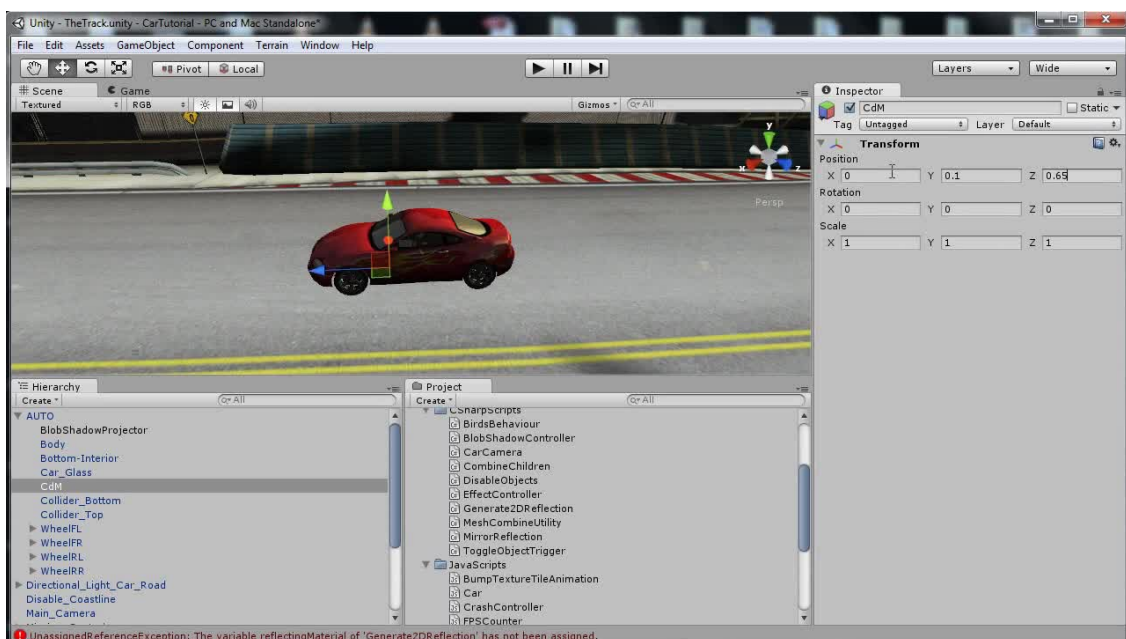
Rinominiamo il nuovo oggetto in **CdM (Centro di Massa)**.



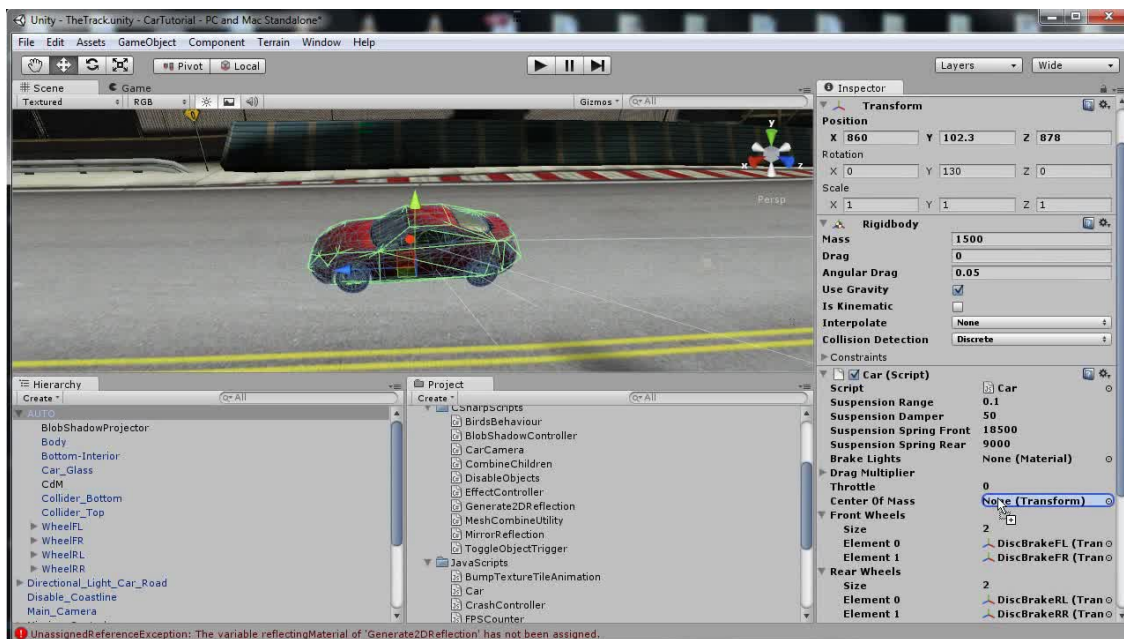
Resettiamo le trasformazioni di CdM (cliccando sull'ingranaggio in *Inspector – Transform* e scegliendo **RESET**); questo darà a **CdM** la stessa posizione di **AUTO**, il suo genitore (immagine nella pagina seguente).



Aggiustiamo la posizione di **CdM** mettendolo dove vogliamo. Possiamo trascinare liberamente o digitare dei valori. Una buona posizione del **CdM** di quest'auto dovrebbe essere 0, 0.1, 0.65.



Nell'Inspector, assegniamo **CdM** al relativo slot nello script di **AUTO** (immagine nella pagina seguente).



In generale, **non è una buona idea** mettere il centro di massa **di lato** rispetto al centro dell'asse X, perché si avrebbero sterzate o sbandate strane, per cui dobbiamo **impostare a 0 la variabile Position X**.

Possiamo anche cambiare il **TARGET** della telecamera, impostando il **Game Object Cdm** anziché **AUTO**, a piacere. Questo produrrà una “sensazione” diversa, durante la guida, per cui è consigliabile giocare un po' e scegliere le impostazioni che si preferiscono.

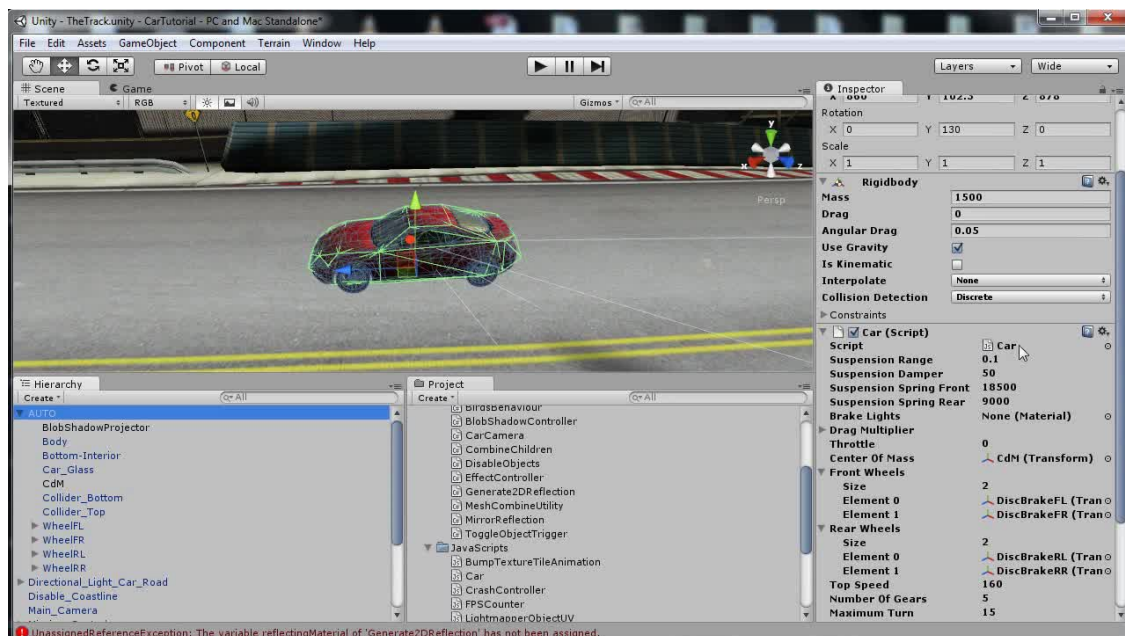
SOSPENSIONI

Un altro fattore che può cambiare profondamente il **comportamento dell'auto** è il blocco sospensioni con le sue caratteristiche. Il compito principale delle **sospensioni** (... beh, almeno, in questo gioco, ecco) è quello di **massimizzare la tenuta di strada**, cosa che contribuisce a garantire la stabilità del veicolo.

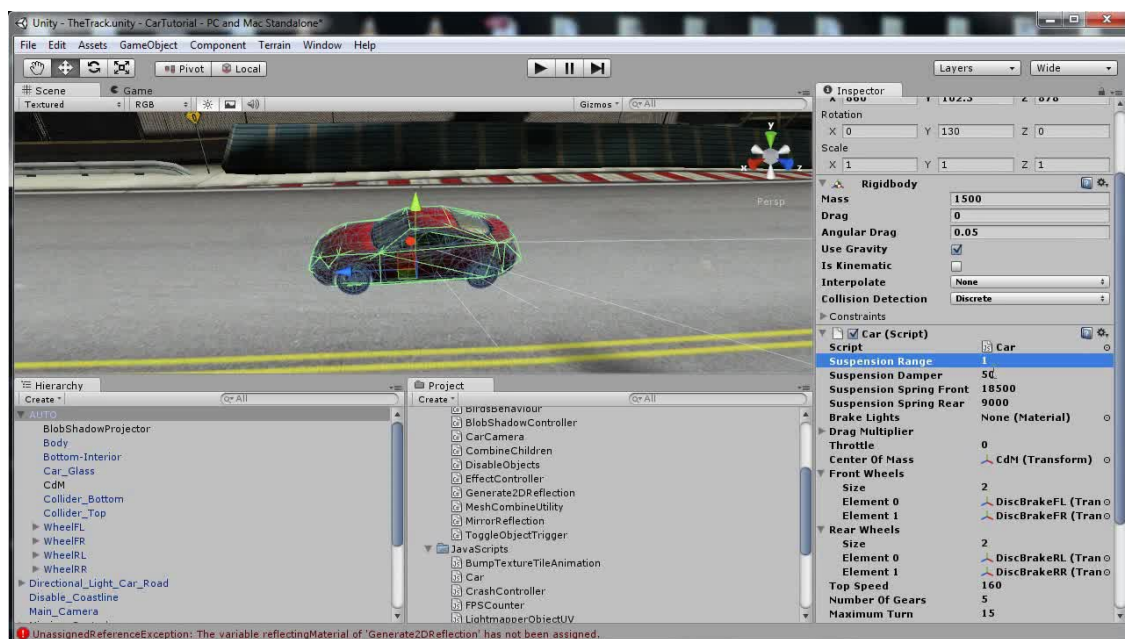
Quando si guida su un dosso o un rilievo, **le sospensioni trasferiscono la spinta verticale** delle ruote al telaio; se non mettiamo questo componente, le ruote possono perdere il contatto col terreno, sollevandosi con l'auto, per poi ricadere bruscamente a terra per via della gravità.

Questo componente è appunto il blocco sospensioni del modello.

Abbiamo tre variabili a riguardo, nell'Inspector: **range**, **damper** e **spring**. Fanno tutte parte della classe **WheelCollider**, usata sulle ruote dell'auto.



Un valore alto di **range** alzerà notevolmente il telaio rispetto alle ruote.



Usando le **tre variabili**, è possibile cambiare anche radicalmente il comportamento dell'auto: da un'auto di Formula 1 a un monster truck; l'aspetto visivo, comunque, deve essere **coerente col comportamento**, per non disorientare il giocatore, quindi attenzione a non esagerare!

SUSPENSION RANGE

Il parametro *Suspension Range* sta per “lunghezza delle sospensioni”, da completamente compresse a massima estensione possibile.

SUSPENSION SPRING

Il parametro *Suspension Spring* determina la **rigidità** (*Stiffness*) della molla della sospensione; con un valore molto alto si avrà la massima distensione, con le ruote molto lontane dal telaio, mentre con valori molto bassi le sospensioni saranno molto più elastiche (*bouncy*).

La massa dell'auto è molto importante nella definizione di questo valore: un'auto pesante richiede sospensioni con una stiffness maggiore rispetto a un'auto leggera. Di default, nell'esempio la stiffness (il valore di Spring) è **maggiore per le ruote anteriori** perché il centro di massa è leggermente in avanti rispetto al baricentro dell'oggetto *AUTO*.

Provare vari valori, sia per il blocco anteriore che per quello posteriore, può portare a risultati molto diversi tra loro.

SUSPENSION DAMPER

Il *Dampening* aiuta a **controllare il movimento delle sospensioni**: senza questo parametro, la molla si estenderebbe e rilascerebbe l'energia in maniera incontrollata, fino a rilasciare brevemente tutta l'energia immagazzinata. Tutto ciò porterebbe ad un'auto **con troppi rimbalzi esagerati**, di fatto incontrollabile.

Il **Damper** (o “*shock controller*”) **trasforma l'energia cinetica** indesiderata in calore liberato dal sistema idraulico, “smorzando” la guida (di fatto, potete considerarlo un **ammortizzatore**).

DRAG e DRAG MULTIPLIER

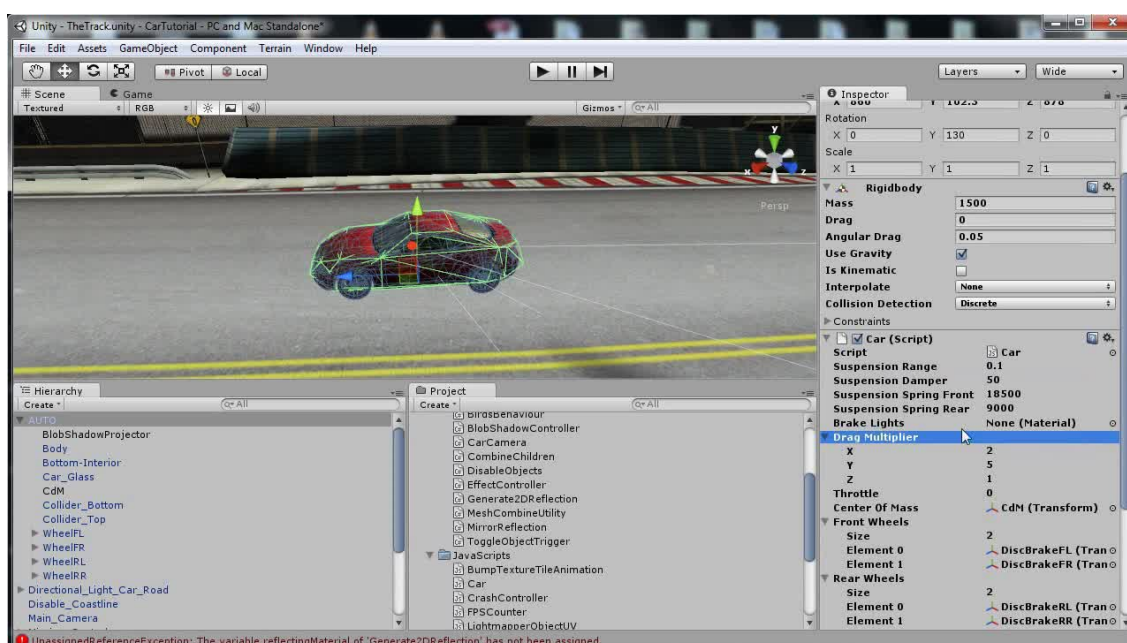
Abbiamo visto che il Rigid Body che abbiamo associato all'auto ha una proprietà **DRAG**.

Si tratta **dell'inerzia o dell'attrito/resistenza** (anche resistenza dell'aria) che rende il Rigid Body

difficile da spostare. Nella progettazione dell'auto, il **profilo aerodinamico** – per ridurre la resistenza dell'aria, durante il movimento – gioca un ruolo importante e va tenuto in considerazione.

Nel nostro progetto, ciò si traduce nella proprietà **DRAG MULTIPLIER**, che useremo al posto della proprietà **DRAG** del **Rigid Body**.

Nell'**Inspector**, per il componente script di **AUTO**, vediamo che la variabile **DRAG MULTIPLIER** è un vettore di tre elementi (x, y e z). Ciò ci consente di impostare valori **DRAG** diversi per le tre direzioni dell'auto, come avviene nella realtà, ottenendo una simulazione accurata; in particolare: x lateralmente, y verticalmente, z frontalmente.

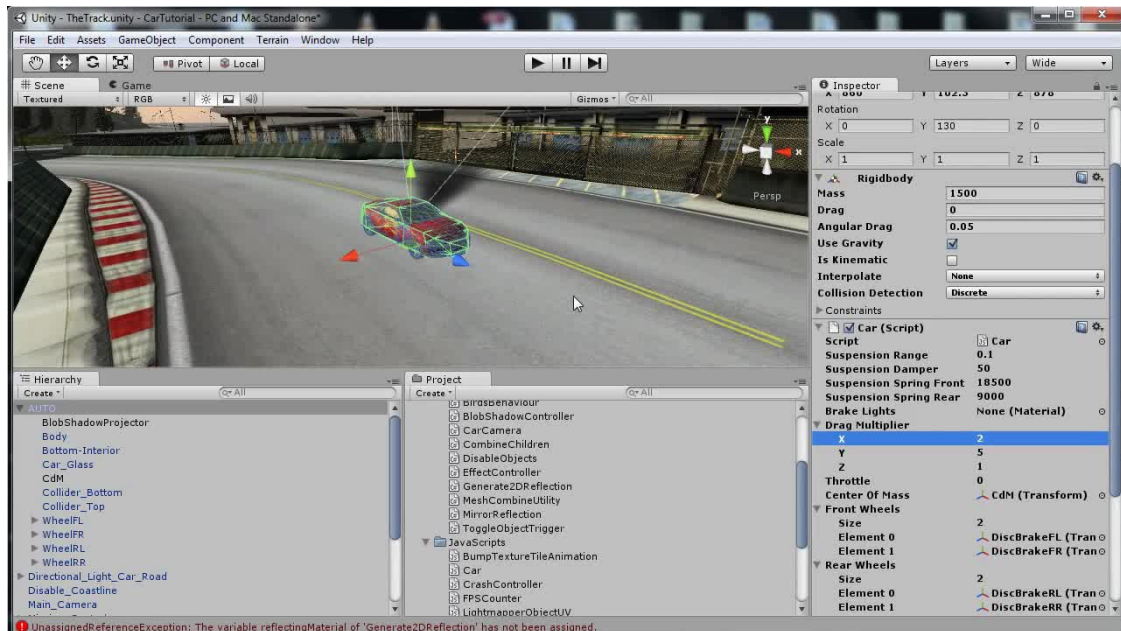


La **x** è quindi la **resistenza laterale**, che impedisce all'auto di slittare o sbandare in curva.

Il valore più interessante è ovviamente **z**, perché **influisce su accelerazione e frenata**.

Con un valore minore di 1 si hanno minor resistenza, maggiore accelerazione e velocità massima più elevata, mentre con un valore maggiore di 1 l'auto dovrà fare un certo “sforzo” per avanzare e, comunque, si muoverà lentamente.

Il parametro ha un'elevata sensibilità, quindi bisogna provare variazioni minime. L'auto non effettua, di per sé, movimenti in verticale, quindi **il valore y non è importante**: la forza più importante su quell'asse è, in effetti, la gravità che agisce sul **Rigid Body** collegato all'AUTO.

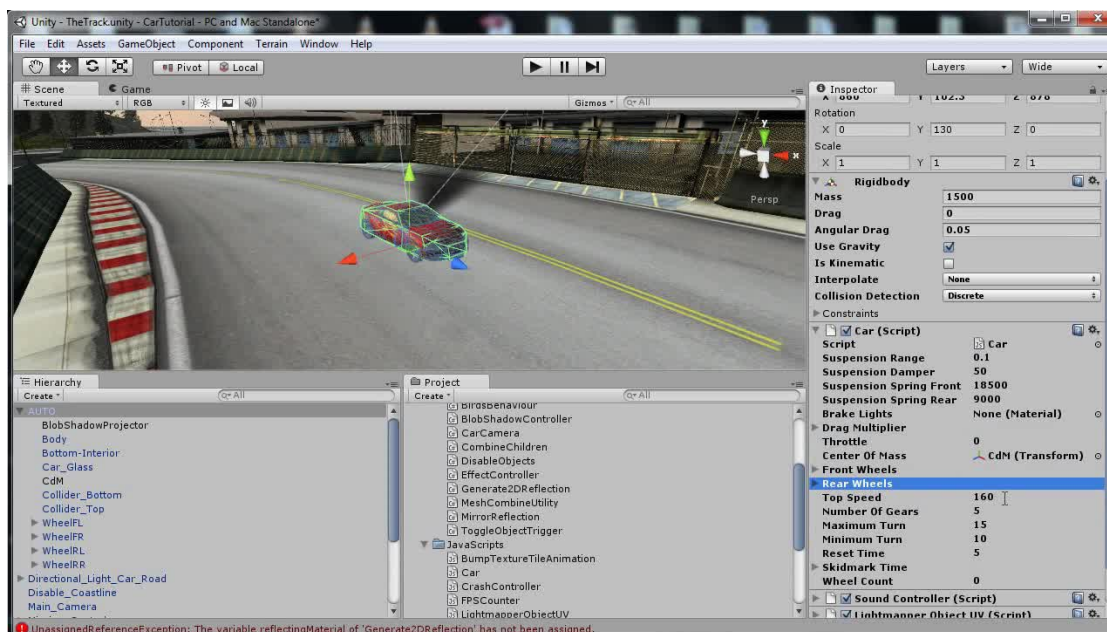


VELOCITA', INVERSIONI E MARCE (RAPPORTI)

Eccoci alle variabili più ovvie ma che, in effetti, hanno l'impatto maggiore sul comportamento dell'auto.

Il valore **TOP SPEED** è solo un limite di velocità massima raggiungibile. Il nostro modello è molto più semplice di quello di una vera auto e, in effetti, **DRAG** e **TOP SPEED** sono gli unici due parametri che utilizzeremo per **influenzare indirettamente l'accelerazione**, quindi per provare i diversi comportamenti / effetti bisogna agire su questi valori.

Per le sterzate, abbiamo due valori: **Maximum Turn** e **Minimum Turn**. In entrambi i casi, un valore alto indica un'ottima capacità di sterzata, mentre un valore basso indica capacità molto ridotte.



I due valori vengono usati in combinazione nello script di **AUTO** e agiscono sulla capacità di sterzata dell'auto in base alla velocità:

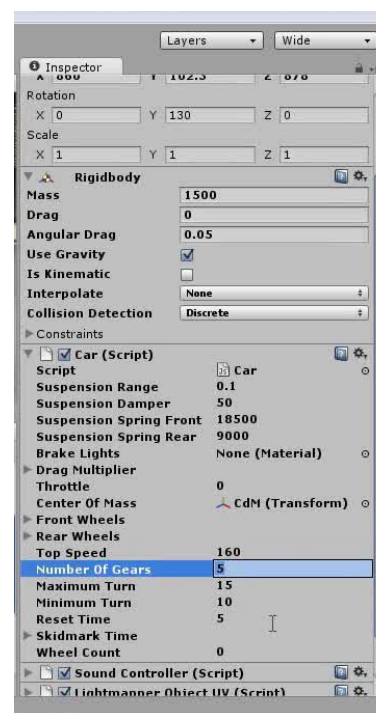
- a velocità molto basse, viene usato il valore di **MaximumTurn**;
- a velocità via via crescenti, il valore usato si avvicina sempre più a quello di **MinimumTurn**.

In conclusione, **più si va veloci e più sarà difficile sterzare**.

I valori di default per **MIN** e **MAX** sono rispettivamente 10 e 15, ma ovviamente si consiglia di **fare delle prove** cambiandoli per valutare altri effetti e comportamenti.

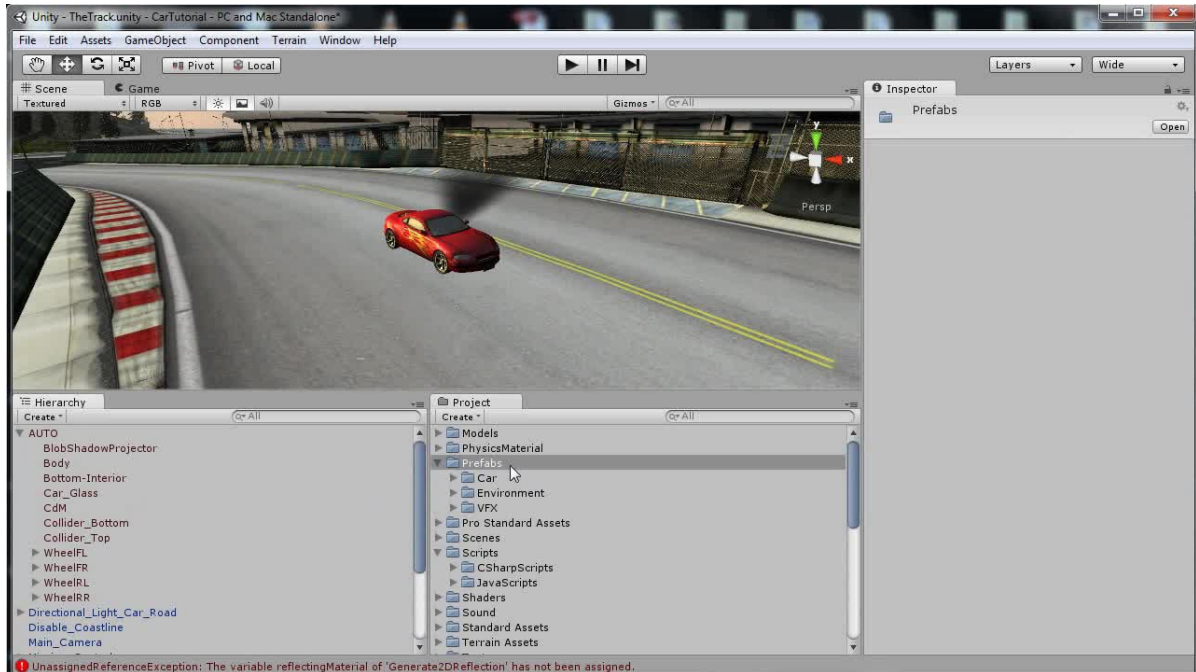
Infine, i **rapporti (GEARS)**. Nella terza parte, riguardante lo script, vedremo cosa viene calcolato usando questa variabile. Essa non simula il comportamento reale, ma **serve per calcolare la forza del motore** e, soprattutto, a **cambiare il suono da riprodurre** per il motore, con il famoso incremento dei giri per minuto e abbassamento da una marcia alla successiva.

Cambiare questo valore darà **l'impressione** (dovuta in realtà agli effetti sonori) di un **diverso numero di marce a disposizione** e dell'atto del cambio di marcia.



ESPORTARE L'AUTO COME PACKAGE

Un modo semplice per utilizzare l'AUTO in altri progetti è quello di realizzare un **Package Unity** partendo dai **Prefabs** richiesti.

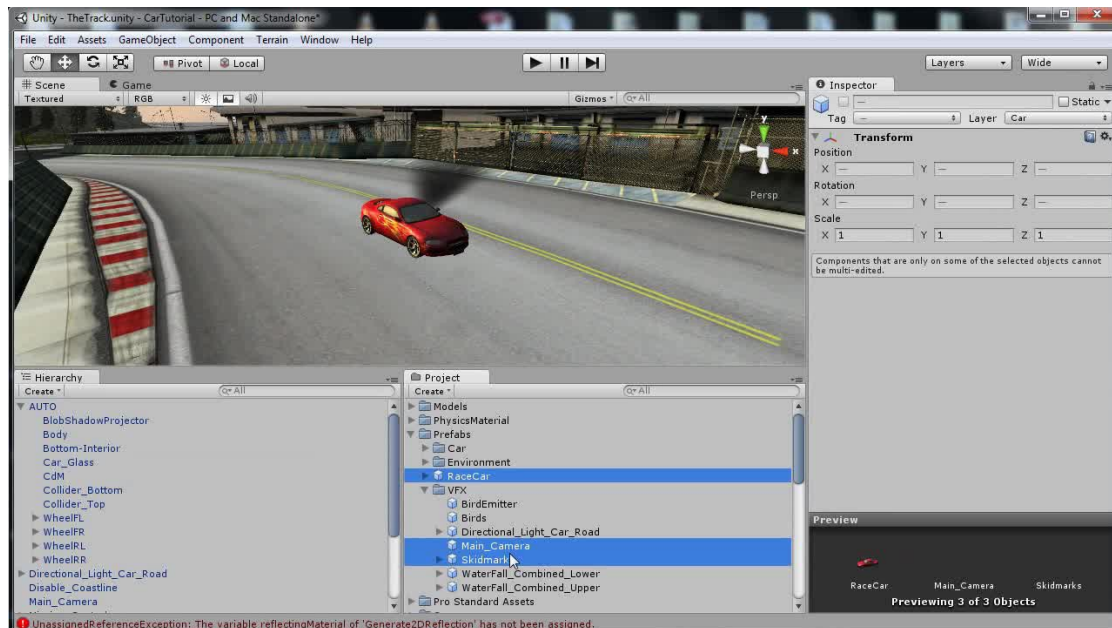


Per prima cosa, dobbiamo **rendere l'auto così definita un Prefab**, così non dovremo rifare assemblaggi e tweaking vari, quindi:

- nella scheda Project, facciamo click destro e scegliamo **CREATE – NEW PREFAB**, ottenendo un nuovo Prefab vuoto chiamato “*new Prefab*”;
- **rinominiamo** il prefab in “Race Car” o qualcosa del genere, a piacere;
- **trasciniamo il Game Object AUTO** dalla Gerarchia sul Prefab “Race Car”.

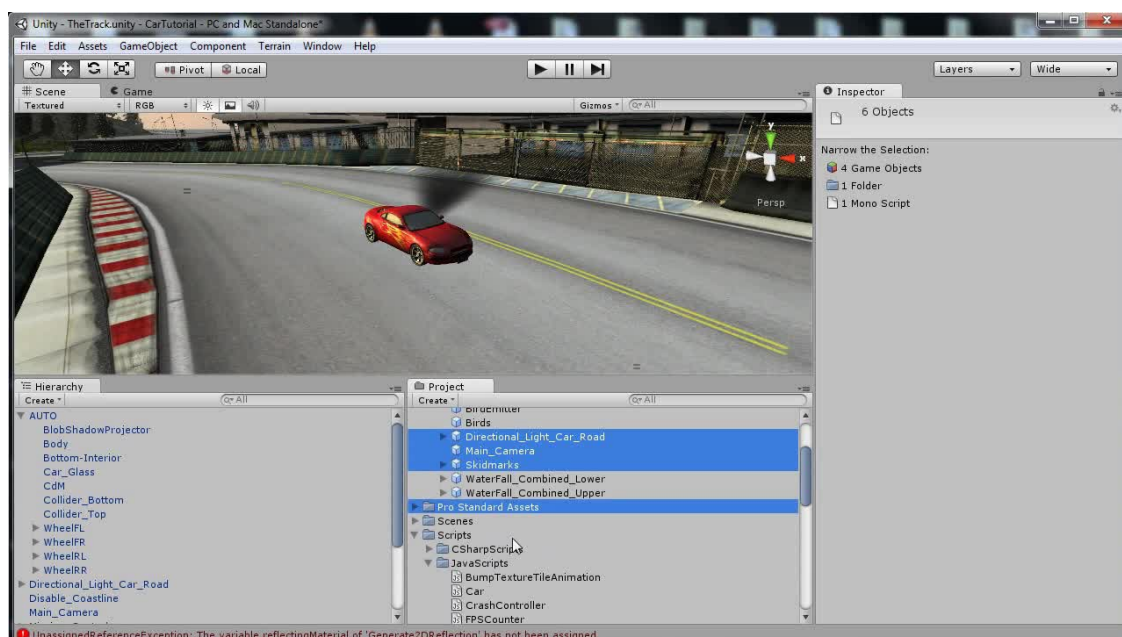
In realtà, c'era già il Prefab di *AUTO*, ma ora... l'abbiamo fatto noi.

Per ottenere, ora, un **Package** completo e funzionante, a parte il Prefab dell'auto dobbiamo **includere certi altri elementi**; questi sono: le strisce, la main camera e *Directional_Light_Car_Road*; per fortuna, **sono già dei Prefabs, nel Progetto**.

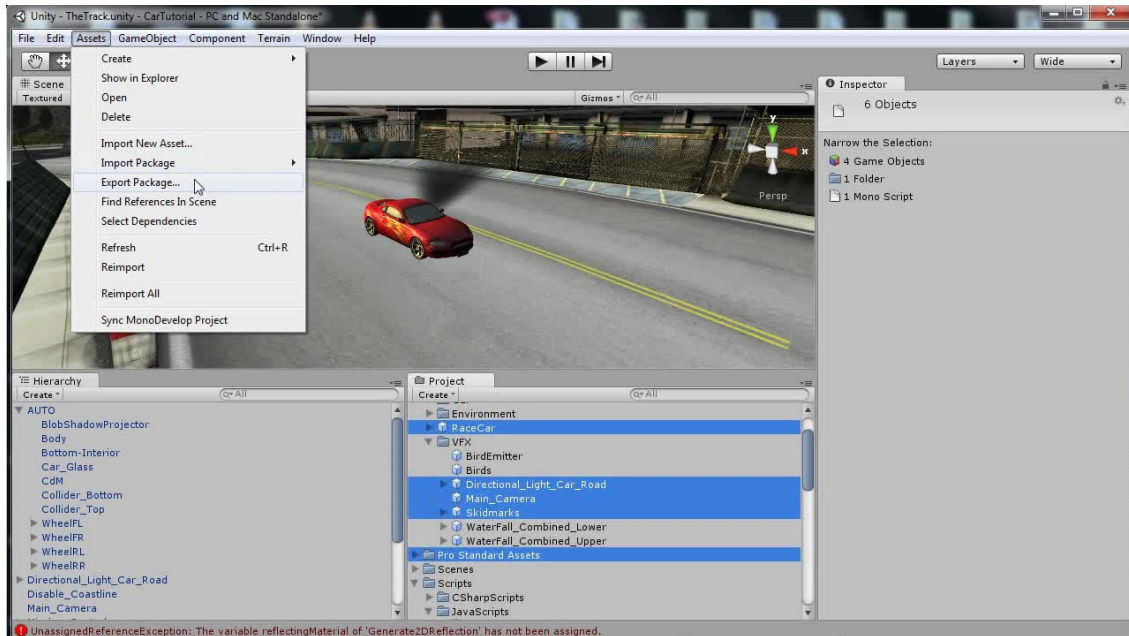


Dobbiamo **includere anche certi script nel Package**, come *SoundToggler.js*, *ImageEffects.cs* e *ImageEffectsBase.cs*. Questi script non sono presenti nella scena, ma vengono richiamati e usati da altri elementi durante il gioco, quindi devono comunque far parte del **PROJECT** di gioco (e del **Package** che stiamo realizzando, che li trasporterà nei nuovi **Projects**).

Gli **script delle immagini** sono inclusi nel pacchetto **Pro Standard Assets**.



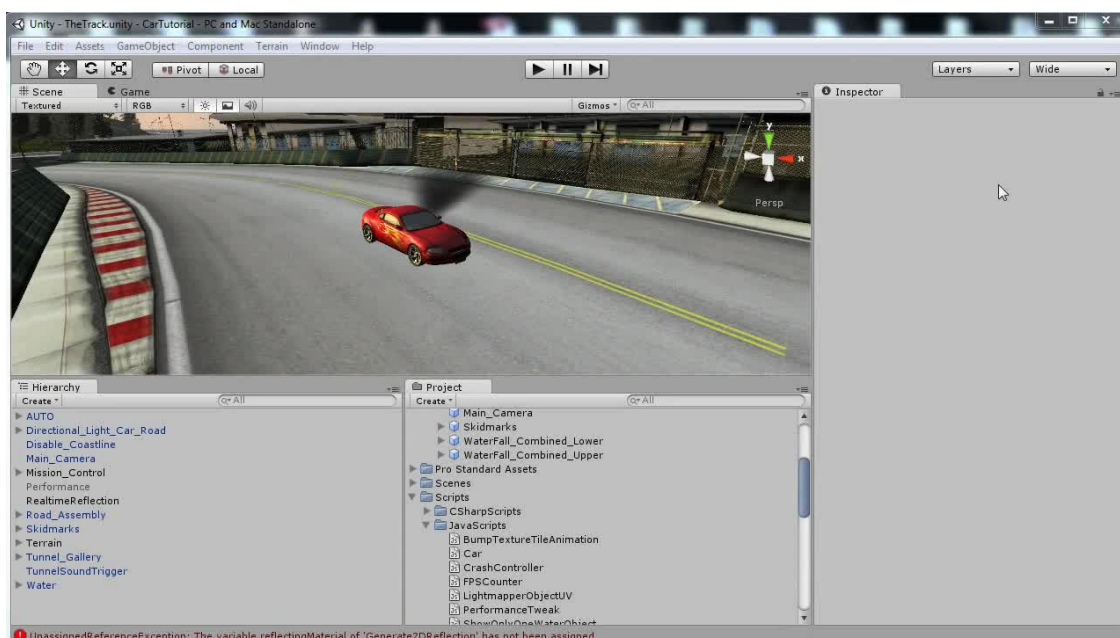
Con tutti gli elementi (Prefabs, Scripts, eccetera) fin qui citati selezionati nella scheda Project, quindi, clicchiamo su **Export Package**, selezioniamo **tutte le dipendenze ed esportiamo**, in modo da avere il **pacchetto su disco** pronto per essere distribuito ed essere riutilizzato in altri progetti.



La fase inversa, quella di **Import**, non la trattiamo, anche perché l'ho mostrata in uno dei dieci capitoli introduttivi su **Unity 3D**.

Il Car Tutorial – Parte 3

Questo terzo (e ultimo) tutorial sul “*Car Project*” di **Unity 3D** dà per scontate delle buone basi di programmazione e anche di **concetti “fisici”** su Rigid Body, movimenti, eccetera, quindi prendetevi tutto il tempo che vi serve per studiarlo, anche “a pezzi”, perché è meglio esaminarlo in più riprese.

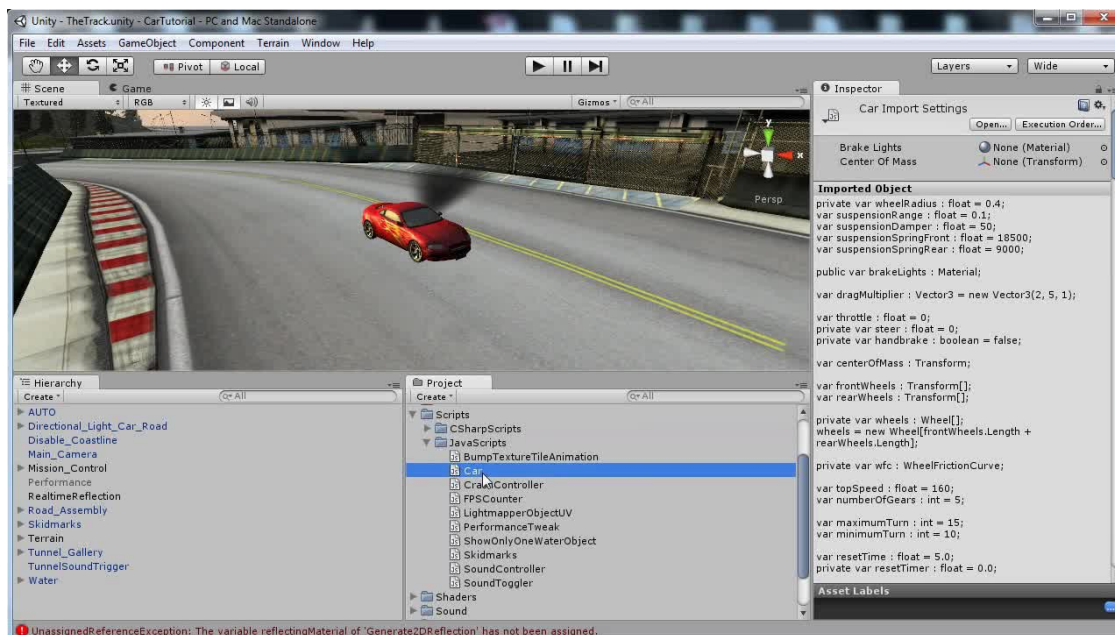


Come anticipato, questi tre tutorial non spiegano come fare un gioco da zero e non vanno nemmeno troppo in dettaglio, ma se non altro sono molto utili **per vedere come altri programmatori, esperti**, hanno affrontato il **setup di un gioco di auto**, quali scelte hanno fatto nel definire gli oggetti, i Components, le funzioni negli script, le variabili, le istruzioni e gli effetti (un esempio su tutti: come fare il testacoda se si usa il freno a mano in corsa, come vedremo in questo tutorial).

Abbiamo visto come realizzare **un'automobile funzionante** partendo da un **modello 3D**, qualche **script** e dei **Components**.

Abbiamo inoltre esaminato le variabili disponibili e come usarle per modificare aspetto e comportamento dell'auto; adesso, dobbiamo dare un'occhiata approfondita “sotto il cofano”, al “motore” dell'oggetto: lo **script Car**.

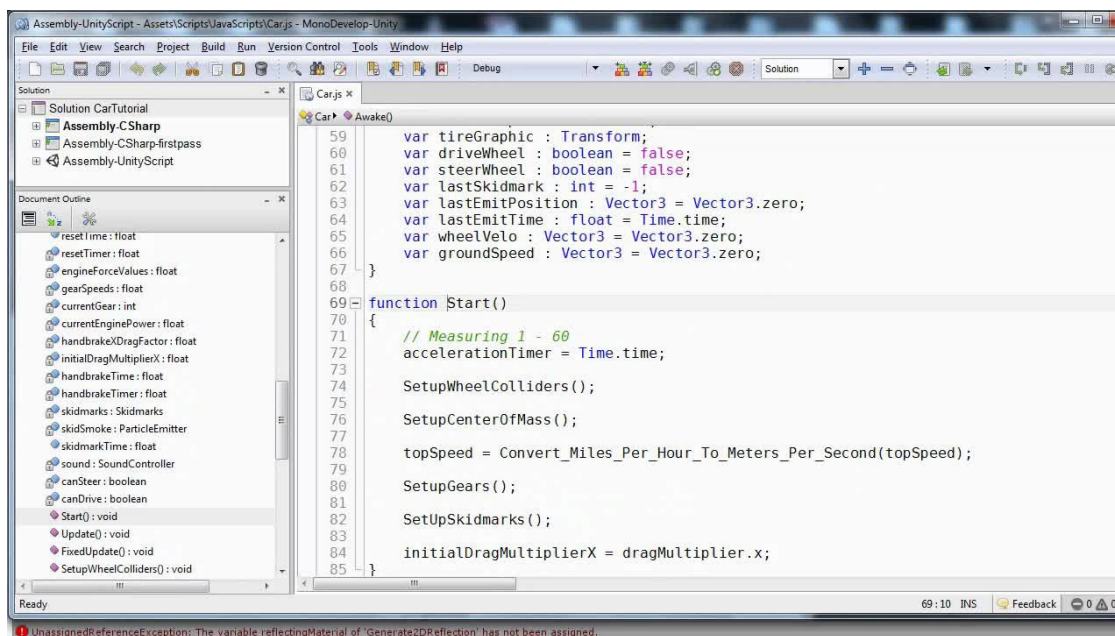
Apriamo **Car.js** in **MonoDevelop**.



Lo script presenta più di 500 linee, tra codice e commenti, ma non disperiamo: tutto è strutturato in modo tale da presentare poche funzioni dai nomi abbastanza significativi, che fanno esattamente quel che dicono di fare; per questo motivo, il codice non è commentato passo per passo, visto che è il codice stesso a dirci cosa fa.

In questi casi (quando si ha a che fare con script ricchi di funzioni, prodotti da terzi, da analizzare e personalizzare) si suggerisce di trovare un buon punto d'inizio e procedere da lì. In questo caso, inizieremo dalle **funzioni Start, Update e FixedUpdate**.

Ciascuna di queste funzioni richiama altre funzioni; ad esempio, la prima funzione chiamata da **Start** è **SetupWheelColliders**. L'idea quindi è di **seguire lo scorrimento** di **Start, Update e FixedUpdate**, esaminando di volta in volta le funzioni chiamate per vedere qual è il **flusso di operazioni all'avvio** (con **Start**) e, **poi, ad ogni frame del gioco**: seguendo questa traccia, avremo un'idea di quello che avviene nel codice, cosa “fa muovere” l'auto e come.



QUALI SONO LE COSE PIU' IMPORTANTI?

Lavorare in **Unity** è facile per molti motivi, ad esempio per via di elementi come l'editor, il sistema drag-n-drop e i componenti forniti di serie. L'automobile è già pronta per metà: **Unity** si occupa di importare il modello e ci bastano pochi click per impostare collisioni, rendering e simulazioni fisiche. Nello script, il lavoro riguarda la manipolazione di queste componenti.

Ci sono **alcuni calcoli da fare** per capire cosa succede all'auto. Questa è una fase dello sviluppo dei videogiochi che non può essere evitata: realizzare degli script e la logica del gioco per gli effetti e i comportamenti che si desidera implementare; comunque, qui si tratta solo di calcoli da fare per passare i valori corretti ai nostri Components.

Se pensate che ci sia troppo codice e non sapete da dove iniziare, ecco una lista di **elementi sui quali concentrarsi**, cercando di capire quali sono le voci che li riguardano e in che modo: il **Rigid Body**, i **Wheel Colliders**, i calcoli che facciamo e l'ordine col quale li facciamo.

Il punto è questo: aggiungendo un **Rigid Body**, possiamo controllare il modello dell'auto in maniera “fisica”, calcolandone le forze che contribuiscono ad accelerare o frenare l'auto; aggiungendo i **Wheel Colliders**, controlliamo il contatto dell'auto con la strada.

IL METODO START: SETUP DELL'AUTO

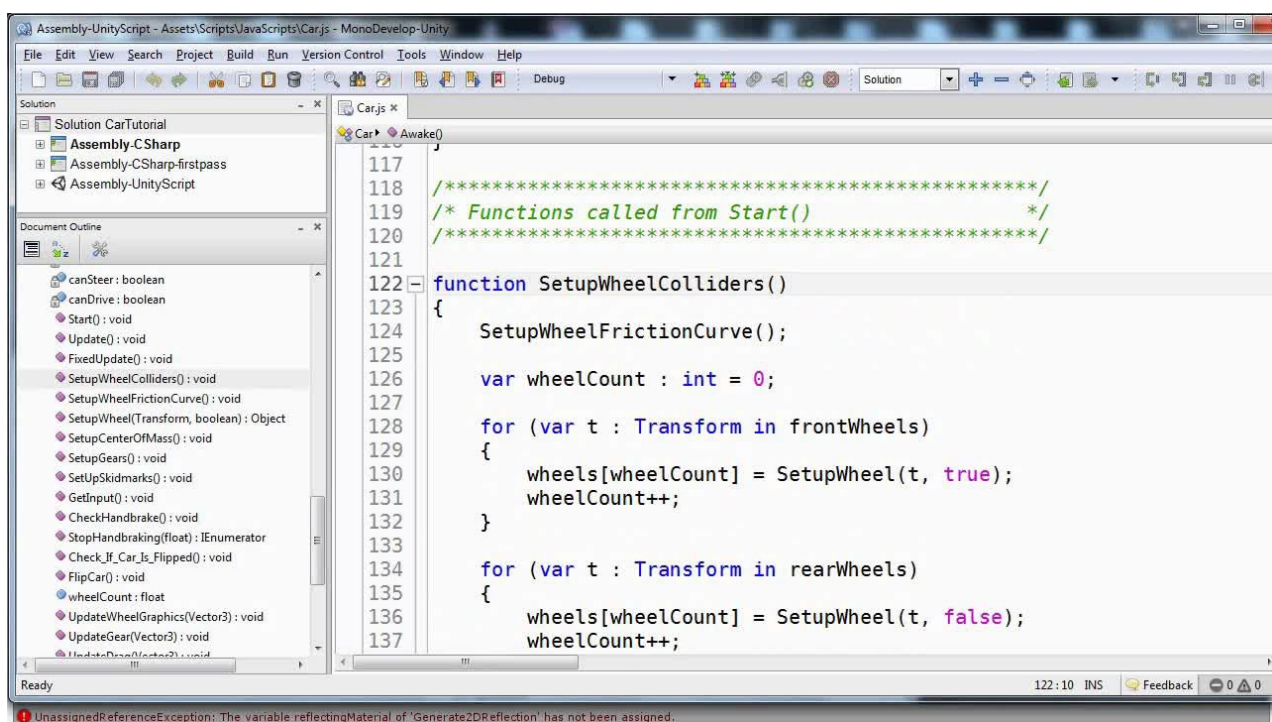
Nel metodo *Start*, come suggerisce il nome, facciamo le inizializzazioni necessarie per l'*AUTO*.

La funzione *Start* viene eseguita **automaticamente, ma solo una volta, all'avvio dello script**, prima della funzione *Update*, per cui viene usata per le (pre)inizializzazioni di variabili e parametri.

La prima funzione richiamata in *Start* che analizzeremo è *SetupWheelColliders*.

SETUP WHEEL COLLIDERS

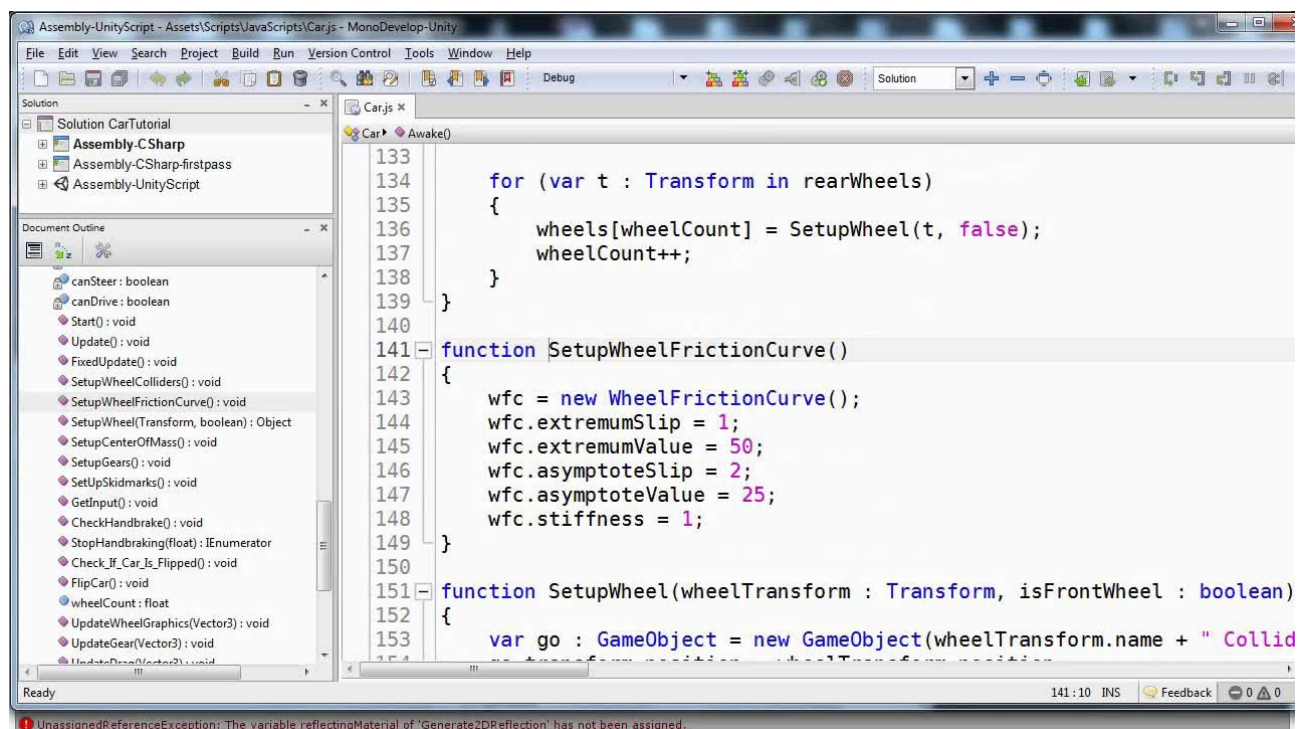
Come sappiamo, abbiamo quattro ruote agganciate all'auto e le abbiamo associate agli array *Front Wheels* e *Rear Wheels* nell'Inspector. In questa funzione, **creiamo i Colliders veri e propri**, facendo interagire le ruote con la superficie sottostante.



Dobbiamo quindi esaminare la funzione *SetupWheelFrictionCurve*.

SETUP WHEEL FRICTION CURVE

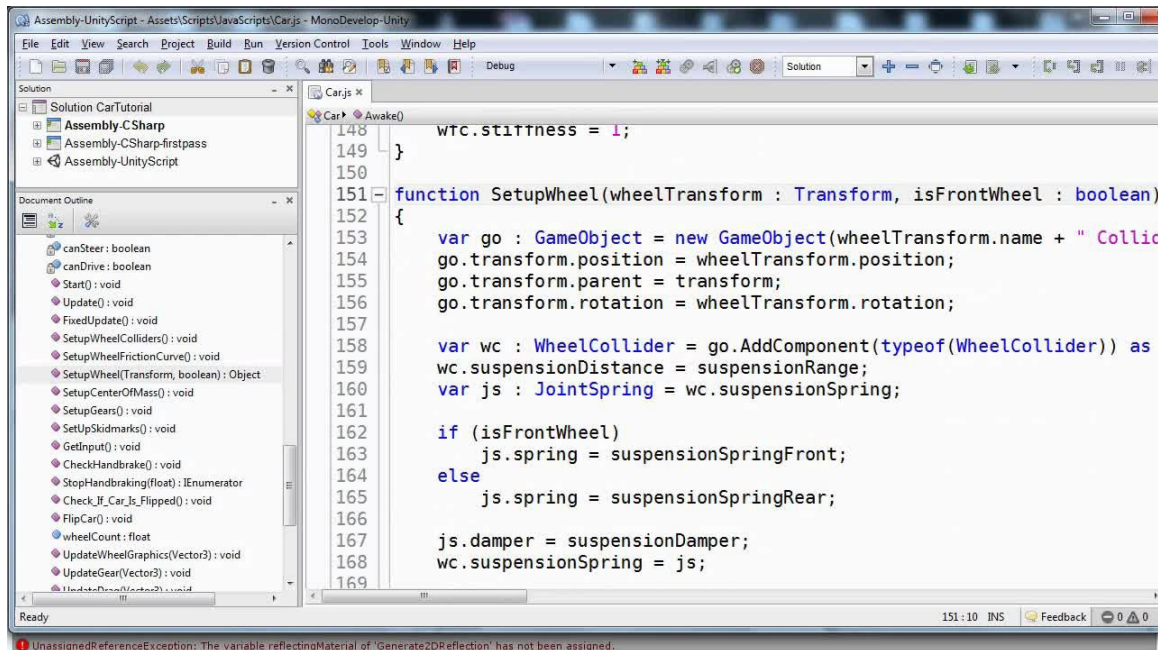
Qui creiamo semplicemente un nuovo *WheelFrictionCurve* e gli assegniamo i valori che riteniamo buoni per la nostra auto.



Questo oggetto *WFC* viene usato dai *WheelColliders* per descrivere le **proprietà di attrito della ruota**. Per approfondire davvero il discorso dei *WheelColliders* forniti di serie da **Unity**, vi rimando alla relativa documentazione; per la trattazione di base, su questo argomento ci fermiamo qui e passiamo a *SetupWheel*.

SETUP WHEEL

Dopo aver impostato il *WFC*, siamo di nuovo in *SetupWheelColliders*, pronti per impostare davvero i *Colliders* per le ruote. Questo avviene chiamando la funzione *SetupWheels* (riportata parzialmente nell'immagine nella pagina seguente; gli screenshot riportati in questo capitolo, comunque, servono più da “riferimento visivo”: si consiglia, invece, di aprire lo script ed esaminare lo stesso in Mono Develop, mentre si legge questo capitolo) per ciascuna delle ruote.



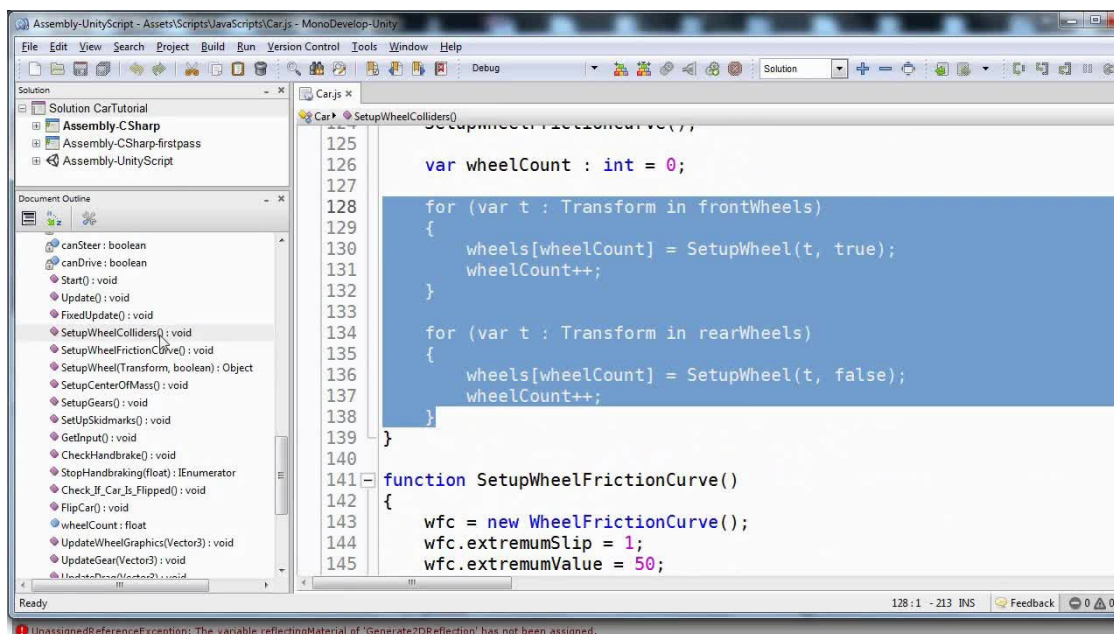
Tale **funzione** prende due parametri: un **Transform** e un **Boolean**, restituendo alla fine un oggetto **Wheel**.

I parametri sono: il **Transform** della ruota (quindi tutte le informazioni su posizione ecc.) e un **Boolean** che è **True** se si tratta di una ruota anteriore, **False** altrimenti.

L'oggetto **Wheel** restituito dalla funzione viene messo nel nostro **array di ruote (Wheels)**, che manterrà i riferimenti alle quattro ruote per tutto il resto dello script:

```
for(var t: Transform in frontWheels) {  
    wheels[wheelCount] = SetupWheel(t, true);  
    wheelCount++;  
}
```

Ecco cosa avviene invece dentro **SetupWheel**: creiamo un nuovo **Game Object** e gli diamo posizione e orientamento della ruota che stiamo elaborando, quindi aggiungiamo un componente **WheelCollider** al Game Object creato.



Impostiamo le proprietà del **WheelCollider** usando le variabili delle sospensioni che abbiamo trattato quando abbiamo personalizzato l'AUTO, ossia **Range**, **Spring** e **Damper**.

A questo punto creiamo un **oggetto Wheel** e gli diamo le proprietà di cui ha bisogno: il **Collider** creato, l'oggetto **WheelFrictionCurve**, l'aspetto visivo delle ruote (l'oggetto **DiscBrake**, impostato via drag and drop nell'Inspector, in precedenza) e l'aspetto visivo dei copertoni (oggetto figlio di **DiscBrake**).

Impostiamo il raggio delle ruote automaticamente, in base alle dimensioni del copertone:

```
wheel.collider.radius = wheel.tireGraphics.render.bounds.size.y / 2;
```

Infine, **verifichiamo se la ruota è anteriore o posteriore** in base al secondo parametro, booleano, passato alla funzione: se si tratta di una ruota posteriore, mettiamo a **True** la variabile **DriveWheel**, ruota motrice, mentre se anteriore mettiamo a **True** la variabile **SteerWheel**, ruota per la sterzata / cambio di direzione.

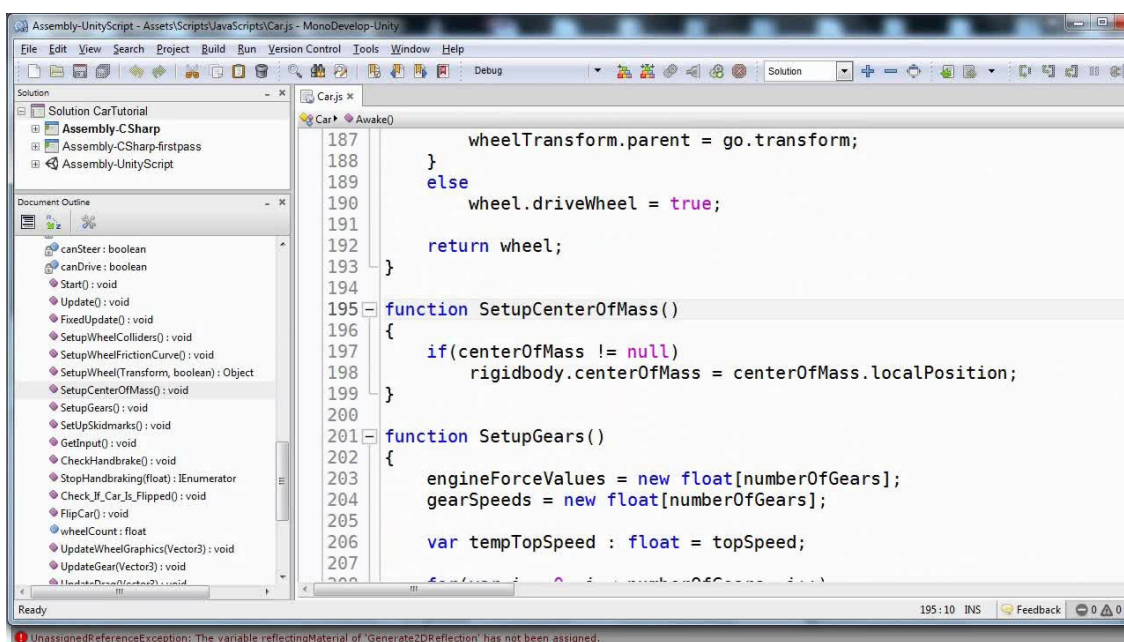
Più in avanti, nel codice, **metteremo dei controlli** per far sì che l'auto possa accelerare solo se almeno una ruota posteriore è a contatto col terreno e sterzare solo se lo è almeno una ruota anteriore.

Aggiungeremo poi, tra ciascuna ruota anteriore e il telaio, un Game Object supplementare, lo **Steer Column**, che useremo **per ruotare la ruota** in fase di sterzata.

Al termine, **la funzione restituisce la ruota così definita**, che viene inserita nell'array delle ruote nella funzione **SetupWheelColliders**; quando abbiamo finito di processare tutte le ruote, la funzione termina e si torna a *Start*, riprendendo il processo di inizializzazione con la funzione successiva: **SetupCenterOfMass**.

SETUP CENTER OF MASS

La funzione **SetupCenterOfMass** imposta il **centro di massa del Rigid Body** usando il Game Object **CdM**, *Centro di Massa*, creato in precedenza.



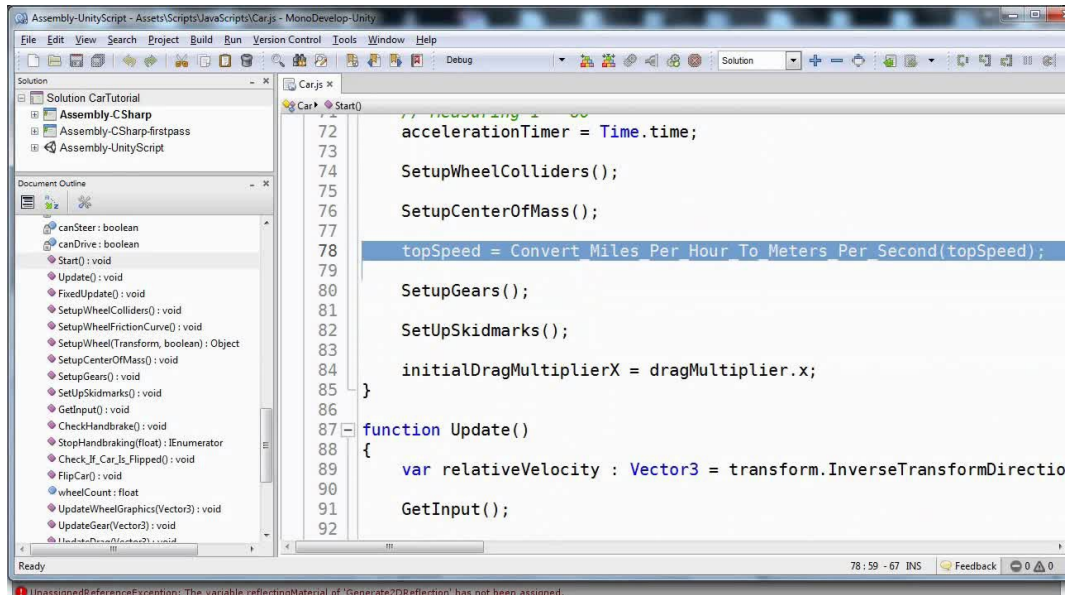
Se questo non è stato definito o impostato, il Rigid Body userà quello calcolato di default da **Unity**.

A questo punto convertiremo il **parametro Top Speed** inserito nell'Inspector mediante una piccola funzione:

```
topSpeed = Convert_Miles_Per_Hour_To_Meters_Per_Second(topSpeed);
```

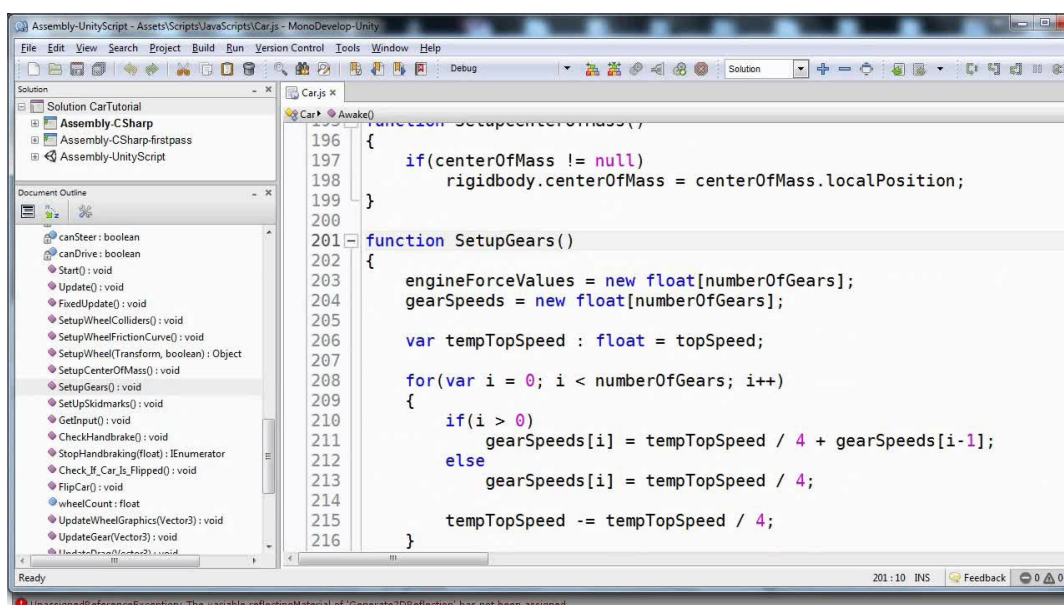
La funzione moltiplica il valore della variabile *topSpeed* per 0.44704, per convertirlo in m/s, il che serve a farci inserire la **velocità in mph nell'Inspector** e fare poi i calcoli nel sistema MKS.

C'è anche una funzione che fa il calcolo inverso, utile se vogliamo mostrare la velocità in mph.



SETUP GEARS

Nella funzione **Setup Gears** (letteralmente: “imposta rapporti” o “imposta marce”), a ciascuna ruota viene assegnato **un valore di velocità massima** (top speed) e si calcola **quanta forza serve** per portare l'auto a quella velocità. Il calcolo avviene usando i parametri **Friction** e **Drag** (variabili pubbliche), quindi in pratica una versione lineare sull'asse Z del calcolo della *Friction* nelle funzioni di *Update*.



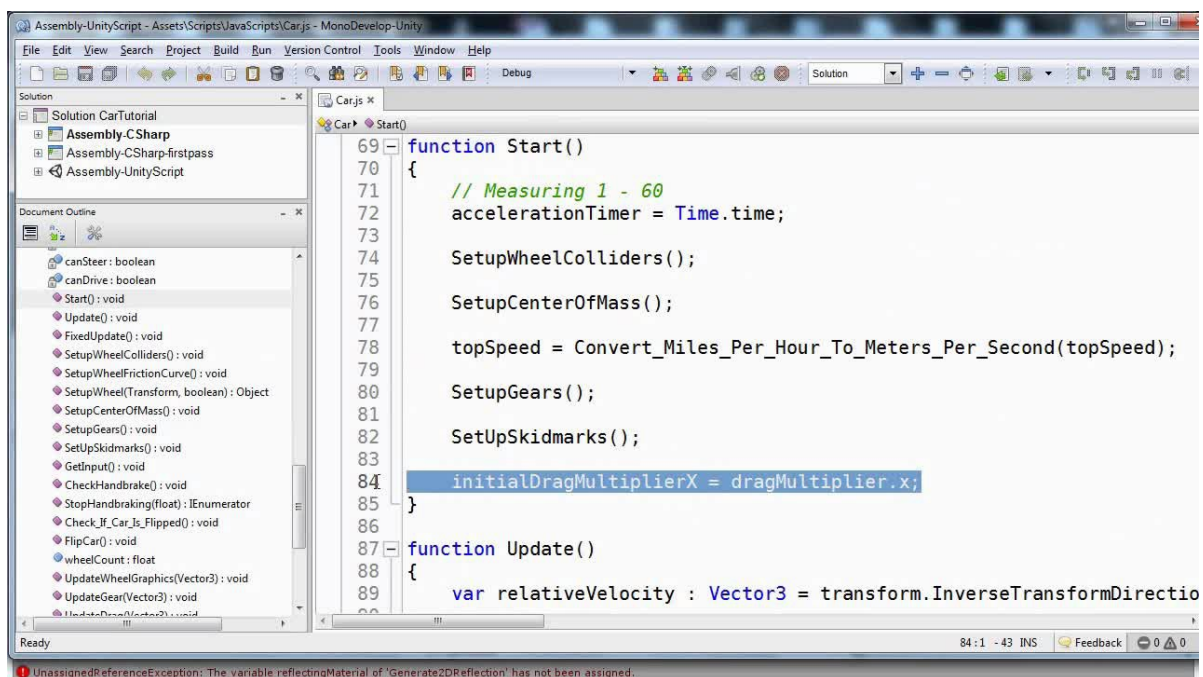
SETUP SKIDMARKS

La funzione *SetupSkidmarks* trova il **Game Object Skidmark** nella scena e memorizza un riferimento ad esso e al suo *Particle Emitter*, usato per il fumo.

In questo tutorial non esamineremo in dettaglio il codice usato per gli *Skidmarks*, potete approfondire per conto vostro, se desiderate.

Alla fine di *Start* assegniamo il **valore x** della variabile array/vettore *dragMultiplier* a una variabile:

```
initialDragMultiplierX = dragMultiplier.x;
```



La memorizziamo perché, quando si usa il **freno a mano**, la **componente x di dragMultiplier** viene modificata, quindi ci serve mantenere il valore originale da qualche parte per ripristinarlo quando non usiamo il freno a mano.

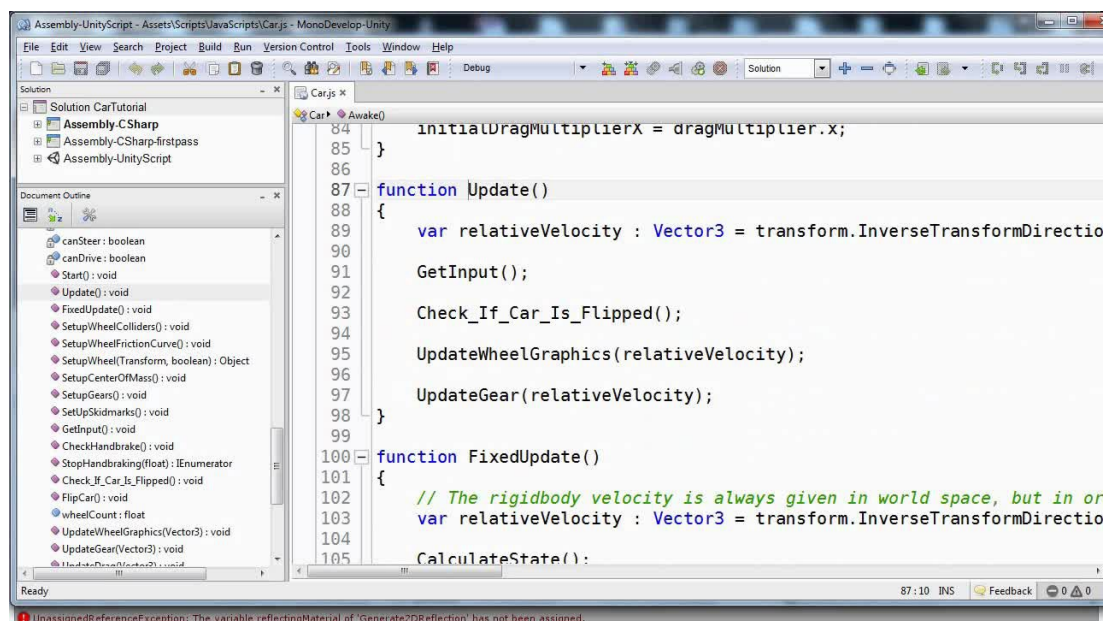
Per quanto riguarda le funzioni del blocco *Start*, è tutto.

Passiamo quindi al ciclo di esecuzione principale, richiamato **frame per frame**, con le funzioni del gruppo *Update*.

UPDATE

La funzione Update viene chiamata automaticamente da Unity ad ogni frame del gioco, una volta che lo script è stato inserito nella scena e l'esecuzione di Start è terminata.

Nel nostro caso, quindi, ad ogni frame viene (ri)creata la variabile **relativeVelocity** e vengono invocate quattro funzioni: **GetInput**, **Check_If_Car_Is_Flipped**, **UpdateWheelGraphics** e **UpdateGear**.



GET INPUT

La prima cosa da fare in **Update** è rilevare l'**input del giocatore**.

Le prime due linee di **Get Input** rilevano gli **assi orizzontali e verticali** e salvano i valori rispettivamente nelle variabili **steer** (sterzo) e **throttle** (acceleratore):

```
throttle = Input.GetAxis("Vertical");
steer = Input.GetAxis("Horizontal");
```

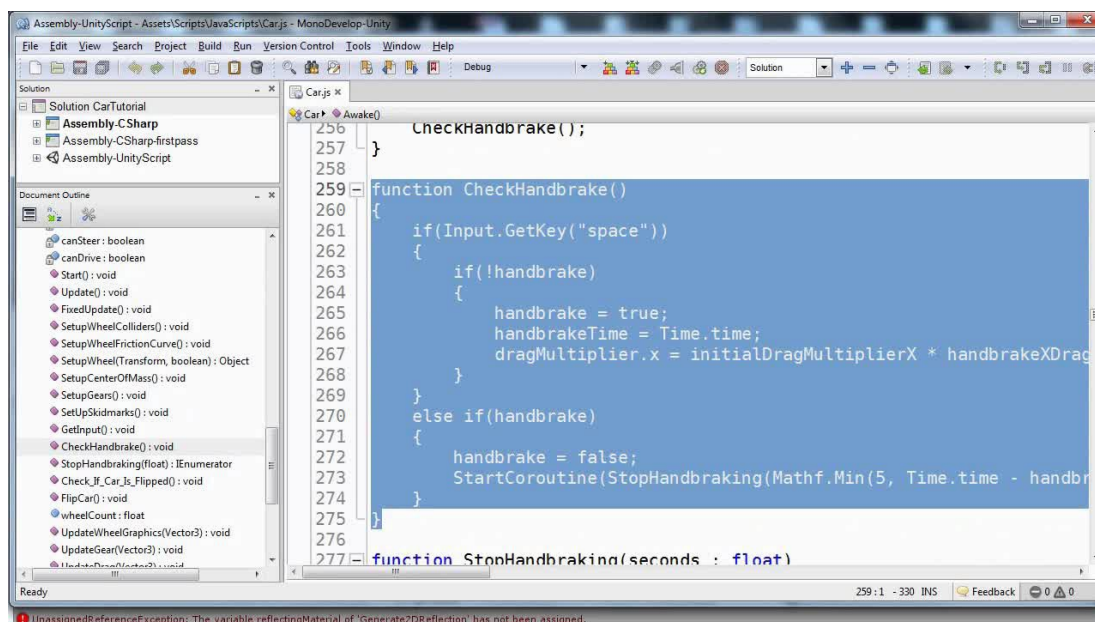
Questi “assi” vanno impostati in **Edit – Project Settings – Input** (ossia, nell'Input Manager di **Unity**); di default, l'asse verticale è associato ai **tasti “W” e “freccia in su”** per il verso positivo e **“S” e “freccia in giù”** per quello negativo, con il valore letto che viene utilizzato, dopo, per calcolare la forza dell'accelerazione. L'asse orizzontale è associato a **“A” e “freccia a sinistra”** in un verso e a **“D” e “freccia a destra”** nell'altro, per determinare la sterzata.

CHECK HANDBRAKE

La funzione **CheckHandbrake** verifica se la barra spazio (corrispondente, nello schema dei controlli, al freno a mano) è premuta e, se sì, esegue delle operazioni:

- quando premiamo la barra spazio, impostiamo la variabile **handbrake** a **True**, avviamo un timer per il freno a mano e cambiamo il valore di **dragMultiplier.x** (cosa che renderà l'auto più difficile da gestire, soggetta a sbandate e scossoni);
- quando la barra spazio non risulta premuta, viene eseguito il codice nel blocco else, ma solo se la variabile **handbrake** è a **True**; ciò significa che viene eseguito solo quando l'utente rilascia la barra spazio, visto che, nel blocco, mettiamo la variabile **handbrake** a **False**; a questo punto, parte la **coroutine StopHandbraking**:

```
StartCoroutine(StopHandbraking(Mathf.Min(5, Time.time -  
handbrakeTime)));
```



STOP HANDBRAKING

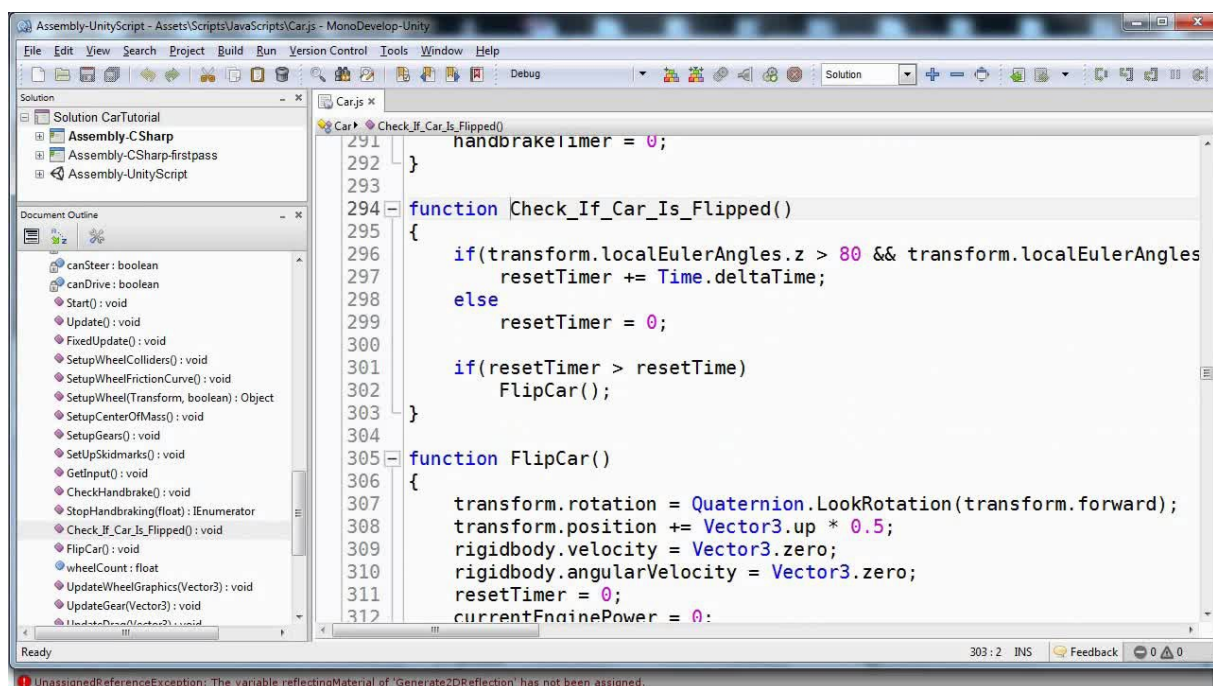
La funzione **StopHandbraking** prende in input una variabile che definisce quanti secondi impiegare per riportare il valore di **dragMultiplier.x** al valore originale. Il valore dato è il minimo tra 5 e il valore del timer del freno a mano attivato quando si è “tirato” il freno a mano (al massimo, 5).

La funzione implementa quindi questo **ritardo**, trascorso il quale **dragMultiplier.x** torna al valore specificato nella parte finale di **Start** e la macchina torna a essere guidata normalmente, **senza “sbandate”**.

CHECK IF CAR IS FLIPPED

La **funzione** *Check_if_car_is_flipped*, che si trova in *Update* (e quindi viene chiamata a ogni frame), fa il... controllo della tartaruga: verifica se l'auto è in orientamenti che non consentono di guidarla (ad esempio, se è **completamente capovolta**) e, se sì, aggiorna una variabile **timer**, altrimenti (cioè, se l'angolo va bene), il **timer** viene riportato a 0.

Se il valore di questa variabile timer supera quello definito nella variabile **resetTimer** (default: 5 secondi), viene invocata la **funzione** *Flip Car*.

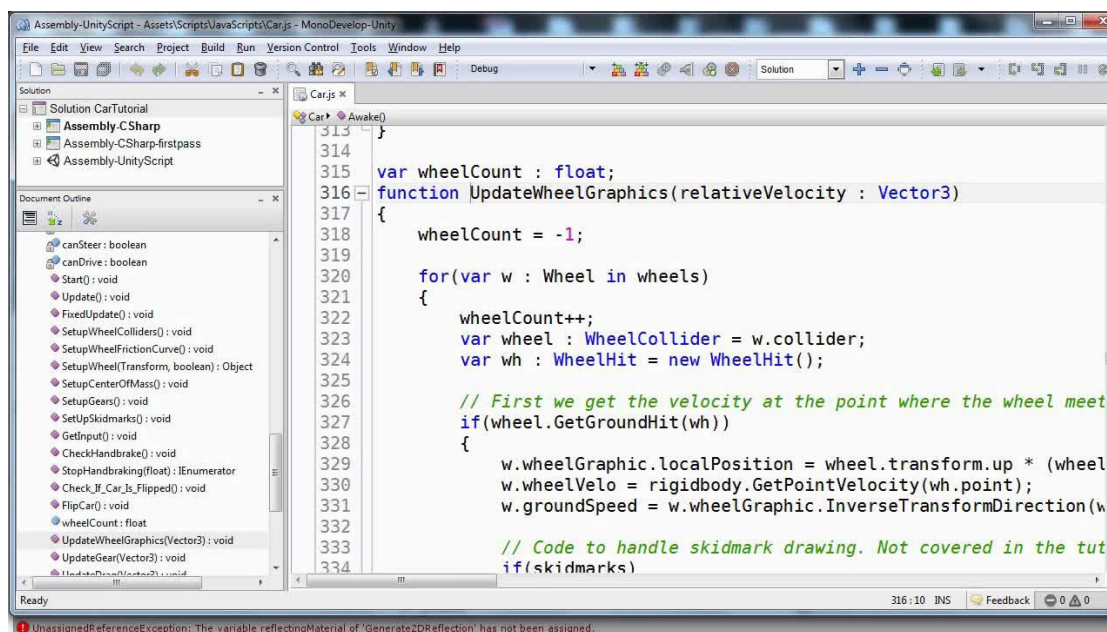


FLIP CAR

La funzione **FlipCar** riporta l'auto sulle quattro ruote, cioè col giusto orientamento; inoltre, **riporta a 0 la velocità**, per ripartire da fermi.

UPDATE WHEEL GRAPHICS

UpdateWheelGraphics è la funzione più lunga e complessa, tra quelle richiamate da Update, ma la parte centrale ha a che fare con le *Skidmarks* e non la tratteremo; la parte importante in questa funzione, invece, riguarda **posizione e orientamento (rotazione) delle ruote**.



Per ogni ruota, iniziamo controllando se è a **contatto col terreno**; se sì, impostiamo l'immagine (la “grafica della ruota”) nel punto ove tocca il suolo, ma in avanti di una misura data dal raggio della ruota: questo posizionerà correttamente il centro della ruota in relazione al telaio:

```
w.wheelGraphic.localPosition = wheel.transform.up * (wheelRadius +  
wheel.transform.InverseTransformPoint(wh.point).y);
```

Dopo aver posizionato la ruota, **recuperiamo la velocità del Rigid Body** nel punto di contatto col suolo, la traduciamo **in coordinate locali della ruota** e la memorizziamo in tale oggetto:

```
w.wheelVelo = rigidbody.GetPointVelocity(wh.point);  
w.groundSpeed = w.wheelGraphic.InverseTransformDirection(w.wheelVelo);
```

Se la ruota in questione **non è a contatto col terreno**, la posizioniamo in base alla posizione dell'oggetto genitore e la dimensione delle sospensioni.

A questo punto, gestiamo **l'orientamento**.

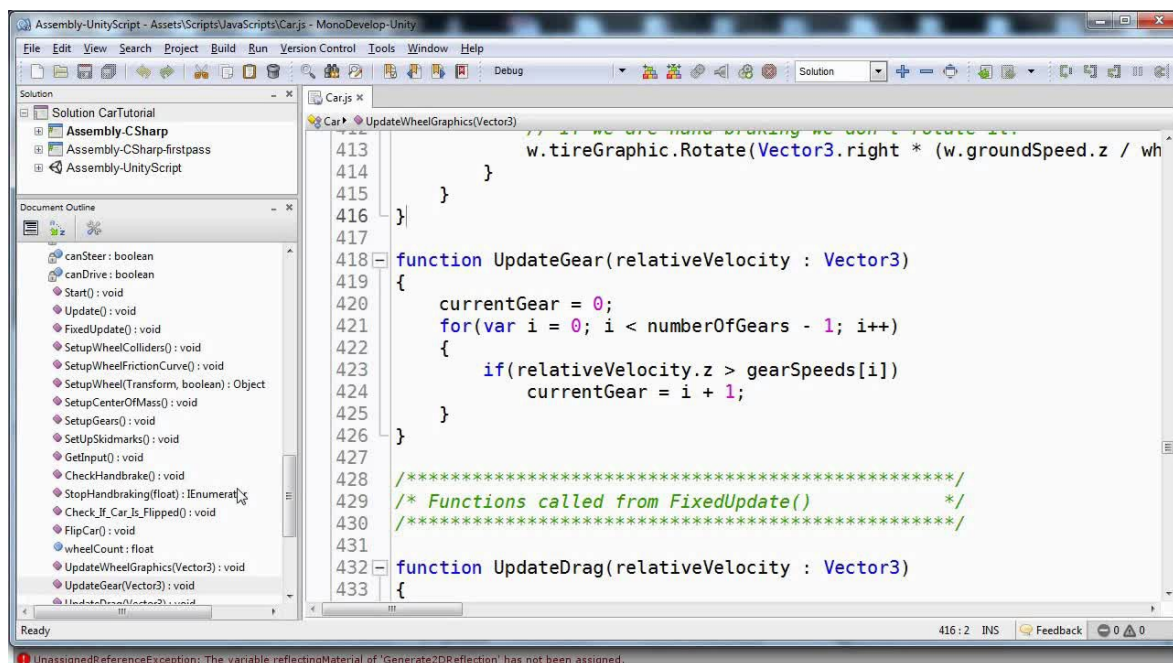
Se è una ruota anteriore e stiamo sterzando, applichiamo la sterzata ruotando, in realtà, l'oggetto *Steer Column*, che è il padre della *Wheel Graphics* e che abbiamo creato precedentemente, usando come fattore di moltiplicazione il prodotto tra la variabile *steering* (basata sull'input dell'utente) e l'angolo massimo di sterzata da noi definito.

Per tutte le ruote, poi, applichiamo **una rotazione che rappresenta la velocità**, ruotando la ruota in avanti **in base a velocità e raggio**:

```
w.tireGraphic.Rotate(Vector3.right * (w.groundSpeed.z / wheelRadius) *  
Time.deltaTime * Mathf.Rad2Deg);
```

UPDATE GEAR

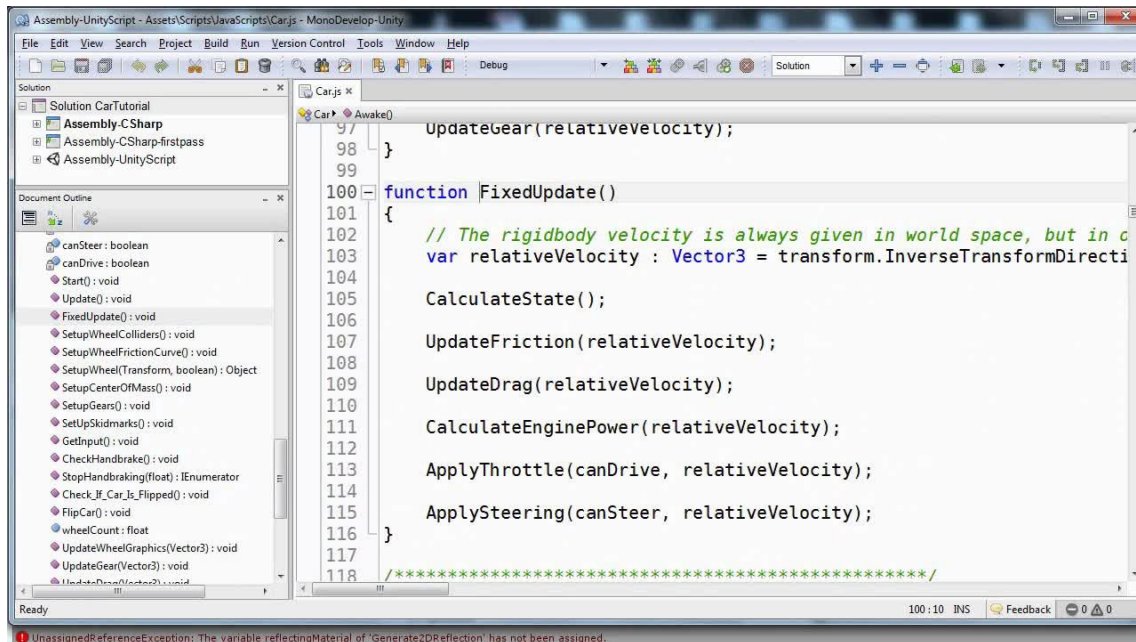
L'ultima funzione chiamata in *Update* è *UpdateGear*, una piccola procedura che **determina la marcia corrente** confrontando la velocità con i valori che abbiamo definito per ciascuna marcia in *SetupGears*, chiamata a sua volta in *Start*.



Quanto detto conclude la panoramica per la funzione *Update*; ora, dobbiamo passare ai **calcoli di fisica** implementati in *FixedUpdate*.

FIXED UPDATE

I calcoli della fisica vanno effettuati in modo tale da **mantenere fluida l'esecuzione del gioco**, per cui è meglio farli a certi **intervalli**: questo è lo scopo per il quale è stata creata **FixedUpdate**.



La documentazione dice che **se il frame rate è basso**, **FixedUpdate** può essere chiamata anche più volte per frame; se però **il frame rate è alto**, può **non essere chiamata** affatto tra un frame e l'altro.

Tutti i calcoli fisici e gli aggiornamenti vengono effettuati subito prima di **FixedUpdate**.

In **FixedUpdate** vengono chiamate ed eseguite **varie funzioni**, tutte riguardanti le **forze** applicate all'auto.

UPDATE DRAG

Per **Drag** intendiamo, qui, la **resistenza all'aria**, quindi una forza **opposta alla direzione** del moto della vettura. Nella funzione **UpdateDrag**, per calcolare questo valore useremo il quadrato della velocità dell'auto:

```
Vector3 ( -relativeVelocity.x * Mathf.Abs(relativeVelocity.x),  
-relativeVelocity.y * Mathf.Abs(relativeVelocity.y),  
-relativeVelocity.z * Mathf.Abs(relativeVelocity.z));
```

il che significa che, **come avviene nel mondo reale**, all'aumentare della velocità la resistenza dell'aria aumenterà ancora di più.

Dopo, **relativeDrag** viene scalato con il già trattato **dragMultiplier**, per tenere conto del fatto che i profili laterale, superiore e frontale dell'auto sono molto diversi tra loro.

Se stiamo usando il freno a mano (“**hand braking**”), applichiamo forze extra ai valori di **Drag** frontale e laterale, in base alla velocità e all'orientamento dell'auto in quel momento; se, invece, non stiamo usando il freno a mano, aggiorniamo solo il valore x come segue:

```
drag.x *= topSpeed / relativeVelocity.magnitude;
```

Alla fine della funzione **applichiamo la forza al Rigid Body** con:

```
rigidbody.AddForce(transform.TransformDirection(drag) * rigidbody.mass *  
Time.deltaTime);
```

UPDATE FRICTION

La funzione **Update Friction** calcola e applica **l'attrito tra ruote e terreno**.

Si fa uso di **WheelFrictionCurve**, definita in precedenza, che restituisce in output una forza, **separata in due componenti** (e direzioni): l'attrito in avanti (per accelerazione e frenata) e quello laterale (per mantenere l'auto dritta, allineata).

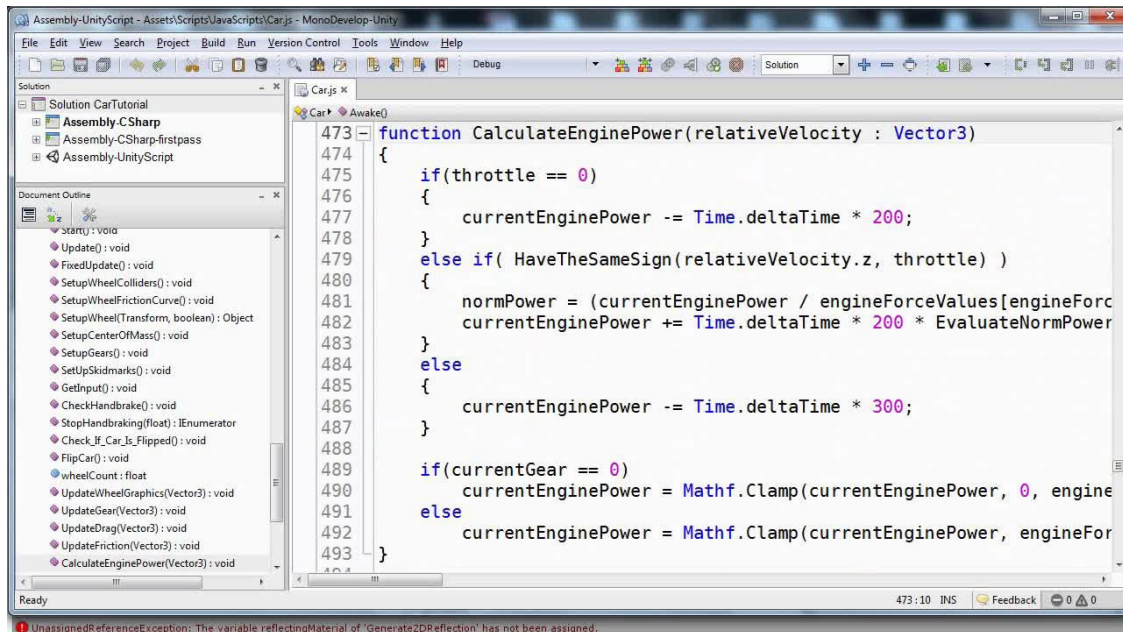
È quindi tramite la **Wheel Friction Curve** che viene aggiornato l'attrito tra ruote e terreno:

```
w.collider.sideways.Friction = wfc;  
w.collider.forwardFriction = wfc;
```

Con la prima istruzione, comunque, cambiamo **la forza di attrito laterale (sideways)**: lo facciamo per evitare che l'auto inizi a **scivolare**, a slittare lateralmente sulla superficie quando ruota (cioè sterza) in curva.

CALCULATE ENGINE POWER

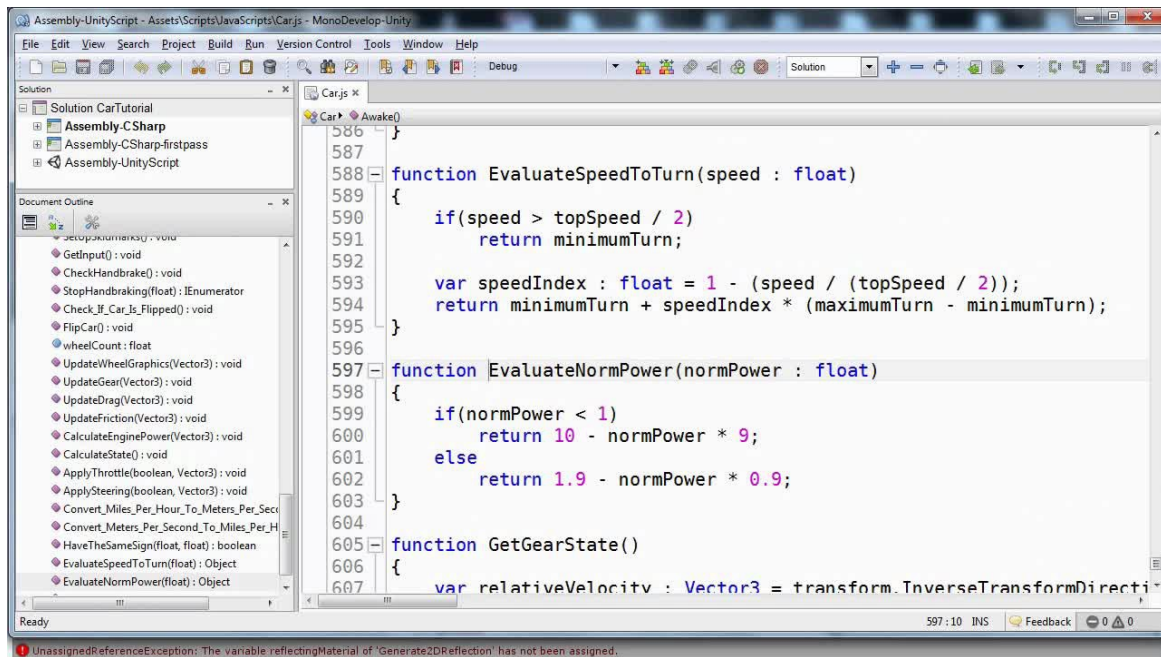
La **funzione** *CalculateEnginePower* serve, come suggerisce il nome, a calcolare la potenza del motore, un dato da usare poi per **applicare una forza al Rigid Body**; si tratta di un calcolo abbastanza lineare, ma ci sono un paio di punti che meritano attenzione.



Se non stiamo accelerando, dobbiamo solo **far diminuire il valore della potenza** del motore nel tempo, facendo rallentare la vettura.

Se stiamo accelerando nella stessa direzione del moto della vettura (condizione, questa, verificata dalla funzione *HaveSameSign*), calcoliamo un valore da **aggiungere alla forza motrice** (engine *force*, non *power*). Quello che segue può sembrare un po' strano: calcoliamo un valore, “*norm power*”, come divisione tra l'attuale potenza del motore e la massima potenza dello stesso (ottenendo un valore normalizzato, tra 0 e 1), quindi moltiplichiamo il risultato per 2, per cui il valore può andare da 0 (potenza minima) a 2 (potenza massima).

A questo punto chiamiamo la **funzione** *EvaluateNormPower* (riportata nell'immagine nella pagina seguente).



Questa funzione restituisce un valore tra 1 e 10 se il valore “**norm power**” dato in input è tra 0 e 1; se questo valore è, invece, tra 1 e 2, la funzione restituisce un valore tra 0 e 1.

Il valore risultante viene usato nell'equazione che aggiunge forza al motore:

```
currentEnginePower += (Time.deltaTime * 200 *
    EvaluateNormPower(normPower));
```

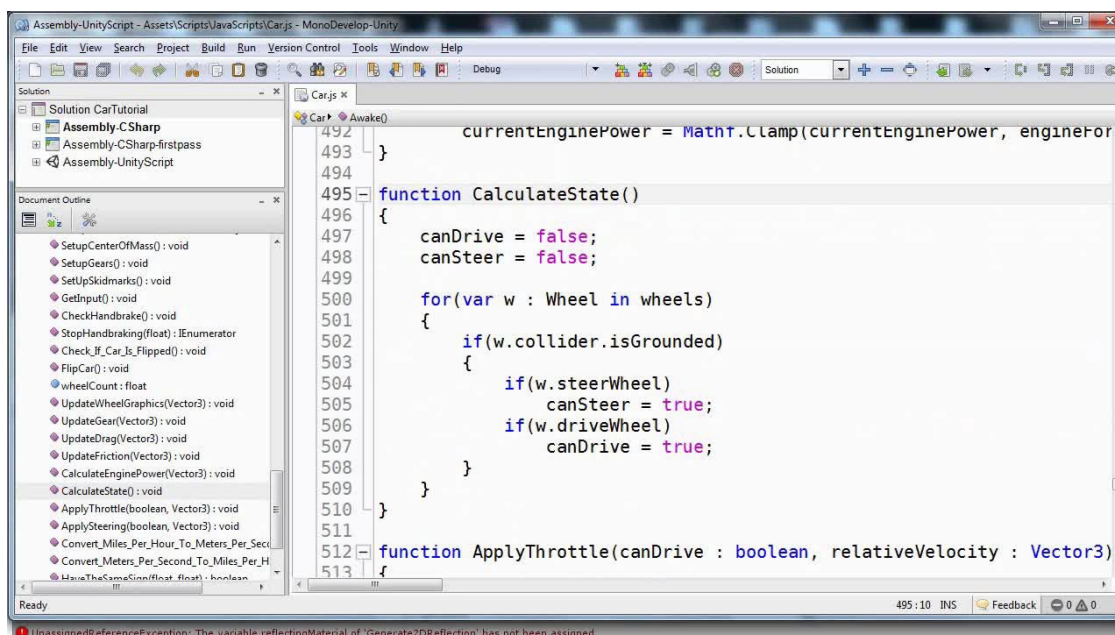
Il risultato di tutto questo è che **aggiungiamo più forza** (e acceleriamo maggiormente) **quando stiamo andando piano**; se, **poi**, siamo alla velocità massima, **non si accelera più**, mantenendo la velocità costante.

Accelerare nella direzione opposta (con la “freccia in giù” o il tasto “S”), invece, equivale a **frenare**.

Alla fine, **la forza motrice calcolata** viene troncata (*clamped*) tra il valore della forza associato alla marcia corrente e quella precedente.

CALCULATE STATE

CalculateState è una piccola funzione che chiamiamo perché dobbiamo sapere **se le ruote** anteriori e posteriori sono a **contatto col terreno** (informazioni che ci servono per le funzioni che chiameremo a breve).



Le operazioni svolte da *CalculateState* sono semplici:

- di default, impostiamo a False le variabili “*canDrive*” e “*canSteer*”;
- controlliamo quindi ciascuna ruota **nell'array Wheels** per vedere se è a contatto col suolo: ***if(w.collider.isGrounded)***.

Se lo è, verifichiamo **di che tipo di ruota** si tratta: se è una ruota posteriore (motrice, *drive wheel*), mettiamo *canDrive* a *True*, altrimenti se è una ruota anteriore (*steer wheel*, per sterzare) mettiamo *canSteer* a *True*.

Quel che segue è che **se almeno una ruota posteriore è a contatto** col suolo possiamo **guidare**; inoltre, se lo è almeno una ruota anteriore possiamo **sterzare**.

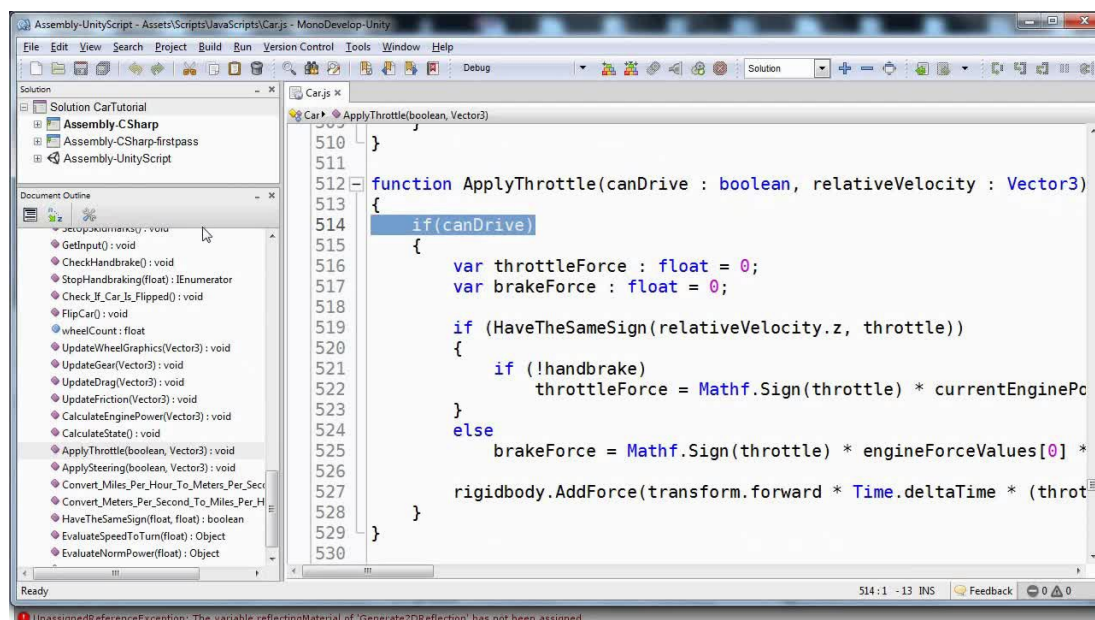
Siamo giunti alle **ultime due funzioni**, che poi sono quelle che in effetti applicano i nostri calcoli al Rigid Body dell'*AUTO*. Qui andremo un po' più in dettaglio, così da capire meglio la logica e i calcoli che, alla fine, ci consentono di guidare e sterzare.

APPLY THROTTLE

Questa funzione viene eseguita solo se *CalculateState* ha settato a *True* la variabile *canDrive*.

Se possiamo guidare, quindi, per prima cosa confrontiamo l'accelerazione data dall'utente mediante il dispositivo (device) di input con il valore della variabile *relativeVelocity.z* (la velocità dell'auto in avanti); se questi due valori hanno lo stesso segno (cosa che si verifica mediante *HaveSameSign*), allora stiamo accelerando nella direzione del moto, quindi bisogna “dare una spinta” (la forza dell'accelerazione) al *Rigid Body*:

```
throttleForce = Mathf.Sign(throttle) * currentEnginePower *  
rigidbody.mass;
```



Se l'accelerazione è negativa (l'utente, quindi, sta premendo il tasto del **freno**), il segno sarà -1, quindi calcoleremo una forza negativa da sommare alla velocità – anch'essa negativa – dell'auto; al contrario, se l'utente accelera si calcola una forza positiva da sommare alla velocità – anch'essa positiva – dell'auto.

Se i due valori sono discordi, stiamo frenando o comunque accelerando in direzione contraria al moto; utilizziamo qui la variabile *brakeForce*, basata sulla massa dell'auto e la forza della prima marcia del motore:

```
brakeForce = Mathf.Sign(throttle) * engineForceValues[0] * rigidbody.mass;
```


(si fa sempre uso del **segno di throttle** perché stavolta, essendo di segno opposto al moto, anche la forza calcolata da sommare algebricamente al moto lo sarà).

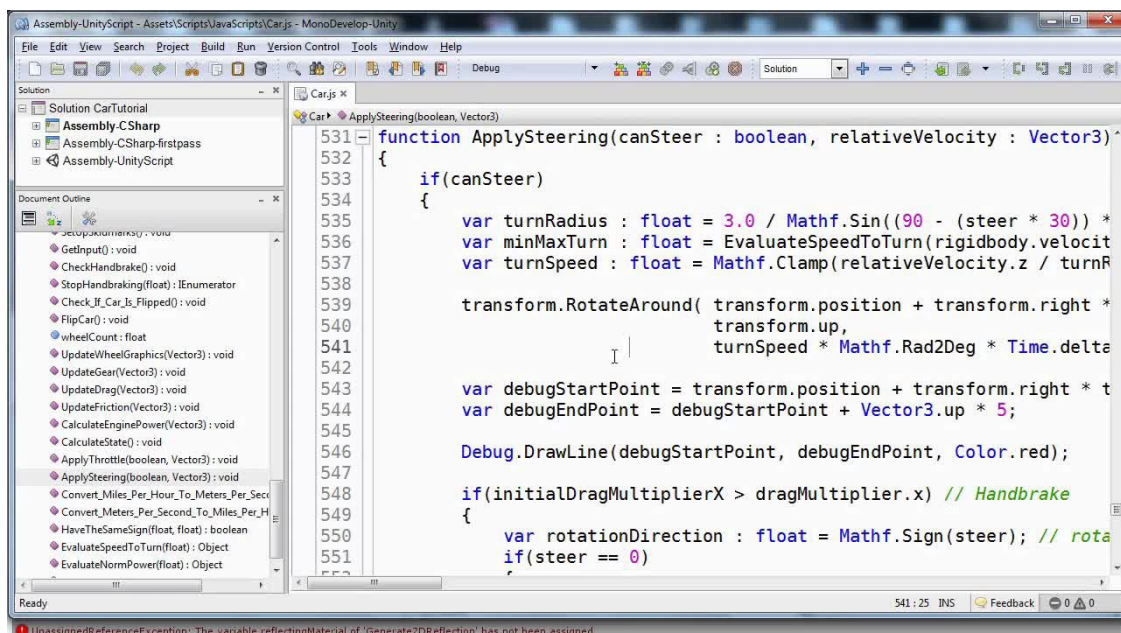
Definita la forza (con il suo segno positivo o negativo), **la si applica alla direzione frontale** (ossia: vettore **forward** relativo) del Rigid Body, con:

```
rigidbody.AddForce(transform.forward * Time.deltaTime * (throttleForce +  
brakeForce));
```

APPLY STEERING

Non viene applicata alcuna **sterzata** se le ruote anteriori non sono a **contatto** col terreno (condizione verificata precedentemente).

All'inizio della funzione **ApplySteering** calcoliamo la variabile **turnRadius**, in base all'input dell'utente.



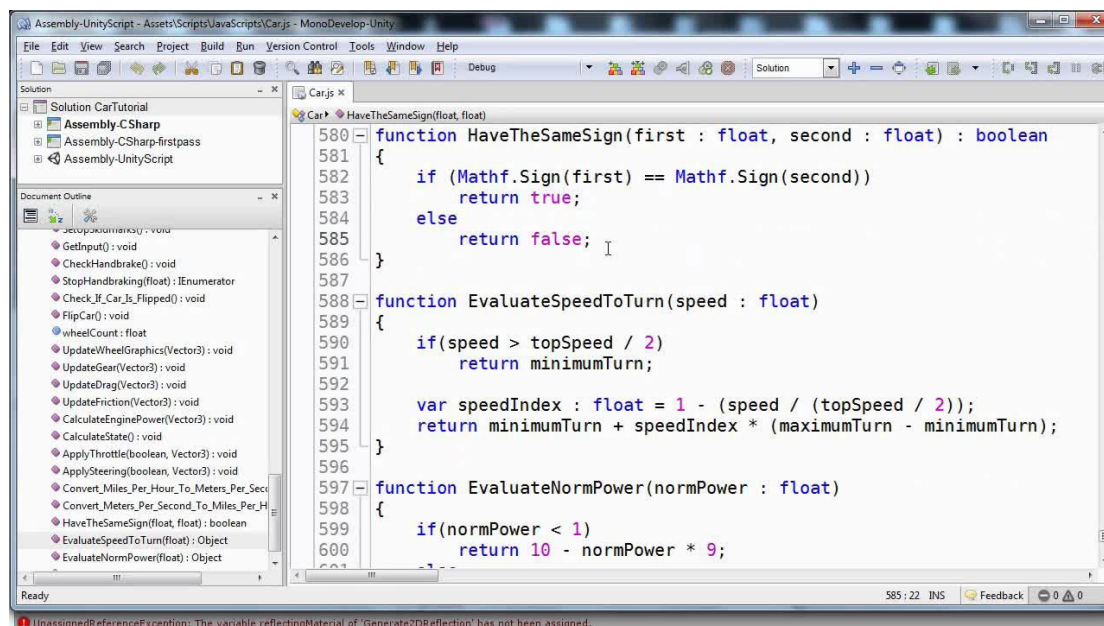
L'equazione fa aumentare **turnRadius** quando si sterza.

Il valore **minMaxTurn** viene calcolato mediante la prossima funzione, **EvaluateSpeedToTurn**.

EVALUATE SPEED TO TURN

Questa funzione restituisce un **valore** calcolato in base alla **velocità della vettura**.

Come spiegato in precedenza, maggiore sarà la velocità, più vicino al **minimumTurn** sarà questo valore, e viceversa.



Tornando in **ApplySteering**, il valore **turnSpeed** che calcoliamo è legato direttamente a **turnRadius** e alla velocità (in avanti) dell'auto; maggiore il raggio, minore l'angolo di rotazione a ciascun frame.

A questo punto ruotiamo l'auto:

```
transform.RotateAround( transform.position + transform.right * turnRadius
    * steer, transform.up, turnSpeed * Mathf.Rad2Deg * Time.deltaTime *
    steer);
```

La funzione **RotateAround**, quindi, ruota un **Transform** intorno a un punto e un asse specificati; prende, come parametro, anche un angolo: l'entità della rotazione.

Quando non stiamo sterzando o curvando, il punto intorno al quale giriamo è il centro esatto dell'auto; **quando premiamo un tasto per sterzare** verso un lato, invece, il punto si sposta dall'auto verso il lato ove vogliamo girare.

La variabile **steer** viene ricavata, lo ricordiamo, dall'input **Horizontal Axis**, che è negativo verso sinistra e positivo verso destra; più giriamo, più aumentano i valori di **turnRadius** e **steer**.

Quando li moltiplichiamo per il vettore **transform.right**, otteniamo un punto basato sul centro dell'auto e spostato sul lato di curvatura (verso il quale sterziamo).

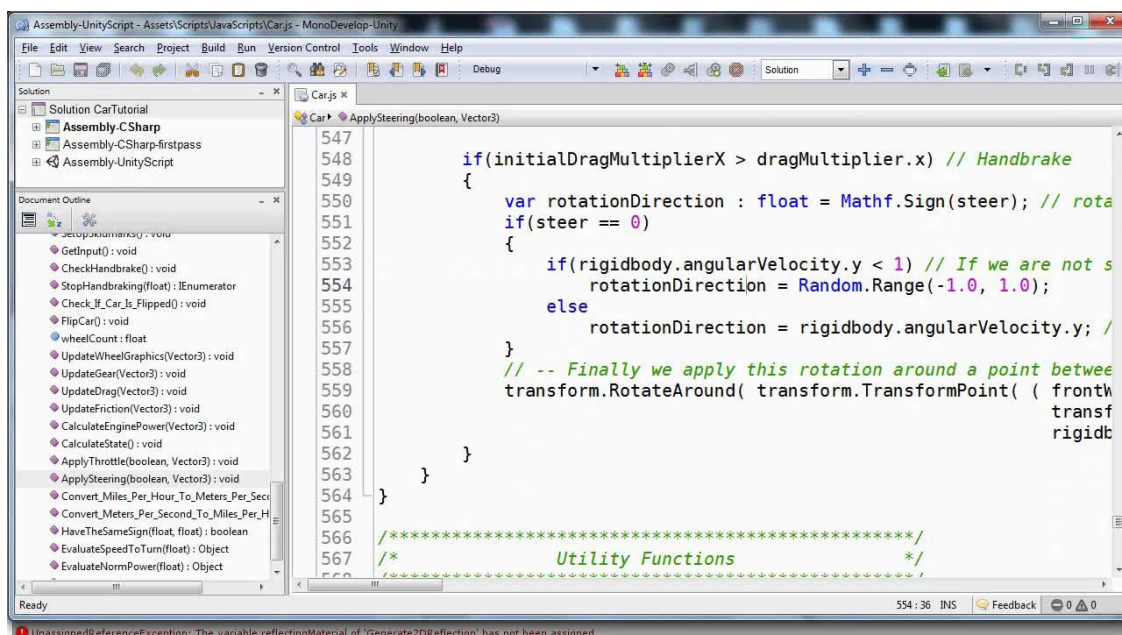
L'asse intorno al quale giriamo è quello **y** (verticale), per cui ruotiamo l'auto sul **piano xz**.

L'angolo viene quindi ricavato dal prodotto tra l'appena calcolato **turnSpeed** e **steer**, per ottenere la **direzione sinistra/destra**.

Guardiamo ora:

```
if(initialDragMultiplierX > dragMultiplier.x)
```

che sappiamo essere **True** quando usiamo il freno a mano; in pratica, verifichiamo **se stiamo sterzando mentre tiriamo il freno a mano**.



Se non stiamo sterzando, verifichiamo se l'auto sta curvando controllando il valore di **angularVelocity.y** del Rigid Body: se questo valore è 0 o comunque è molto basso, non stiamo girando o stiamo curvando molto poco e applichiamo un valore casuale alla direzione di rotazione, per simulare le piccole scosse o sbandate che ha l'auto se si tira il freno a mano mentre si va dritti.

Se il valore non è basso, applichiamo il valore di **angularVelocity.y** alla direzione di rotazione.

Se, in effetti, stiamo sterzando, la **direzione di rotazione** sarà -1 andando verso sinistra e 1 verso destra (in quanto, di default, la direzione di rotazione è data dal segno del valore della variabile *steer*).

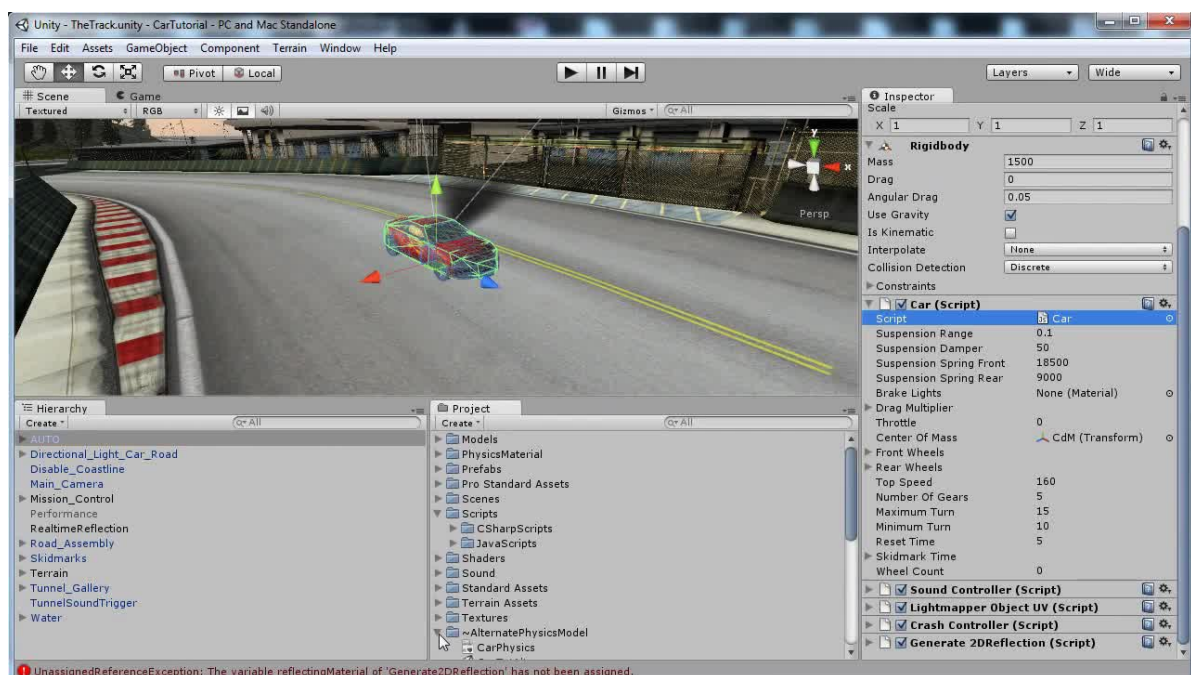
Una volta **determinata questa direzione di rotazione** “extra”, data dall'utilizzo del freno a mano, **applichiamo la rotazione all'auto**, ma questa volta intorno a un altro punto:

```
(frontWheels[0].localPosition + frontWheels[1].localPosition) * 0.5;
```

Questo punto è **a metà tra le due ruote anteriori**; il risultato della rotazione intorno a questo punto è che **la parte anteriore dell'auto** resta ferma mentre il resto ruota, producendo la tipica manovra – **fino al testa coda** – che si ha tirando il **freno a mano** mentre si sterza o curva in velocità.

A questo punto, **un ciclo di esecuzione è completo**: *Update* e *FixedUpdate* ripartiranno da capo al prossimo frame, ovviamente con nuovi input dall'utente e dalle condizioni dello scenario, calcolando nuove forze ed effetti... come in una vera esperienza di guida.

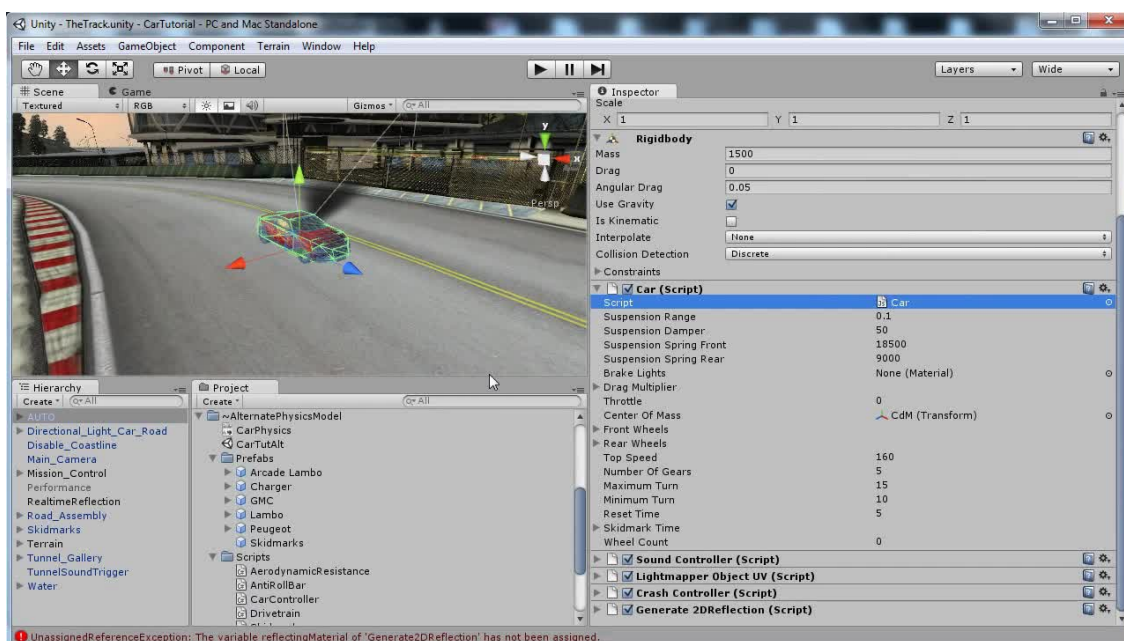
Questo completa la descrizione e la personalizzazione dell'auto, almeno per quanto riguarda le impostazioni di base: c'è sempre spazio per nuovi effetti e personalizzazioni.



ALTERNATE PHYSICS MODEL

Dedichiamo ora qualche parola al pacchetto *AlternatePhysicsModel*.

La cartella contenente i dati di questo motore fisico alternativo si trova all'interno del **Project** e contiene script e Prefab per simulare un **comportamento fisico “reale” dell'auto**.

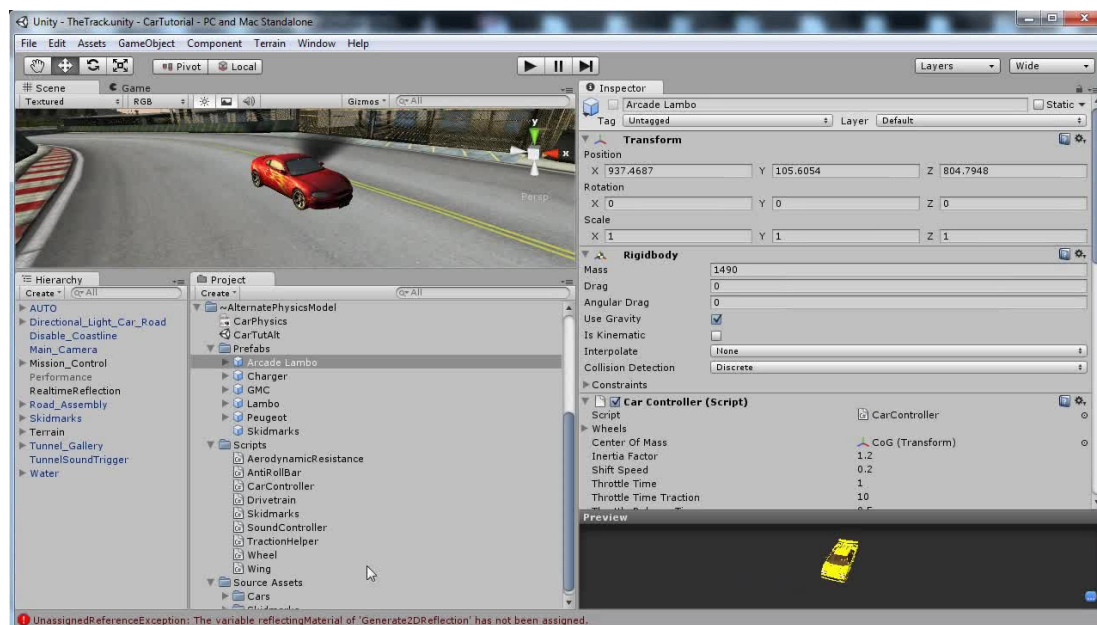


La soluzione presentata in questo pacchetto non usa i **Wheel Collider** ma dei **Collider** realizzati mediante script usando *Physics.Raycast*, più un altro **sistema di formule** per calcolare le **forze** da applicare all'auto.

In realtà, probabilmente non vi serve sapere COME funziona davvero il tutto, ma basterà **aggiungere i Prefab** e provare a cambiare e personalizzare i valori; tra l'altro, **tutti i parametri sono documentati negli script** mediante i commenti, per spiegarne le funzioni.

Nella cartella ci sono **5 Prefab** di automobili e di strisce (*skidmarks*).

Per provarli, **trascinate il Prefab di un'auto a piacere** e quello delle strisce nella scena (anche se, probabilmente, quello delle strisce ci sarà già); dovrebbe essere possibile guidare sin da subito l'auto, in modalità **Play** ovviamente, con i tasti freccia.



Ci sono 4 diverse auto, ciascuna con le sue caratteristiche realistiche, più un'auto sportiva, con una configurazione più... “arcade”.

Tutti gli script si basano **sulle reali leggi della fisica**, ad eccezione dello script *TractionHelper*, creato **per rendere le auto più controllabili** mediante input digitali (tastiera, mouse) quando si va sottosterzo; in pratica, tenta di simulare ciò che avviene nella realtà con i **sistemi ESP** (*Electronic Stability Program*, controllo elettronico della stabilità).

Il **Car Tutorial** originale di Unity non descrive in dettaglio gli script inclusi in *AlternatePhysicsModel*, per cui ora segue solo una breve descrizione di tali elementi, non una trattazione approfondita del loro contenuto e del loro funzionamento.

Abbiamo:

- ***AerodynamicResistance.cs***: una copia di questo script andrebbe assegnata ad ogni auto per calcolare l'attrito o resistenza aerodinamica.
- ***AntiRollBar.cs***: aggiungerne opzionalmente uno per asse (cioè per coppia di ruote) per simulare gli stabilizzatori anti-rollio.
- ***CarController.cs***: per gestire l'input. Ce ne vuole uno per auto. Modificare questo script per cambiare il modo in cui viene “pilotata” l'auto o per implementare una gestione via intelligenza artificiale o via rete, da parte quindi di altri giocatori, delle altre auto. Serve

anche per impostare certi parametri del telaio, come centro di massa e inerzia.

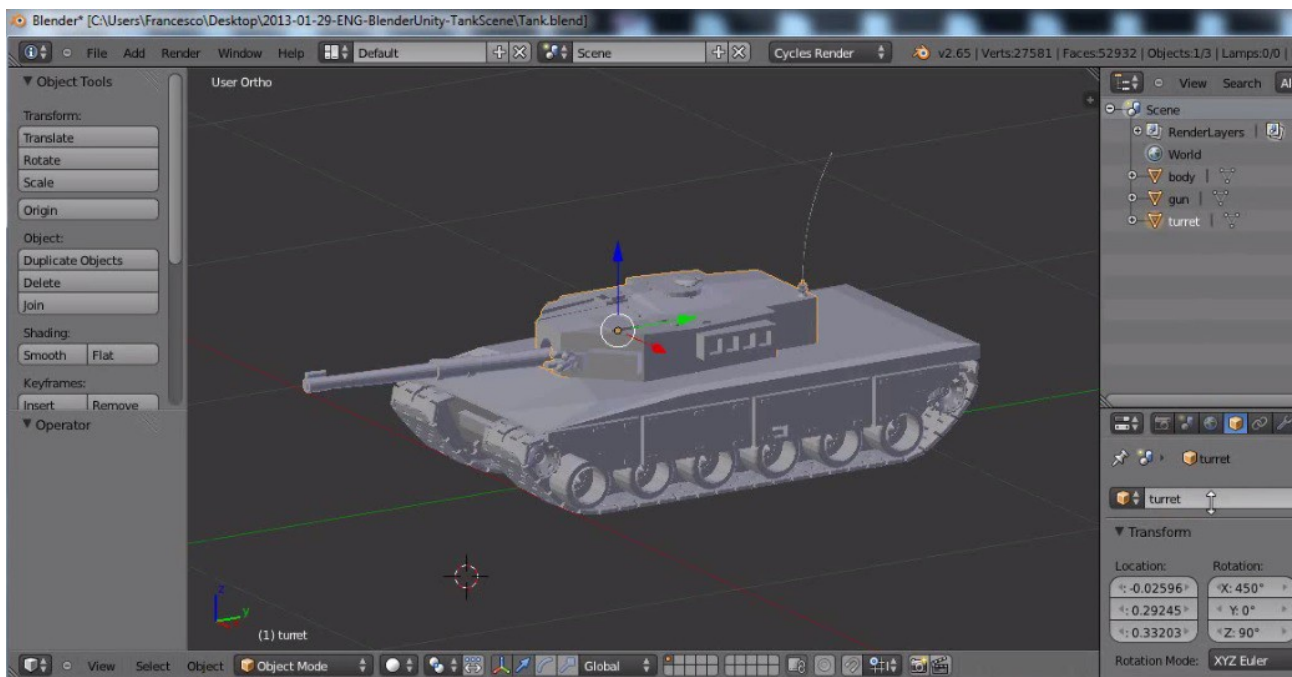
- **DriveTrain.cs**: qui vengono specificate marce e caratteristiche del motore. Ce ne vuole uno per auto.
- **Skidmarks.cs**: gestore globale delle strisciate. Aggiungete un *Prefab Skidmark* alla scena, che poi userà questa classe per gestire e renderizzare le strisce per tutte le auto.
- **SoundController.cs**: semplice classe per riprodurre i suoni del motore, delle frenate, eccetera... aggiungerne una per auto.
- **TractionHelper.cs**: aggiungerne uno per auto, per rendere il veicolo più stabile (ma è opzionale).
- **Wheel.cs**: simula le caratteristiche di aderenza, trazione e sospensioni, inoltre sostituisce il **WheelCollider** di Unity.
- **Wings.cs**: aggiungerne uno o più per simulare le forze aerodinamiche dell'auto (serve a migliorare l'aderenza al suolo).

Unity + Blender: rotazione con parenting e script Javascript

In questo tutorial parleremo un po' di **Blender** e un po' di **Unity**, mostrando come impostare una mesh composta da più elementi e con **Pivot Point** specifici in **Blender** e come importarla, con tali impostazioni, in **Unity**, dove tra l'altro vedremo come accedere, via script **JavaScript**, a oggetti figli di un oggetto principale e come ruotarli intorno a uno o più assi.

Si tratta quindi di un **tutorial** rivolto a chi ha già una **conoscenza di base** dei due programmi.

Partiamo dalla **scena 3D di Blender** visibile nell'immagine seguente, dove sono presenti tre mesh: corpo, torretta e cannone, relativi alle varie parti mobili (cingoli non animati, quindi) del carro armato.



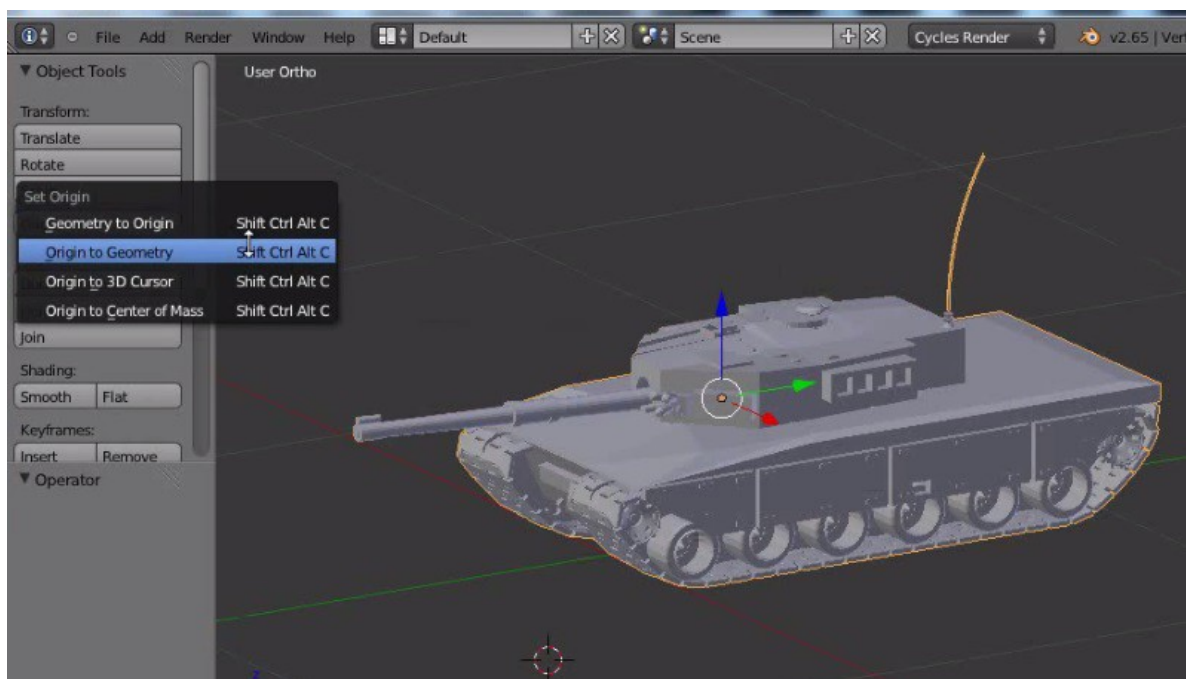
Selezionando i vari elementi, si nota che l'**Origin** dei vari oggetti non è nel baricentro o, nel caso del cannone, nel perno della rotazione, che dovrebbe trovarsi all'interno della torretta.

Notiamo inoltre che **gli oggetti non sono imparentati**, ossia si tratta di elementi separati, senza una catena gerarchica.

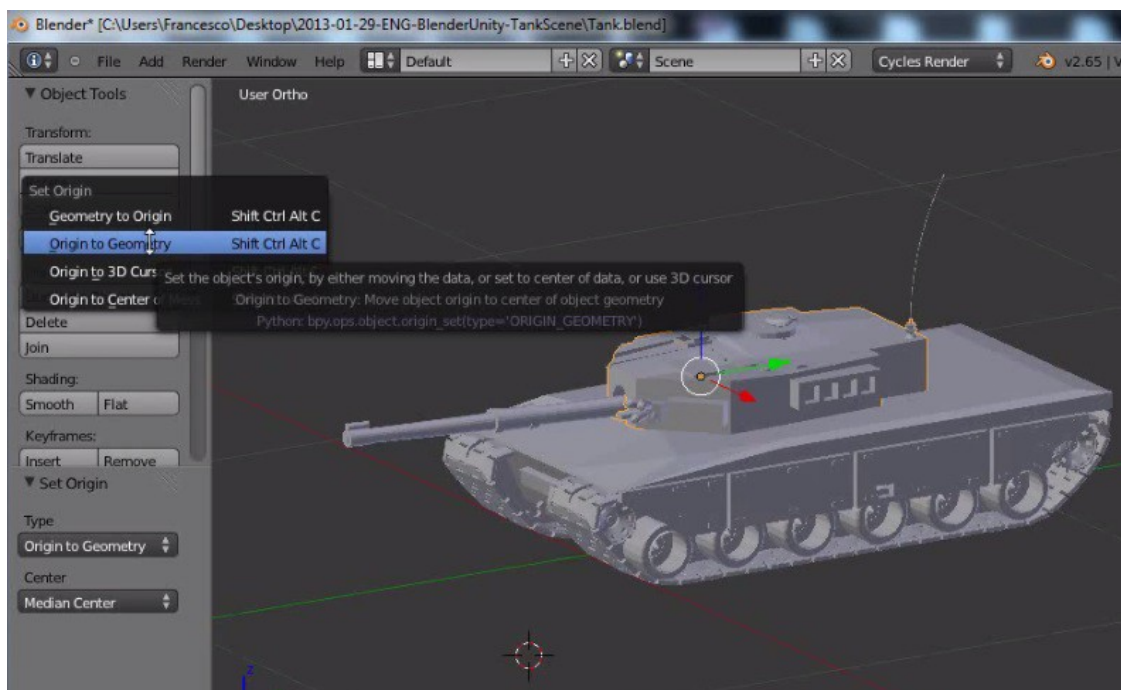
In **Unity** è possibile **imparentare gli oggetti** semplicemente trascinando un elemento su un altro nel pannello **Gerarchia**, per rendere il primo figlio del secondo, tuttavia un'operazione del genere, fatta con questo **asset** importato così com'è, farebbe perdere la relazione **Prefab** all'oggetto di scena... e, in generale, è meglio lavorare con i **Prefab**, anche perché, come visto nei dieci capitoli introduttivi su **Unity**, modificando il file *.blend* in **Blender** avremo le modifiche applicate in automatico da **Unity** a tutti gli oggetti della scena **istanze di quel Prefab**.

In **Blender**, quindi, dobbiamo fare **due operazioni preliminari** su questo modello: impostare correttamente le **Origins** degli elementi e **imparentarli**. Iniziamo dalle **Origins**.

Selezioniamo **il corpo del cannone**: questo è semplice, nel senso che l'**Origin** qui può coincidere tranquillamente con il baricentro della mesh, per cui, con tale oggetto selezionato in *Object Mode*, clicchiamo su **Origin** nella *Tool Shelf* e scegliamo “**Set Origin to Geometry**”. Fatto!



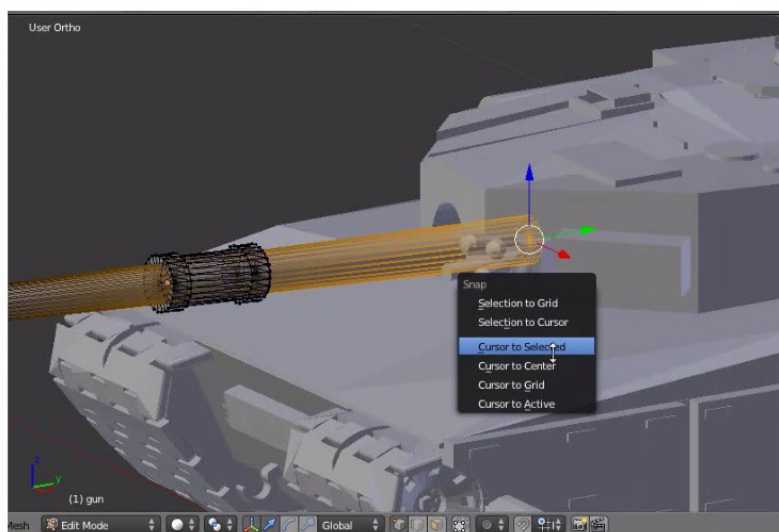
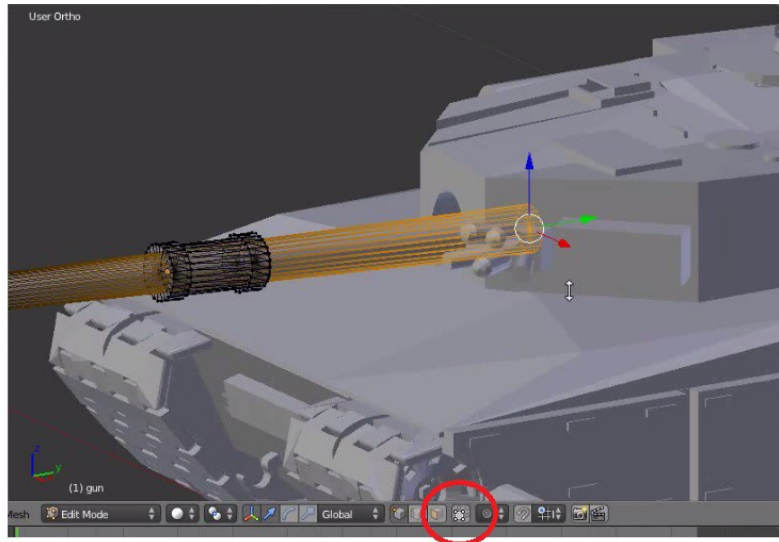
Anche per la **torretta** possiamo fare un reset al volo, basato sul baricentro, per cui, anche qui, selezioniamo la mesh, in *Object Mode*, clicchiamo su *Origin* e scegliamo “**Set Origin to Geometry**”.



Per quanto riguarda il **cannone**, il discorso è diverso. Il punto è questo: quando, in **Unity**, scriveremo gli script di rotazione per gli oggetti, **la rotazione avverrà intorno alle Origins** degli stessi, per cui ruotando il cannone con queste impostazioni (provate ad esempio con R Z Z), avremo una rotazione intorno al centro della mesh, che chiaramente non va bene!



Selezioniamo quindi il cannone e passiamo in **Edit Mode**; nel caso in cui i vertici della mesh che si trovano dentro la torretta non dovessero essere visibili, **disattiviamo lo Z Clipping** nell'header della 3D View e selezioniamo i vertici della circonferenza alla base del cannone, quindi spostiamo lì il **3D Cursor** con **SHIFT S** e **“Cursor to Selected”**.

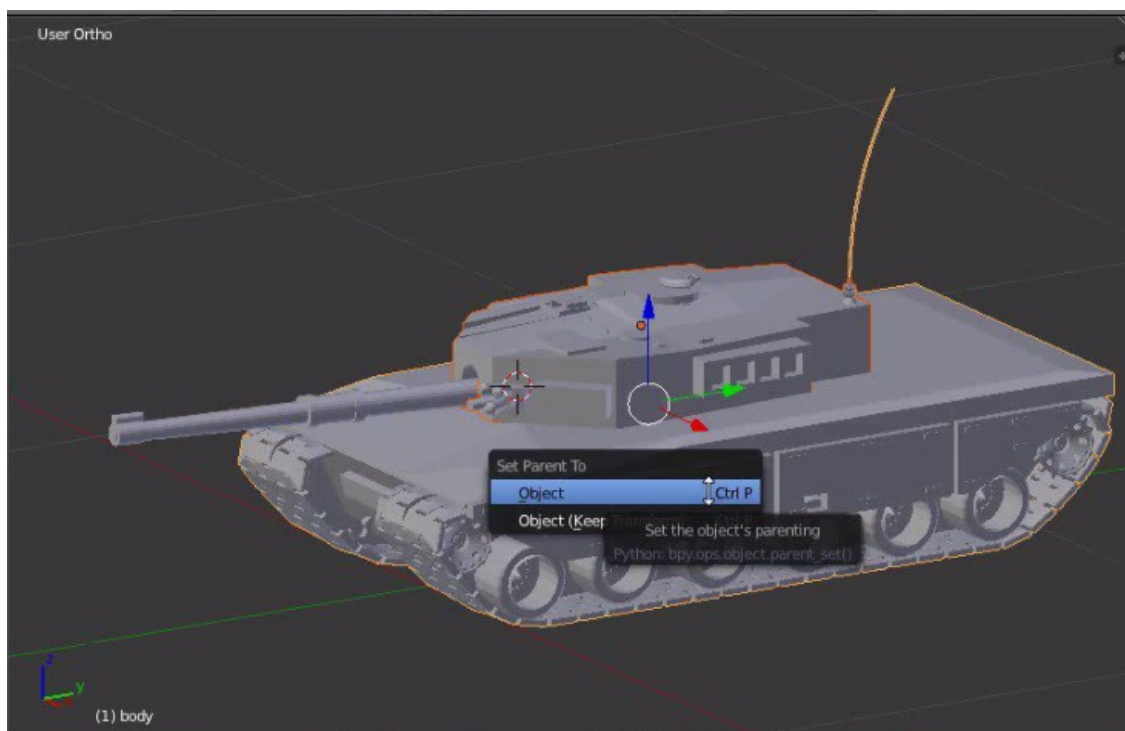


Adesso torniamo in *Object Mode*, clicchiamo su **Origin** nella *Tool Shelf* e scegliamo “**Origin to 3D Cursor**”. Come potete vedere premendo R Z Z e muovendo il mouse, ora la rotazione avviene correttamente.



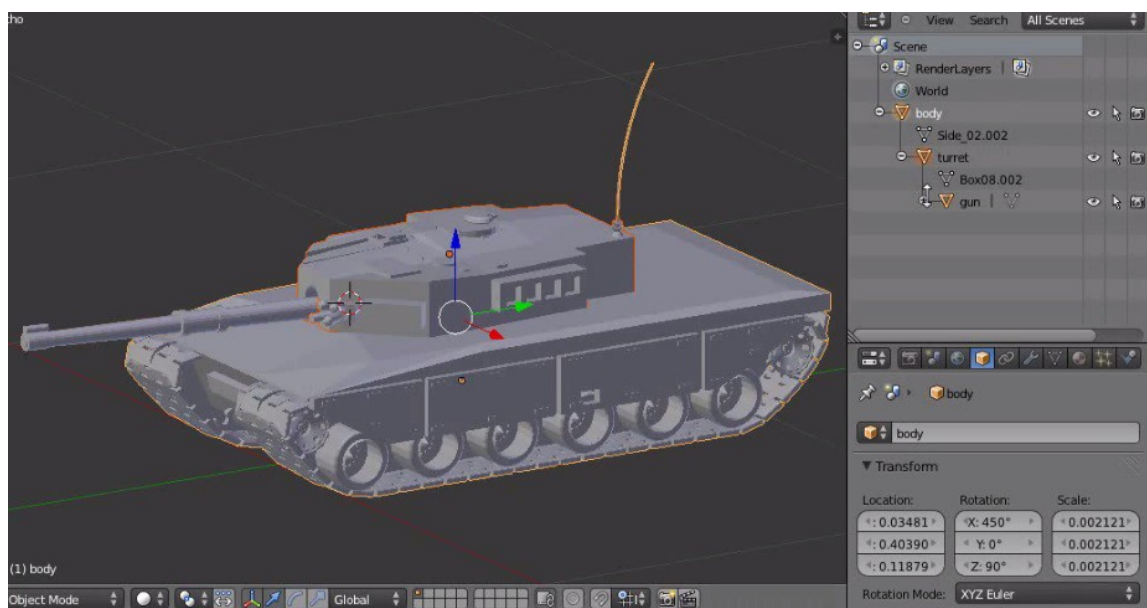
Resta quindi il discorso del **parenting**, per creare la **gerarchia**, ma questo è semplice: la relazione di parentela avviene con **una selezione multipla**, selezionando **prima i figli** e, per ultimo, il padre, quindi premendo **CTRL P** e confermando.

Selezioniamo **il primo figlio della catena**: il cannone, che deve essere figlio della torretta (ruotando la torretta, ruoteremo anche il cannone); selezioniamo, quindi, la torretta, con **SHIFT e click destro** per ottenere una selezione multipla; premiamo ora **CTRL P** e confermiamo l'operazione nel menù che apparirà a video (immagine nella pagina seguente).



Selezioniamo ora **solo la torretta**, come primo elemento, quindi – sempre con una selezione multipla – selezioniamo il **corpo principale** e imparentiamo i due elementi con **CTRL P** e confermando.

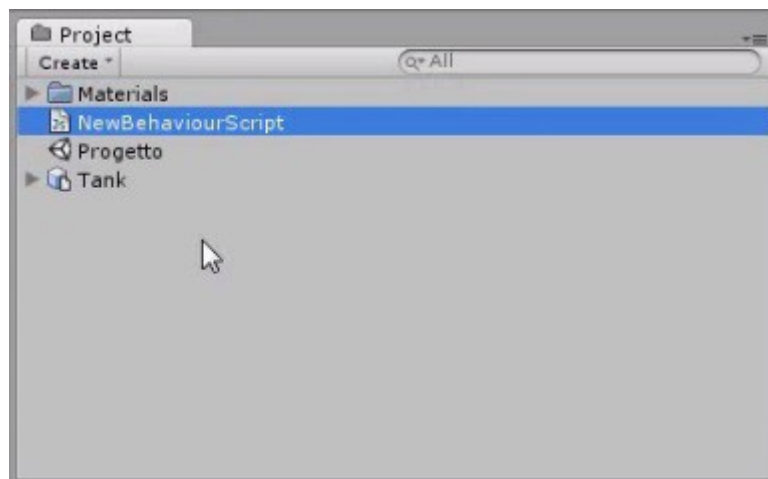
Ricapitolando, **la catena è questa**: il corpo principale, che è genitore della torretta, che a sua volta è genitore del cannone.



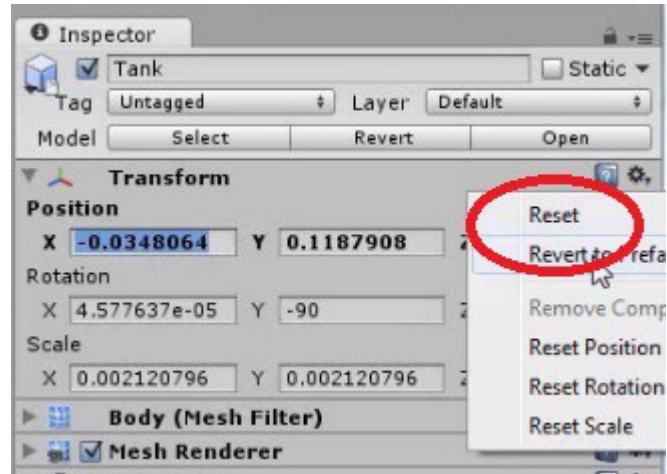
Le **trasformazioni** vanno applicate quindi in questo modo: la **traslazione** va fatta SOLO per il corpo principale che, essendo in cima alla catena, muoverà tutti gli altri elementi collegati; il **corpo** può avere, ovviamente, una **rotazione** sinistra-destra, che ruoterà tutto l'oggetto; la **torretta** dovrà avere una rotazione sinistra-destra; il **cannone** avrà una rotazione su-giù.



Salviamo il file **.blend** per apportare le modifiche e spostiamoci in **Unity 3D**, quindi **importiamo** proprio il file **.blend** come **nuovo Asset del progetto** (nell'immagine seguente, è l'elemento chiamato “*Tank*”).

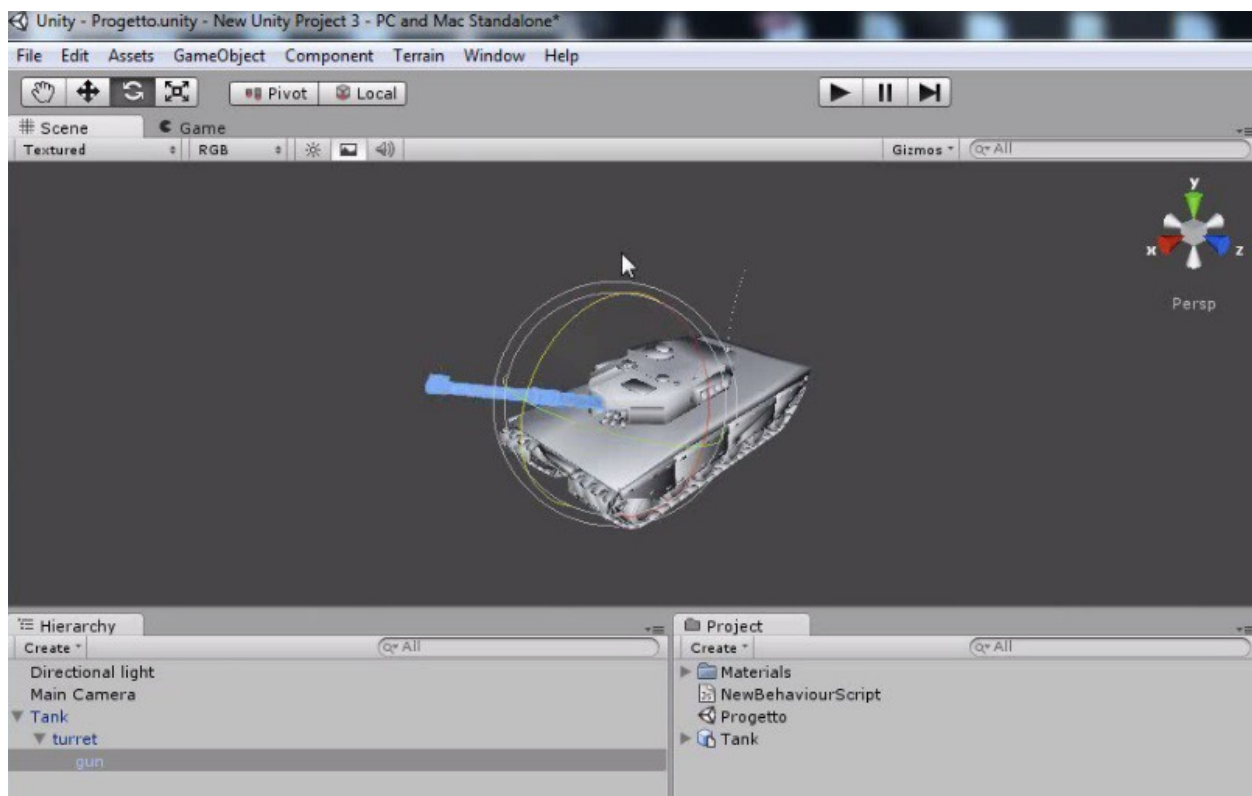


Importato l'Asset nel **Project**, creiamone un'istanza nella scena 3D, quindi selezioniamo tale oggetto nella **Gerarchia** e resettiamo le coordinate di posizione, rotazione e scaling (per farlo, cliccate sulla ruota dentata nell'Inspector e scegliete “Reset”, come visibile nell'immagine a lato), posizionando magari la **Main Camera** in (0, 0, -250), in modo da inquadrare il carro armato da una certa distanza, ma comunque potete traslare, ruotare e scalare gli elementi a piacere, come preferite, non è importante.



Per migliorare l'ombreggiatura, inseriamo una semplice fonte di luce **Directional Light** nella scena e orientiamola a piacere.

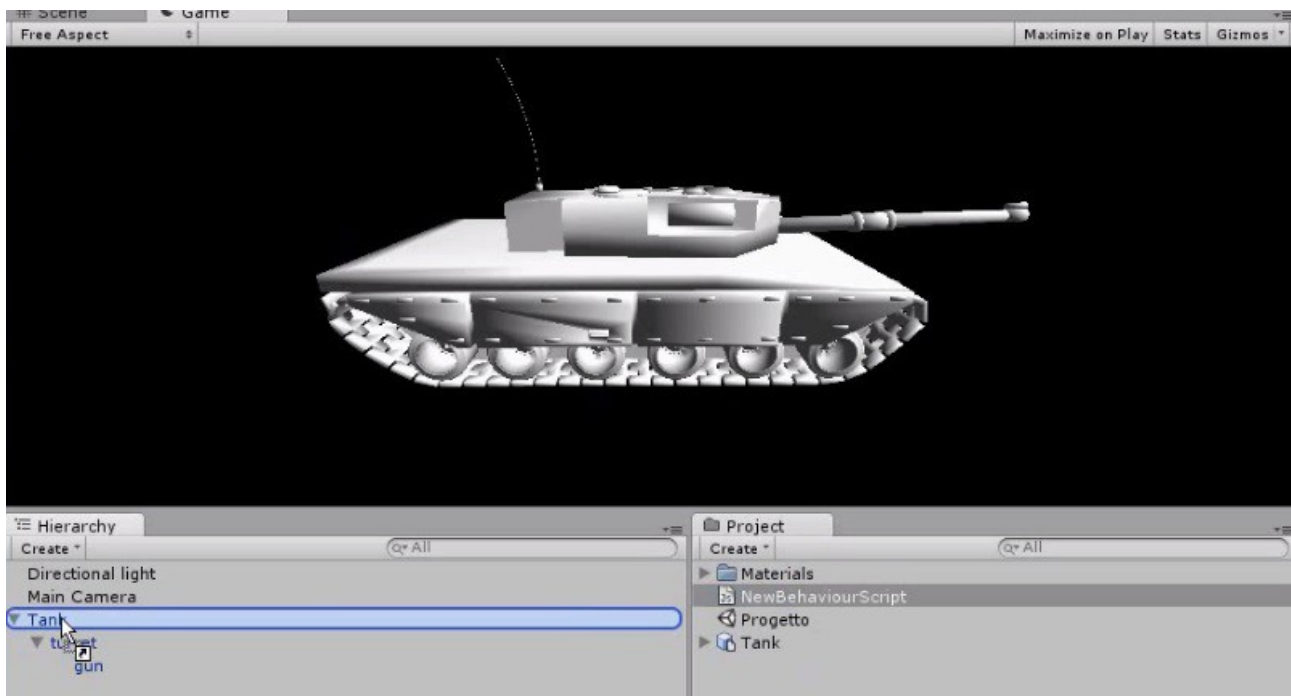
Come mostra la catena gerarchica del carro armato nel pannello **Gerarchia** (immagine seguente), gli elementi sono **correttamente imparentati** e provvisti dei giusti **Pivot Point** (le *Origins*).



È giunto quindi il momento di creare **uno script per guidare il carro armato e orientare torretta e cannone**; per chi è alle prime armi con **Unity**, i problemi in questa fase sono due: **come associare lo script solo al corpo** ma accedendo poi agli elementi figli e **come ruotare correttamente gli elementi**. Li risolveremo entrambi.

Per prima cosa, **creiamo uno script JavaScript**, con nome a piacere, nel **Project**.

Associamo immediatamente tale script **al carro armato**, trascinandolo sull'elemento **“corpo”**, la base del carro armato, quindi apriamo lo script in MonoDevelop.



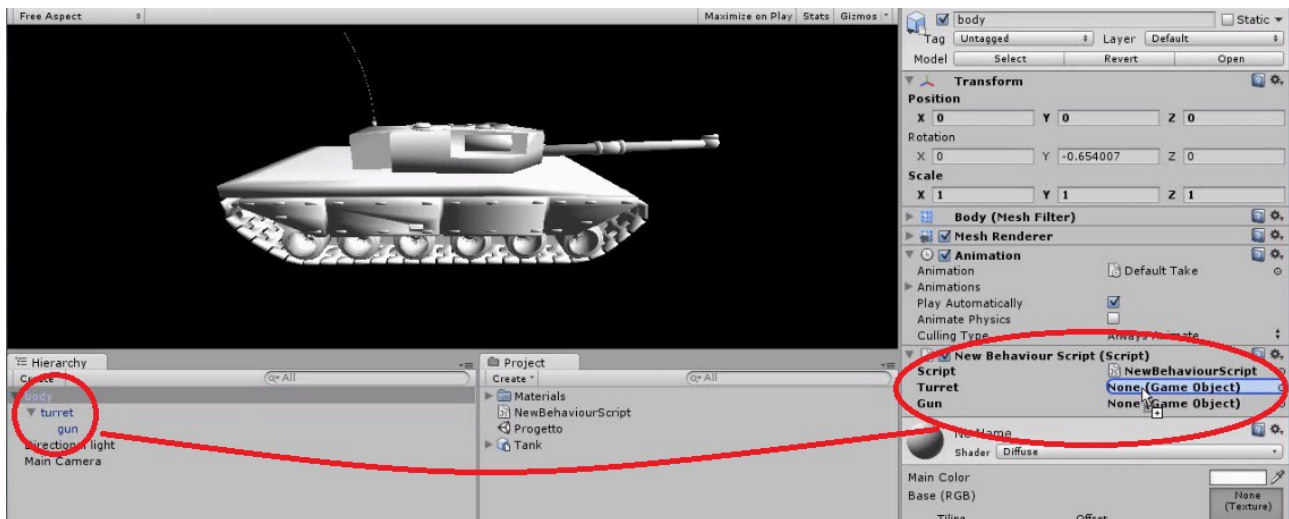
Per i nostri scopi ci bastano le funzioni di default, ***Start*** ed ***Update***, anzi in realtà modificheremo solo ***Update***, ma prima dobbiamo **creare due variabili di tipo Game Object**, chiamate **“torretta”** e **“cannone”**, fuori da qualsiasi funzione, nello **“spazio globale”** dello script, per cui scriviamo:

```
var torretta : GameObject;  
var cannone : GameObject;
```

Salviamo lo script per apportare le modifiche e torniamo alla scena di **Unity**.

Per associare le parti dell'oggetto alle due variabili, è sufficiente **trascinarle** sui relativi slot nell'*Inspector*. Ecco come procedere:

- selezioniamo, nel pannello **Gerarchia**, l'oggetto **corpo** (la base del carro armato), individuando nell'*Inspector* il Component Script con le due variabili non ancora inizializzate;
- clicchiamo **SENZA RILASCIARE** sull'elemento **torretta**, nella **Gerarchia**, e **trasciniamo** tale voce nello slot **“torretta”** nel Componente Script della base, nell'*Inspector*;
- clicchiamo **SENZA RILASCIARE** sull'elemento **cannone**, nella **Gerarchia**, e **trasciniamo** tale voce nello slot **“cannone”** nel Componente Script della base, nell'*Inspector*.



Abbiamo associato gli oggetti di scena alle variabili nello script. Possiamo quindi tornare allo script in MonoDevelop e **scrivere il codice di movimento e rotazione** nella funzione **Update**.

Iniziamo dalle cose semplici: la **traslazione**.

Osservando l'oggetto carro armato nella scena 3D in modalità **Local**, notiamo che il **verso frontale** è dato dall'asse X positivo, quindi **l'avanzamento riguarderà tale coordinata**.

Scriviamo quindi:

```
if(Input.GetKey(KeyCode.UpArrow)) transform.position.x += 1;  
if(Input.GetKey(KeyCode.DownArrow)) transform.position.x -= 1;
```

Salvando lo script per rendere effettive le modifiche e avviando ora il gioco con **Play**, con quasi assoluta certezza si manifesterà un **problema** (e, se non ora, si manifesterà in altre forme dopo, fidatevi): probabilmente, non vedrete il carro armato.

Vi dico subito qual è l'origine del problema: la componente “*Animation*” presente di default nel **Prefab** importato da **Blender**. Dobbiamo quindi selezionare l'oggetto corpo nella **Gerarchia** e **rimuovere questo Component *Animation*** dall'Inspector.

Adesso, avviando il gioco e premendo i tasti freccia in su e freccia in giù, **vedremo avanzare e indietreggiare il carro armato**.

In questo esempio NON vi mostrerò come far curvare il carro armato, evento “complesso” in un certo senso, ma come vi ho detto parleremo di come impostare una rotazione via script... e questo sì che lo vedremo, con la torretta e il cannone, che possono muoversi sul posto senza problemi.

Un errore comune, quando si deve programmare la rotazione di un elemento via script, è quello di **tentare di sommare o sottrarre valori** alle proprietà *rotation.x* e simili di un *transform*, in maniera del tutto simile a quanto visto poco fa con *transform.position.x*; in realtà, **per ruotare un elemento intorno a uno o più assi** dobbiamo usare la funzione ***Rotate*** di *transform*, che prende in input un *Vector3*, quindi tre valori: le entità delle rotazioni rispettivamente intorno agli assi X, Y e Z.

Creiamo i *Vector* delle rotazioni a parte, fuori dalla funzione *Update*, in modo da richiamarli senza ricrearne di nuovi ogni volta; in particolare, **la torretta deve ruotare intorno all'asse verticale**, ossia Y, il secondo componente della rotazione, per cui scriviamo:

```
var destra : Vector3 = Vector3(0.0, 1.0, 0.0);  
var sinistra : Vector3 = Vector3(0.0, -1.0, 0.0);
```

Adesso, **all'interno della funzione *Update***, scriviamo:

```
if(Input.GetKey(KeyCode.A)) torretta.transform.Rotate(sinistra);  
if(Input.GetKey(KeyCode.D)) torretta.transform.Rotate(destra);
```

Stiamo accedendo al *transform* dell'oggetto di gioco ***torretta*** mediante una **variabile** che è stata associata all'oggetto trascinando quest'ultimo dalla **Gerarchia** allo slot dello script nell'Inspector.

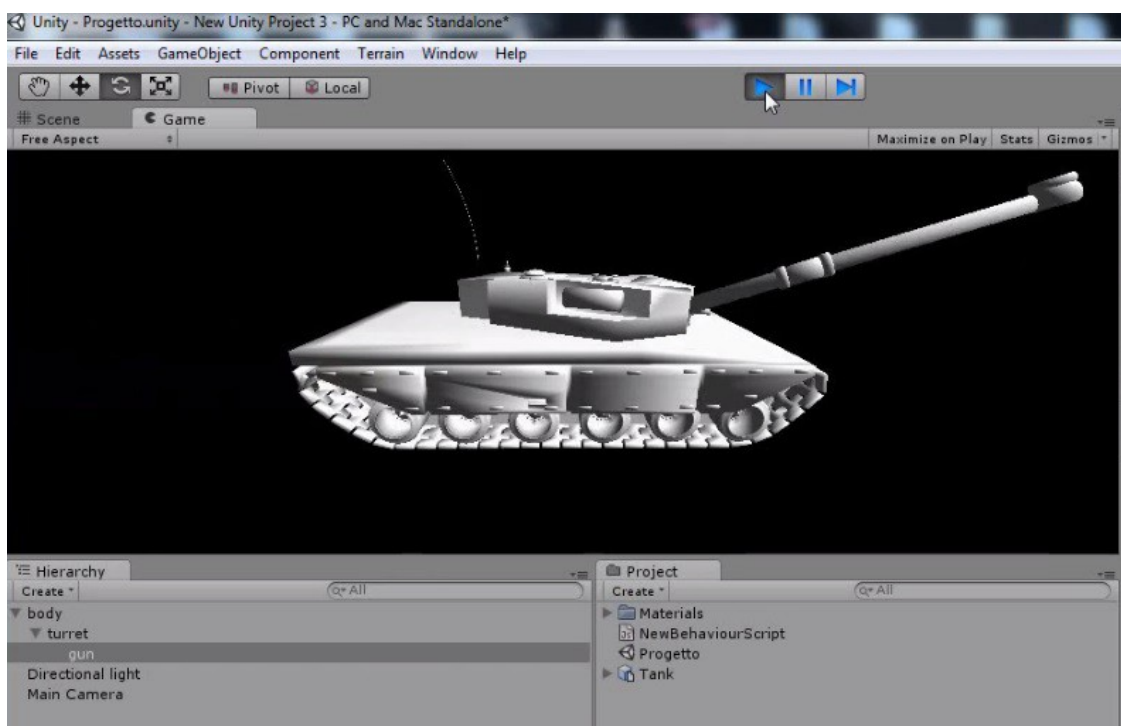
Per quanto riguarda **la rotazione del cannone**, il discorso è molto simile: dobbiamo creare dei *Vector3* per l'asse **Z locale**, come si può constatare selezionando il cannone nella scena 3D (dove si vede, tra l'altro, che il perno della rotazione coincide con l'Origin dell'oggetto così come lo abbiamo impostato in **Blender**), per cui fuori dalle funzioni scriviamo:

```
var sopra : Vector3 = Vector3(0.0, 0.0, 1.0);  
var sotto : Vector3 = Vector3(0.0, 0.0, -1.0);
```

quindi, nel corpo della funzione Update, aggiungiamo le seguenti righe di codice:

```
if(Input.GetKey(KeyCode.W)) cannone.transform.Rotate(sopra);  
if(Input.GetKey(KeyCode.S)) cannone.transform.Rotate(sotto);
```

Avviando il gioco nella scena 3D di **Unity** e premendo i tasti freccia avanti, freccia indietro e i tasti WASD, potremo spostare il carro armato in avanti e indietro, ruotare la torretta e modificare l'alzo.



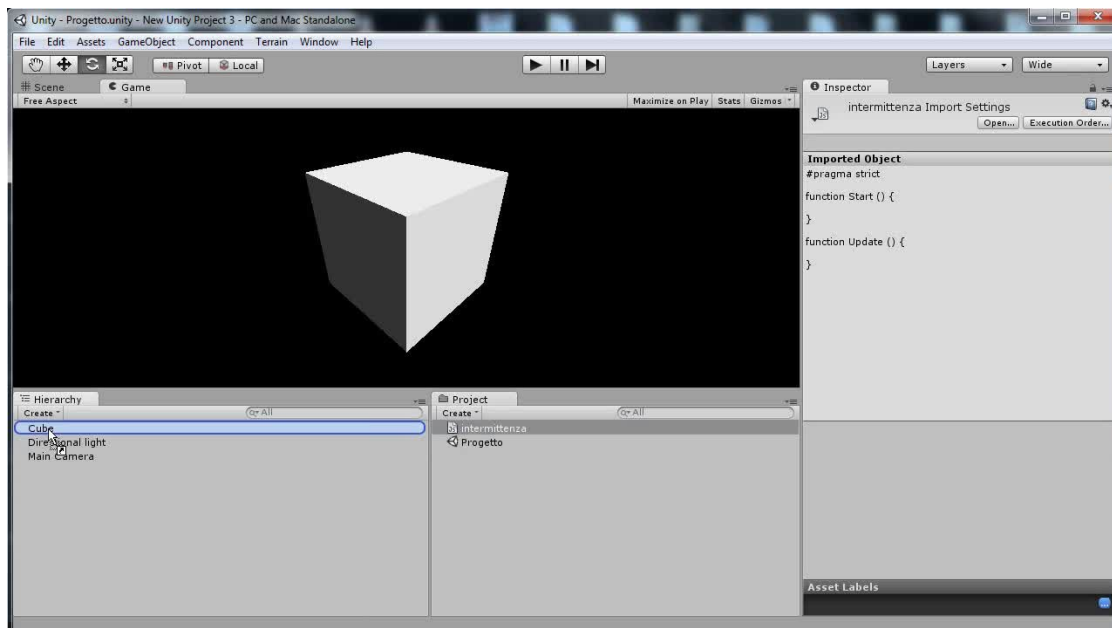
Per quanto riguarda il tutorial, è tutto; come “esercizi per casa”, potete provare a far curvare il carro armato e limitare, mediante controlli sugli angoli Y e Z, le rotazioni della torretta e del cannone, in quanto per ora sono completamente liberi di ruotare, cosa che, chiaramente, non va bene, soprattutto per quanto riguarda il cannone

* * *

L'operatore modulo, Time, renderer e “l'effetto intermittenza”

Questo è un breve tutorial di **programmazione JavaScript in Unity 3D**, in particolare sull'utilizzo di **Time** e dell'operatore modulo per ottenere un **effetto “intermittenza”**, per... qualsiasi cosa; nell'esempio pratico, lo useremo per far apparire e scomparire un oggetto.

Lo script di per sé può essere applicato a varie proprietà, ma per fare un esempio prendiamo un oggetto (nell'immagine sotto, si vede l'utilizzo di una primitiva Cube) di una scena di **Unity** e associamogli **uno script JavaScript** chiamato *“intermittenza.js”*, quindi apriamo tale script in **MonoDevelop** per iniziare a scrivere il codice necessario per implementare l'effetto.



Prima di scrivere, però, parliamo di quello che vogliamo ottenere: **cambiare il valore di una proprietà**, in questo caso ***renderer.enabled***, da *True* a *False* e viceversa ogni secondo, per ottenere appunto **l'effetto on/off**, a intermittenza. Dovremo quindi assegnare **un valore a *renderer.enabled***.

È possibile pensare a vari modi per risolvere questo problema ma il più semplice consiste nell'utilizzare l'**operatore modulo** sul valore del tempo corrente espresso in secondi.

L'operatore **modulo** restituisce il **resto della divisione** tra due valori.

Se dividiamo un valore per 2, il resto può essere 0 (nel caso di un numero pari) o 1 (per un numero dispari).

Time.time restituisce il **tempo corrente dall'avvio** dell'applicazione esprimendolo in **secondi** (e decimi, centesimi e millesimi di secondo); per vedere velocemente come viene espresso questo valore (di tipo *float*), all'interno della funzione **Update** scriviamo:

```
Debug.Log(Time.time);
```

quindi salviamo lo script per applicare le modifiche, torniamo nella finestra principale di **Unity** e premiamo **Play**: in basso, nella schermata di gioco, vedremo il **valore di Time**, in quanto la **funzione Log di Debug** (occhio alle maiuscole!) serve appunto a scrivere, in quella porzione di schermo, i valori passati come parametri (in questo caso, **Time.time**).

Torniamo allo script (cancellando, se vogliamo, l'istruzione debug inserita in **Update**).

Adesso scriverò una versione particolarmente brutta dello script di intermittenza, in pratica una bozza per isolare gli elementi che ci servono per ottenere l'effetto; dopo, farò la versione ottimizzata, che richiederà una sola linea di codice.

Creiamo una variabile di tipo intero per i **secondi**:

```
var secondi : int = Time.time;
```

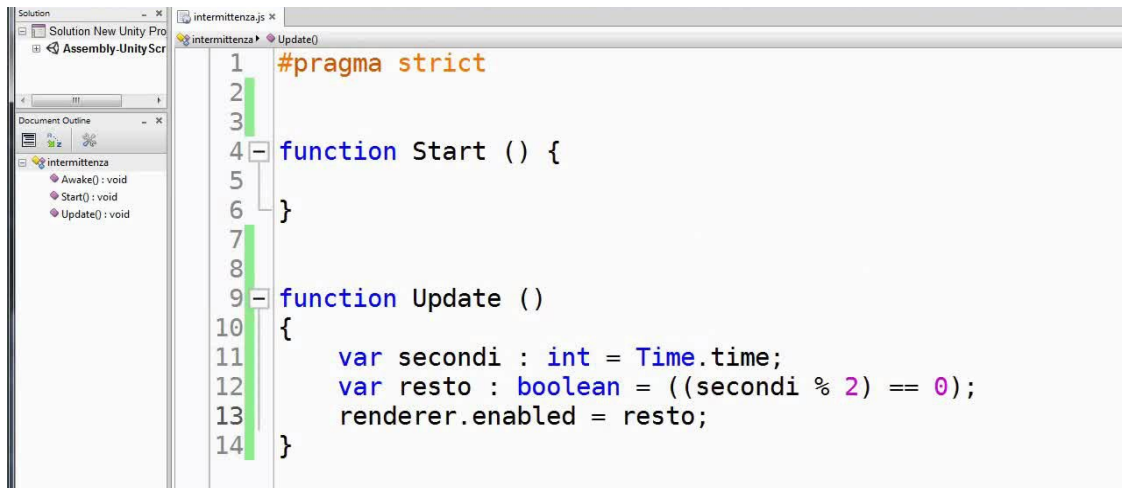
quindi una variabile booleana che sarà **True** se il **resto della divisione** tra secondi e 2 sarà 0 (quindi se secondi sarà un numero pari) e **False** altrimenti (numero dispari):

```
var resto : boolean = ((secondi % 2) == 0);
```

vi ricordo infatti che il **==** va letto come una domanda, ossia “è uguale a...?”, e **Unity** prende direttamente la risposta a questa domanda, ossia **True** o **False**; infine, diamo a **renderer.enabled** questo booleano:

```
renderer.enabled = resto;
```

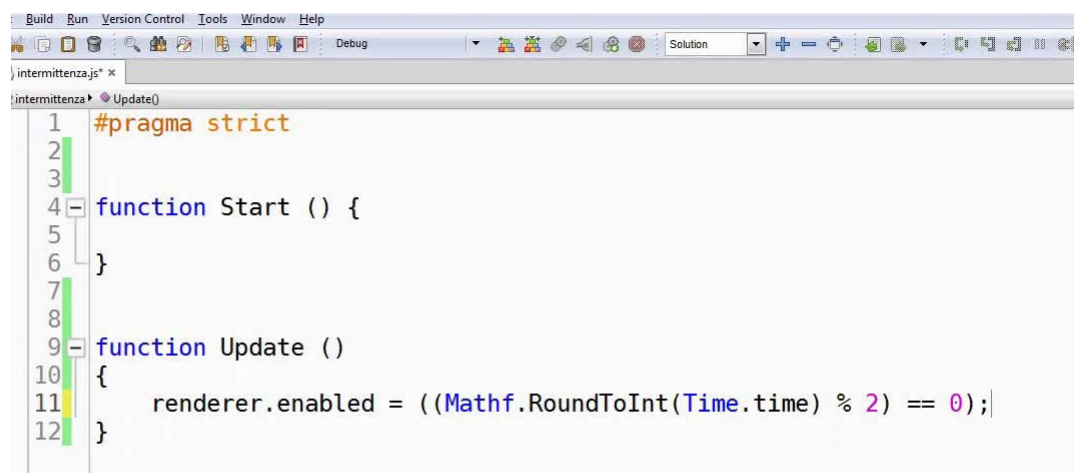
il tutto ovviamente verrà **aggiornato automaticamente** da **Unity** a ogni esecuzione di **Update**.



Salviamo tutto, torniamo nella schermata principale del programma e premiamo **Play** per osservare il risultato: di funzionare, funziona, ma lo script è particolarmente brutto, in quanto per ogni chiamata della funzione **Update** creiamo ogni volta due variabili e in generale il tutto potrebbe essere “compattato”, per così dire, in un'unica istruzione, ossia:

```
renderer.enabled = ((Mathf.RoundToInt(Time.time) % 2) == 0);
```

Cos'è quel *Mathf.RoundToInt*?



Si tratta di una **funzione nativa di Mathf**, che sta per “*Math*” (matematica) e “*float*”, quindi un set di strumenti relativi ai numeri in virgola mobile; in particolare, **RoundToInt** serve a convertire in intero un numero *float*, quale è appunto *Time.time*.

Perché non possiamo utilizzare direttamente **Time.time** con modulo 2?

Perché, se ricordate l'output di Debug.Log, quel valore contiene delle **frazioni di secondo**, dei **decimali** (infatti è un float), mentre % 2 riguarda dei valori interi... in generale, quando trattate vari dati, tenete sempre a mente il TIPO di dati da elaborare e, se si tratta di tipi diversi, come *float* e *int*, dovete fare le dovute conversioni... vi faccio un esempio pratico: sarebbe come sommare un valore in km, ad esempio 2 km, a uno in metri, ad esempio 3 metri, dicendo “il risultato fa 5”... beh non ha alcun senso!

Tra l'altro, noi non diamo a **renderer.enabled** il valore numerico ottenuto dal %, ma il risultato della domanda, per così dire, fatta dal ==, ossia: il risultato di modulo vale 0 (i secondi sono pari)? Se sì, *True* (e *renderer* è *enabled*), altrimenti *False*.

Per vedere il risultato finale, salviamo lo script, torniamo alla finestra principale del programma e premiamo **Play** (qui non ci sono screenshot perché non avrebbero molto senso; a video, si vedrà apparire e scomparire il cubo, alternativamente, ogni secondo).

Quanto visto tratta un caso specifico ma gli elementi esaminati, come **Time.time**, la conversione da *float* a *int* e l'utilizzo dell'operatore modulo, si prestano a molte applicazioni.

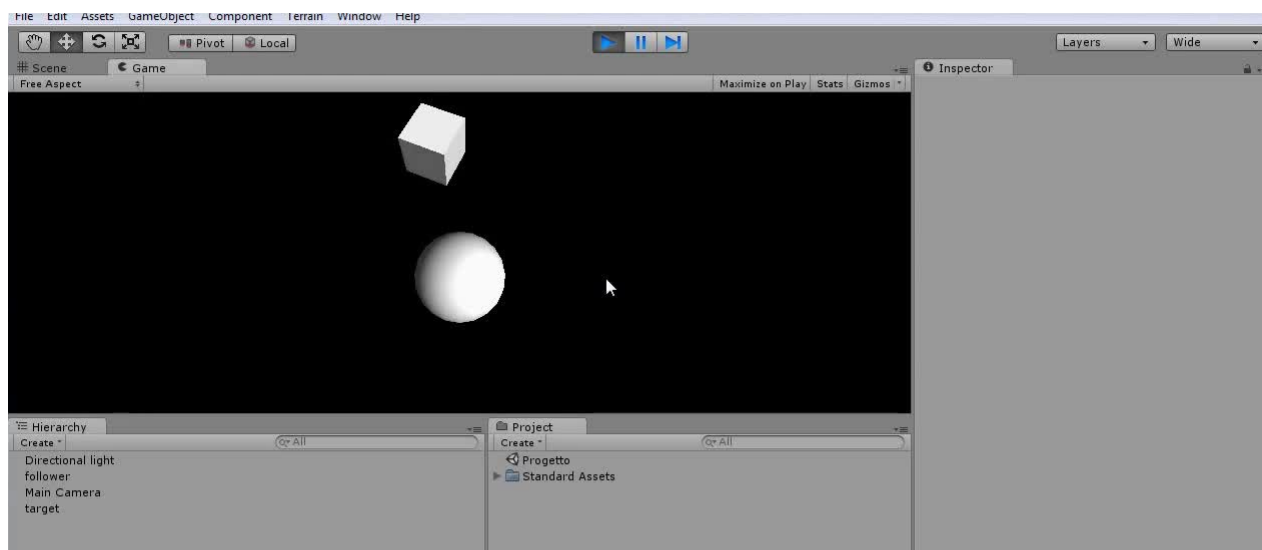
* * *

Script JS: semplice Constraint di posizione per seguire un oggetto

In questo brevissimo **tutorial di programmazione JavaScript in Unity**, nato da una richiesta di un utente del Forum *effe12.com*, vedremo come realizzare un semplice **Constraint di traslazione** per un oggetto, ossia uno **script** che servirà a **spostare un oggetto** di scena in base agli **spostamenti di un altro** elemento (il suo *“target”*, “bersaglio”).

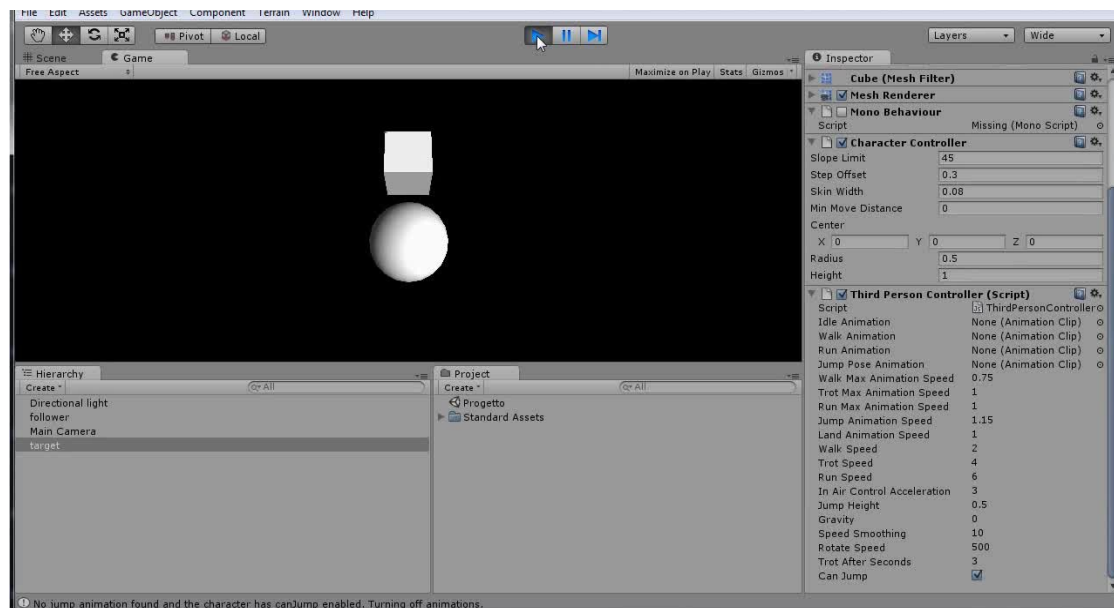
Non si tratta quindi di dare a un oggetto la stessa posizione dell'altro, che può essere animato o spostato a piacere, ma di **fargli fare gli stessi spostamenti** (qui non copieremo anche rotazioni o scaling), quindi di utilizzare la variazione nel ***transform.position***; per questo motivo, dovremo prima **calcolare tale variazione** tra un frame e quello precedente.

Iniziamo da una scena con due oggetti, uno dei quali è provvisto di un **FPS Controller** in modalità **Third Person Controller** (ho mostrato come importare il **package FPS** e configurare questi *controller* nel decimo e ultimo tutorial sulle basi di **Unity 3D**); qui sto usando questo controller per comodità, per spostare l'oggetto principale (il cubo, visibile nell'immagine seguente), detto *“target”*, senza dover creare script per l'occasione.

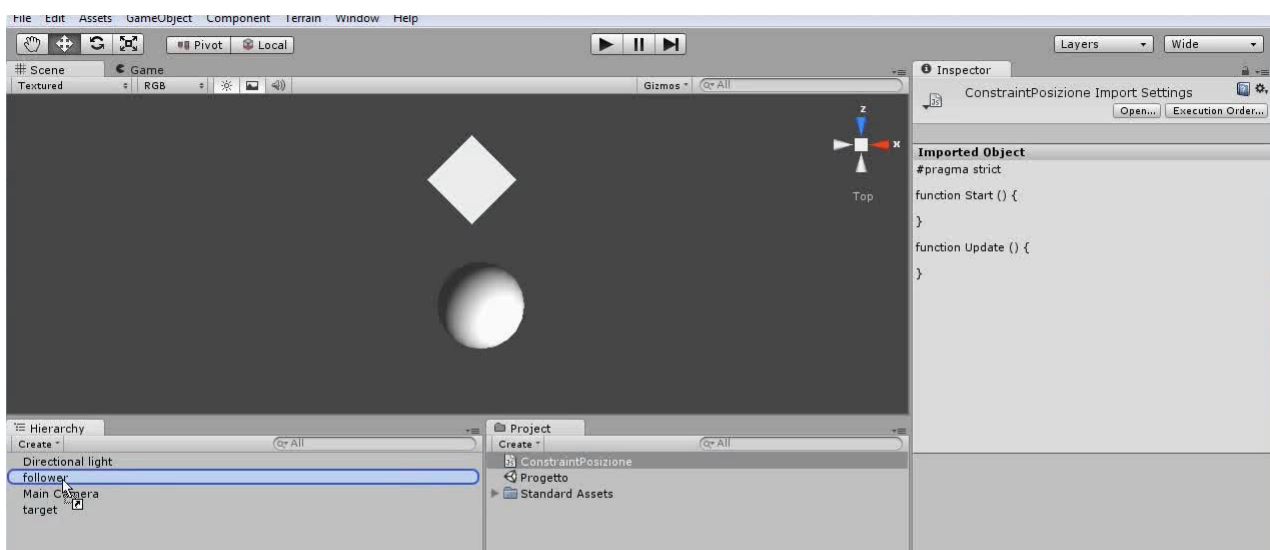


Nella sezione dello script dell'**FPS Controller**, nell'Inspector di “*target*”, ho impostato **Gravity** a 0, per non far cadere l'oggetto nel vuoto all'avvio del gioco.

Avviando quindi il gioco adesso e premendo i tasti freccia, sposteremo l'oggetto “*target*”, mentre l'oggetto “*follower*”, la sfera, resta fermo, in quanto non provvisto di script **Constraint**, ossia di uno script che leghi la sua posizione a quella del “*target*”.



Torniamo in modalità Editing e aggiungiamo uno script **JavaScript** al progetto, chiamandolo ad esempio “**ConstraintPosizione**”. Associamo questo script alla **sfera**, oggetto “*follower*”, tramite trascinamento, quindi apriamolo in **MonoDevelop** per digitare le istruzioni.



Come detto, vogliamo poter recuperare, ad ogni frame, l'entità dello spostamento dell'oggetto bersaglio e applicare tale differenza all'oggetto che possiede lo script.

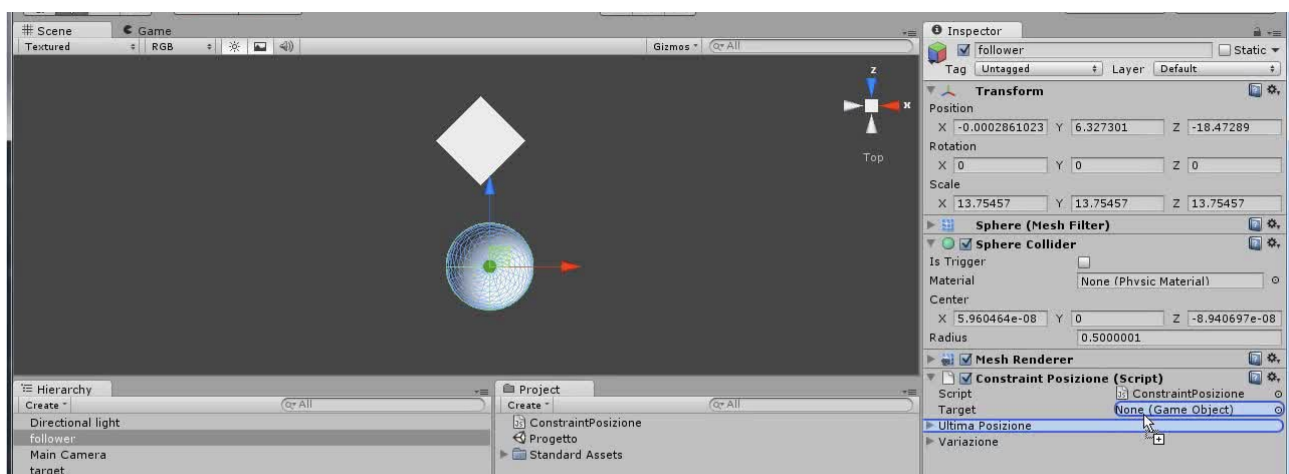
Per poter fare questo calcolo ci servono **tre variabili**: una variabile di tipo **GameObject** che identifica l'oggetto *target*, il bersaglio da seguire; un **Vector3** con le coordinate del target al frame precedente e un **Vector3** con la differenza tra le coordinate al frame precedente e quelle al frame corrente (questo secondo Vector3, come vedremo, è superfluo, ma per il momento procediamo passo per passo, poi puliremo il codice).

Nello spazio globale dello script, fuori da qualsiasi funzione, scriviamo quindi:

```
var target : GameObject;  
var ultimaPosizione : Vector3;  
var variazione : Vector3;
```

Chiaramente, all'avvio dell'applicazione non c'è un “**frame precedente**”, quindi l'inizializzazione della variabile “**ultimaPosizione**” (nome scelto a caso, potete cambiarlo come preferite) avverrà nella funzione **Start**, cioè all'avvio dello script, poi in **Update** calcoleremo la differenza e aggiorneremo **ultimaPosizione** con la posizione attuale, pronti per un altro ciclo al frame successivo.

Salviamo lo script per applicare le modifiche e torniamo alla schermata principale di **Unity**, in quanto l'associazione tra la variabile **target** e l'oggetto di scena “**target**” va fatta via drag-and-drop; in particolare, selezioniamo “**follower**” nella **Gerarchia**, quindi facciamo click senza rilasciare su “**target**” e, trascinando, spostiamo questo elemento nello slot “**target**” dello script “**ConstraintPosizione**” di “**follower**”, in modo da associare l'oggetto di gioco alla variabile.



Torniamo in **MonoDevelop** per continuare a scrivere lo script.

La variabile `target` è quindi già inizializzata, per cui occupiamoci dell'inizializzazione dei due **Vector3** scrivendo all'interno di **Start**:

```
ultimaPosizione = target.transform.position;
variazione = Vector3(0.0, 0.0, 0.0);
```

A questo punto, all'interno della funzione **Update**, per ogni frame dobbiamo calcolare lo spostamento; per nostra fortuna, **ci basta il segno “meno”** per fare la differenza tra vettori **Vector3**, quindi **Unity** farà la **somma algebrica** componente per componente restituendo un **Vector3**.

Scriviamo quindi:

```
variazione = target.transform.position - ultimaPosizione;
```

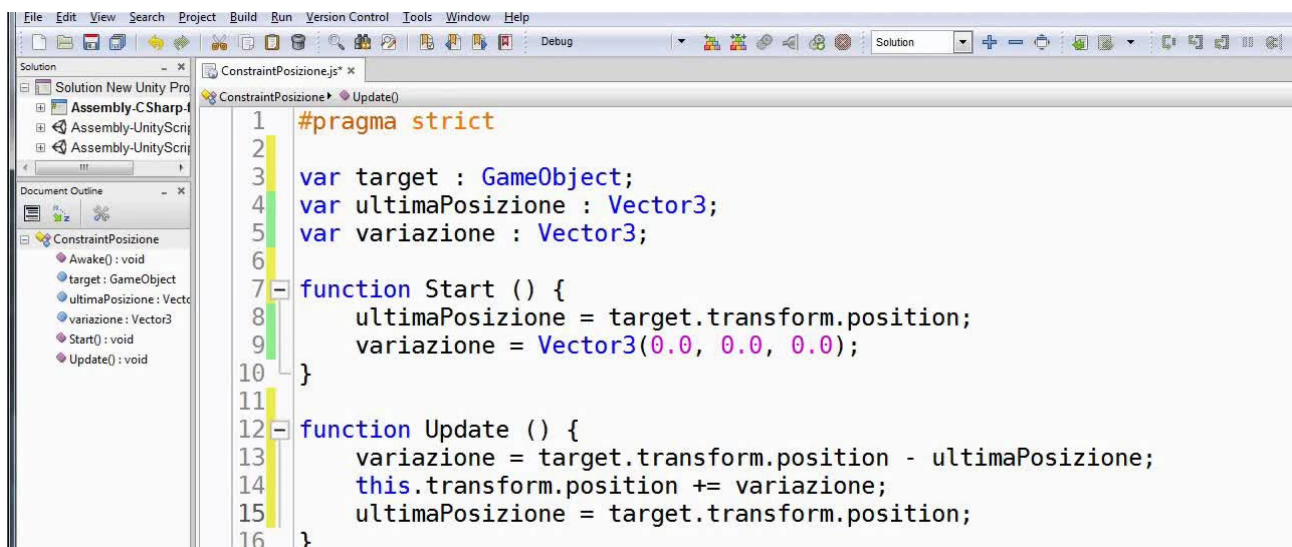
quindi applichiamo questa variazione alla posizione del follower con:

```
this.transform.position += variazione;
```

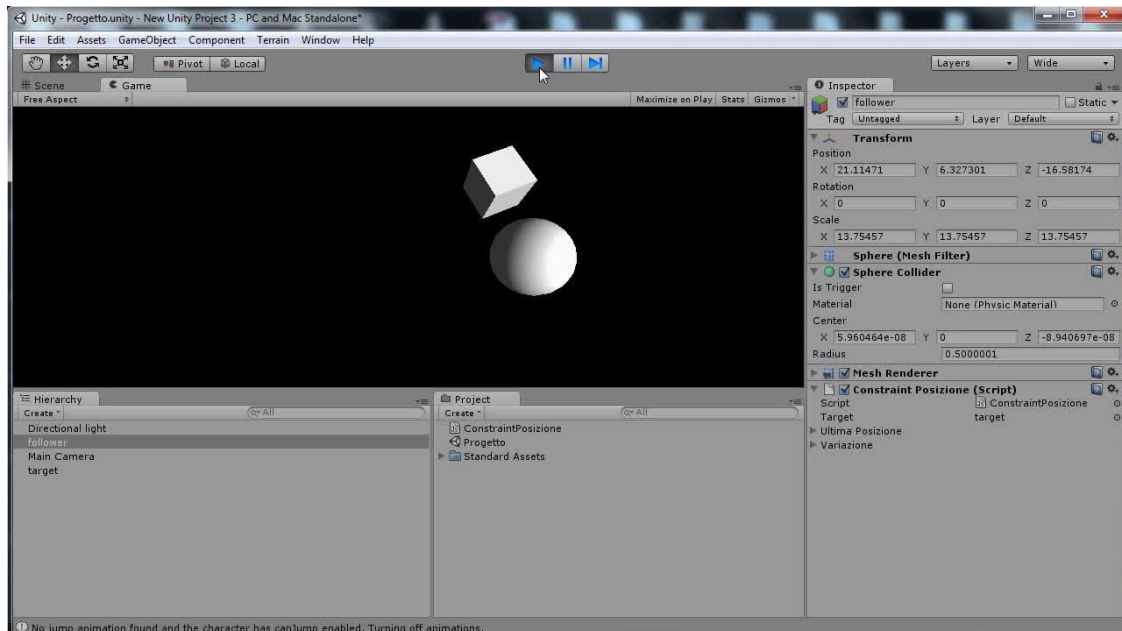
(il `+=` è la forma compatta per dire “al valore originale della tua posizione somma il valore della variazione e sovrascrivi il valore originale con il risultato ottenuto”).

Per concludere il ciclo, aggiorniamo la posizione **ultimaPosizione** con la posizione del **target**, pronti per la prossima iterazione, con:

```
ultimaPosizione = target.transform.position;
```



Salvato lo script, è possibile osservare il risultato nella schermata principale di gioco premendo **Play** e spostando l'oggetto *target* con i tasti freccia, agendo sul suo **FPS Controller**: l'oggetto *follower* replicherà gli spostamenti.



Questo è chiaramente un esempio molto semplice di “*constraint di posizione fatto in casa*”, ma come **esercizio di programmazione** di base può andare bene.

A ben vedere, comunque, la variabile “*variazione*” è superflua, perché può essere calcolata al volo per sommarla alla posizione corrente, ma qui bisogna fare un discorso sulla leggibilità e la flessibilità del codice, anche in vista di modifiche future: sta a voi decidere se lasciarla, rendendo il codice di più facile lettura e modifica (agendo su “*variazione* =” per modificare i calcoli), oppure togliere ogni riferimento a tale variabile e mettere quei calcoli direttamente a destra di “*this.transform.position +=*”.

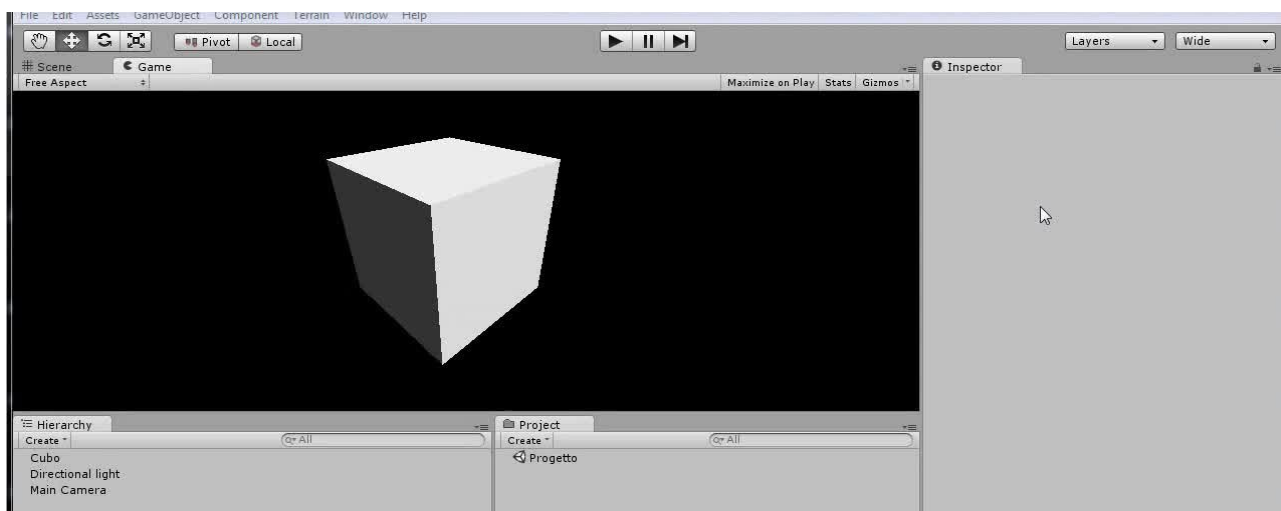
Script JS: Time.deltaTime: a cosa serve, come si usa

Questo è un breve tutorial di **programmazione in Unity 3D**, in particolare sull'elemento ***Time.deltaTime***, un'informazione molto importante per realizzare operazioni, effetti o animazioni **indipendenti dal numero di fotogrammi** al secondo (e, quindi, valide su molte configurazioni hardware e software).

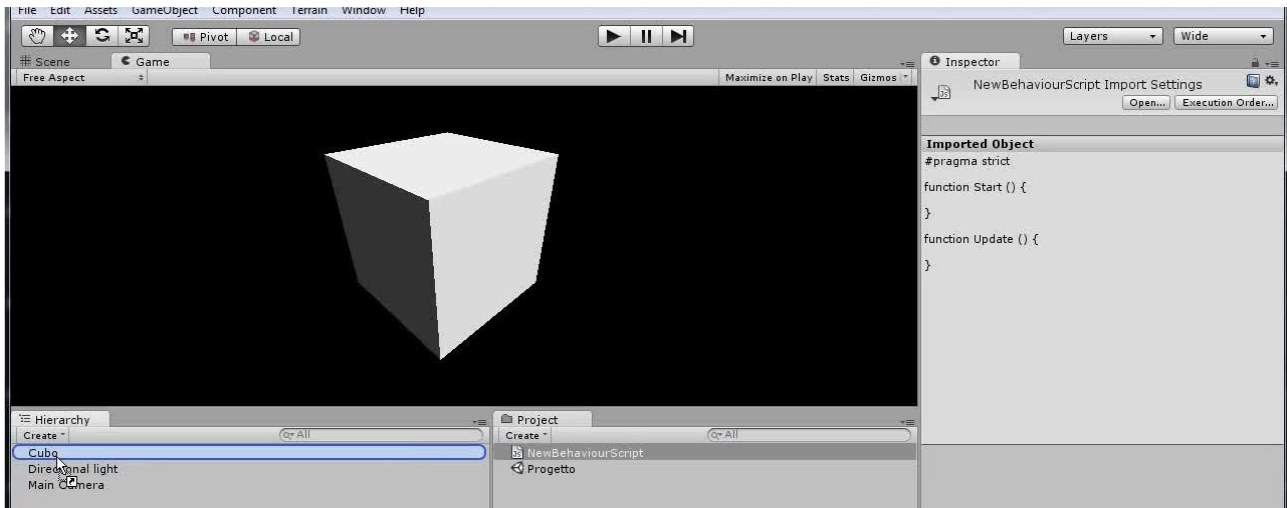
La documentazione online dice che ***Time.deltaTime*** è un valore *float* (cioè un numero con decimali) che indica “*il tempo, in secondi, impiegato per completare il fotogramma precedente*”.

La documentazione ci dice inoltre che questo valore può essere utilizzato per effettuare **operazioni indipendenti dal frame rate** del programma e, tra pochissimo, vedremo un esempio; di contro, ***Time.deltaTime*** non dovrebbe essere utilizzato nelle funzioni ***OnGUI***, per la gestione dell'interfaccia del programma, in quanto la funzione nativa ***OnGUI*** può essere richiamata da ***Unity*** anche più volte all'interno di un unico frame di gioco, per aggiornare l'aspetto.

Nella scena **Unity 3D** visibile nello screenshot qui sotto è presente solo una mesh di tipo **cubo**; vedremo infatti un esempio semplicissimo, per non rischiare di fare confusione, per cui possiamo partire da questa configurazione.



Inseriamo nel progetto uno script di tipo **JavaScript**, con nome a piacere, quindi associamolo al **cubo** mediante trascinamento e apriamolo in **MonoDevelop** per modificarlo.



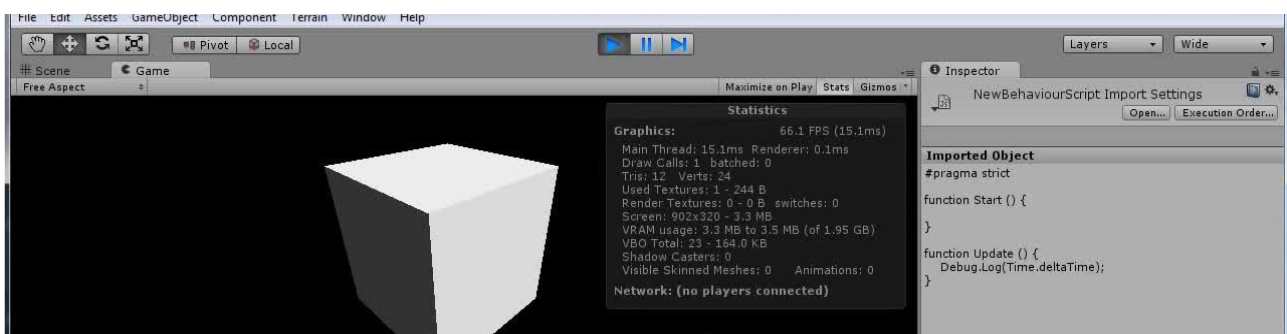
Giusto per vedere un esempio di valori di *Time.deltaTime*, nella funzione *Update* scriviamo la seguente istruzione:

```
Debug.Log(Time.deltaTime);
```

Come visto nel capitolo precedente, *Debug.Log* mostra il valore passato come argomento in basso nella finestra di **Unity** durante la simulazione del gioco.

Salviamo quindi lo script per apportare le modifiche, torniamo alla finestra principale di **Unity** e premiamo **Play**, dando un'occhiata al valore che apparirà in basso e che, come detto, indica **il tempo in secondi** impiegato per processare l'ultimo frame di gioco.

NOTA: durante la fase Play di esecuzione del gioco, potete cliccare sulla tab “*Stats*” che apparirà in alto a destra nella finestra di gioco per visualizzare a video **altre statistiche sull'esecuzione** (immagine seguente), come il tempo impiegato dal thread principale e il frame rate di esecuzione.



Premiamo **Stop** e torniamo allo script in **MonoDevelop** per fare ora un esempio pratico di utilizzo di *Time.deltaTime*.

All'interno della funzione *Update*, che come sappiamo viene eseguita automaticamente da **Unity** a ogni frame di gioco, scriviamo:

```
var spostamento : float = Time.deltaTime * 10;  
transform.Translate(0, 0, spostamento);
```

Queste due righe di codice avranno l'effetto, verificabile salvando lo script e premendo **Play** in **Unity**, di **spostare l'oggetto di 10 unità** (ad esempio, 10 metri) al secondo lungo l'asse Z, ma la cosa interessante è che questo spostamento sarà di **10 unità al secondo su qualsiasi macchina**, indipendentemente dal frame rate o dal *deltaTime* del gioco.

Supponiamo infatti di avere un gioco che vada, sul primo pc, a 10 frame al secondo, quindi ogni frame sul pc A ha un **deltaTime medio di 0,1 secondi**, mentre sul secondo pc lo stesso gioco va a 20 frame al secondo, quindi ogni frame sul pc B dura 0,05 secondi (ossia, un **deltaTime medio di 0,05 secondi**). Ad ogni frame, facciamo fare uno spostamento sull'asse Z pari a **10 volte il deltaTime**, quindi ad ogni frame, sul pc A, lo spostamento sarà di $0,1 * 10 = 1$ unità, mentre sul pc B sarà di $0,05 * 10 = 0,5$ unità.

Come detto precedentemente, sul pc A abbiamo 10 frame al secondo, quindi avremo 1 unità a frame per 10 frame = 10 unità al secondo; sul pc B, dove ci spostiamo di 0,5 unità a frame, abbiamo 20 frame al secondo, quindi li avremo... 10 unità al secondo, proprio come sul pc A!

	fps	deltaTime	Unità per frame	Unità per sec.
PC A	10	0.1	1	10
PC B	20	0.05	0.5	10

A livello visivo, **il movimento sul pc A sarà meno fluido** e più “a scatti”, perché effettuato con **meno frame** e, quindi, **spostamenti maggiori**, più evidenti; al contrario quello sul pc B sarà più fluido, ma il punto importante è che, indipendentemente dal frame rate, lo spostamento sui due pc è in entrambi i casi di 10 unità al secondo.

Qui abbiamo fatto un esempio con un *deltaTime* fisso per tutti i frame, ma potete fare degli esempi scrivendo successioni di *deltaTime* casuali, alla fine avremo comunque **spostamenti coordinati**, perché all'abbassarsi del frame rate avremo *deltaTime* e quindi spostamenti maggiori, e viceversa.

Time.deltaTime è un'informazione utilizzata in molti script, anche semplici, per cui è importante capire bene a cosa serve e come impiegarla, magari partendo da un esempio semplicissimo come quello trattato, per passare poi a casi più complessi; nel prossimo capitolo l'informazione *Time.deltaTime* verrà utilizzata all'interno di un'altra funzione nativa di **Unity**, *Slerp* dell'oggetto *Quaternion*, per ottenere rotazioni “dolci” degli elementi.

Script JS: Quaternion, Slerp e le rotazioni animate

In questo tutorial parleremo di una funzione, **Slerp** della classe **Quaternion** di **Unity**, che ci consente di **ruotare gli oggetti** per far raggiungere loro un certo orientamento in un determinato **arco di tempo**. Utilizzeremo, tra l'altro, anche **Time.deltaTime**, visto nel capitolo precedente.

Slerp sta per “**Spherical Linear Interpolation**” ed è un concetto di Computer Grafica in generale, non solo di **Unity**, introdotto da Ken Shoemake per realizzare **l'animazione di rotazioni nello spazio 3D sfruttando i quaternioni**, argomento matematico che NON tratteremo in questo tutorial... qui utilizzeremo un approccio pratico, limitandoci a mostrare quando e come va utilizzata **Quaternion.Slerp**.

Per questo tutorial, quindi, diremo solo che i **Quaternioni** sono un modo per esprimere gli **orientamenti**, e spero che i matematici mi perdonino per questa semplificazione!

La documentazione di **Unity** dice che la funzione **Slerp** “*effettua l'interpolazione tra un quaternione A e un quaternione B in un tempo T*”, ossia **realizza un'animazione** per andare da un orientamento ad un altro in un certo arco di tempo, ma questa definizione è fuorviante e può dare luogo, nella pratica, a diversi problemi.

Pensate all'**interpolazione** come ad una retta (o, nel caso di **Slerp**, una curva) che va da un punto A (l'orientamento iniziale, con i suoi valori) a un punto B (i valori dell'orientamento finale).

Lungo la retta avremo quindi dei **valori** che via via si allontaneranno da quelli di A per avvicinarsi a quelli di B (**interpolando**, appunto, gli estremi). Adesso, assegnate al punto A il valore 0.0 e al punto B il valore 1.0: in questo modo, dicendo ad esempio 0.5 saremo a metà retta, quindi un valore di orientamento a metà strada tra quello iniziale e quello finale... non posso proseguire con

altri esempi di altri valori, cioè non posso dire che ad esempio a 0.25 saremo a un quarto della differenza dei valori, perché in realtà l'interpolazione non è lineare, ma il senso è questo: **“mappate” il range 0.0 – 1.0 sull'interpolazione dei valori da A a B**; ecco, quel range è il tempo *t*, per cui scrivendo ***Quaternion.Slerp(A, B, 0.2)***, otterremo l'orientamento che l'oggetto dovrebbe avere a un quinto (0.2, tra 0.0 e 1.0), dicevo un quinto della rotazione da A a B.

Il metodo prende quindi **tre parametri** (il quaternione d'inizio, quello di destinazione e il float che indica il tempo) e restituisce un oggetto **Quaternion**.

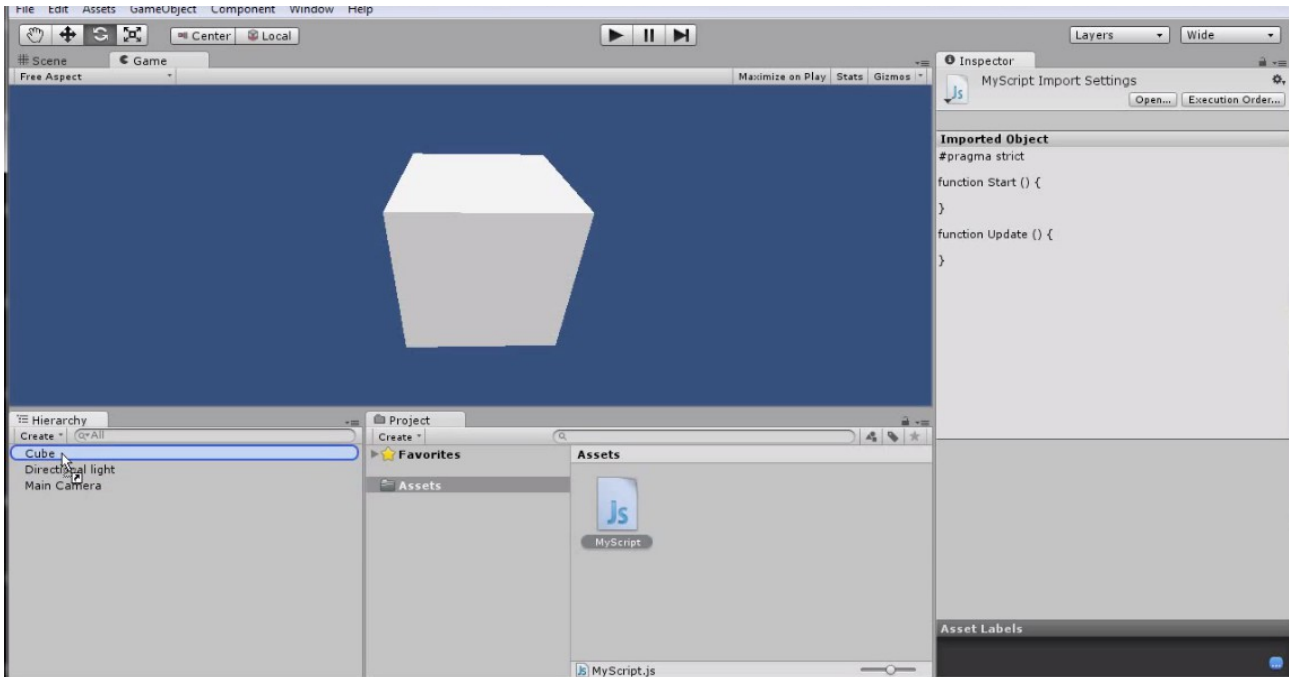
Questo esempio si ricollega anche al minitutorial sull'utilizzo della **funzione Rotate**, al posto del **campo rotation**, per effettuare le rotazioni; *transform.rotation* vuole infatti un *Quaternion*, che indica un orientamento, ed è per questo motivo che, provando a dare a *rotation* un valore in gradi o radianti si hanno risultati strani e, in quel tutorial, vi ho detto di utilizzare la funzione *Rotation*.

Il **Quaternion** calcolato dalla funzione ***Slerp***, invece, può essere assegnato tranquillamente a ***transform.rotation*** come angolo valido per l'orientamento; essendo ***transform.rotation*** un **Quaternion**, inoltre, può essere utilizzato come parametro per ***Slerp***, ad esempio per usare come orientamento ***target*** l'orientamento di un altro oggetto di scena.

Detto a parole, tutto questo può risultare un po' confusionario; per schiarirci le idee, facciamo prima **un esempio pratico semplice**, poi ne vedremo uno più interessante che considererà anche l'**input del mouse** per realizzare animazioni interattive, in base ai comandi dati dall'utente.

Nella scena **Unity** che sto utilizzando per questo esempio sono presenti solo la **telecamera**, una fonte di **luce** direzionale e il classico **cubo**, oggetto banale ma molto comodo per dei semplici esperimenti come questi.

Creiamo uno script **JavaScript** con un nome a piacere, quindi **assegniamolo al cubo** mediante trascinamento dal **Project** alla **Gerarchia** ed infine apriamolo in **MonoDevelop** per iniziare a scrivere il codice.



In particolare, **fuori da qualsiasi funzione** definiamo i tre parametri che ci servono, ossia due **Quaternion** e un valore **float** che, qui, porremo a 0.0, inizio del range temporale mappato sull'interpolazione dall'angolo iniziale a quello finale; scriviamo quindi:

```
var inizio : Quaternion;  
var destinazione : Quaternion;  
var tempo : float = 0.0;
```

I due **Quaternion** non sono inizializzati; per prima cosa, quindi, in **Start** definiamo due **Quaternion** “standard” (prendono quattro valori, coordinate x,y,z e w; noi useremo valori semplici), poi **sovrascriveremo** gli orientamenti mediante la funzione **SetEulerAngles**, che ci consente di esprimere gli orientamenti per gli assi X, Y e Z in una maniera decisamente più comoda.

Scriviamo quindi, all'interno della funzione **Start**:

```
inizio = new Quaternion(0.0, 0.0, 0.0, 1.0);  
destinazione = new Quaternion(0.0, 0.0, 0.0, 1.0);  
inizio.SetEulerAngles(0.0, 0.0, 0.0);  
destinazione.SetEulerAngles(0.0, 1.0, 0.0);
```

In realtà, questa scrittura non è così comoda, visto che **SetEulerAngles** considera gli angoli per gli assi X, Y e Z in **radianti**, non in gradi, per cui quell'1.0 come angolo di destinazione per Y non indica un grado ma un radiante, che vale poco più di 57°... comunque, sempre meglio di dover pensare in **quaternioni**.

Adesso, all'interno di **Update** (invocata, come sappiamo, ad ogni frame), scriviamo l'istruzione che, frame per frame, valuterà l'orientamento in base all'inizio, alla destinazione e al tempo, nel range 0.0 – 1.0, quindi assegnerà questa rotazione (definita come **Quaternion**) a **transform.rotation**, ruotando l'oggetto.

```
transform.rotation = Quaternion.Slerp(inizio, destinazione, tempo);
```

Chiaramente, mettendo questa istruzione in **Update** così com'è, valuteremo SEMPRE lo stesso angolo, ossia quello iniziale, perché tempo vale 0.0 e se non lo cambiamo questo vorrà dire restare fermi al punto 0.0 dell'interpolazione, cioè all'inizio; per questo motivo, in **Update** scriviamo anche un'istruzione di **incremento**:

```
tempo += 0.01;
```

In questo modo, ad ogni frame incrementeremo di 0.01 il valore di tempo, quindi la funzione **Slerp** valuterà **l'interpolazione tra gli angoli** in momenti diversi e, di conseguenza, assegnerà orientamenti diversi all'oggetto, generando di fatto **una rotazione animata**.

La durata dell'animazione dipende quindi dal valore dell'incremento e soprattutto dal frame rate del sistema, in quanto **Update** viene chiamata, incrementando tempo, ad ogni frame di gioco.

Per provare quanto detto, **salviamo lo script** per applicare le modifiche, torniamo nella schermata principale di **Unity** e premiamo **Play** per osservare il risultato.

La soluzione adottata non è molto elegante; ad esempio, aprendo l'*Inspector* durante l'esecuzione, notiamo che la variabile **tempo** continua ad **aumentare all'infinito**, anche se non ha più effetto sull'oggetto perché oltre il valore 1.0.

Un altro problema è dato **dall'incremento di un valore fisso frame per frame**, cosa che rende questo esempio dipendente dal frame rate del sistema in uso. Provate a risolvere questo punto ripassando quanto visto nel **tutorial su *Time.deltaTime***.

Soluzioni più interessanti riguardano ad esempio **l'utilizzo di funzioni esterne**, lanciate magari mediante **coroutine** e terminate non appena l'animazione arriva a 1.0, oppure definite con altri trucchetti come l'utilizzo di ***Time.time*** o di ***Time.deltaTime***, nell'esempio che stiamo per vedere (che riguarda anche l'input del mouse).

* * *

Eccoci alla seconda parte di questo tutorial su **Quaternioni** e rotazioni animate con ***Slerp***.

In questa seconda parte vedremo come ruotare il cubo in base all'input del mouse e come riportarlo all'orientamento iniziale, quando il mouse è fermo, mediante ***Quaternion.Slerp***.

In questo particolare caso, anziché scrivere il codice riga per riga e commentarlo, lo scriverò tutto in un colpo perché, come vedrete, così sarà più facile capire cosa succede.

Sostituiamo quindi lo script creato un attimo fa con queste nuove istruzioni:

```
#pragma strict
var rotazioneZ : float;
var rotazioneX : float;
var target : Quaternion;

function Start () {
}

function Update () {
    rotazioneZ = Input.GetAxis("Mouse X");
    rotazioneX = Input.GetAxis("Mouse Y");
    target = Quaternion.Euler(rotazioneX, 0, rotazioneZ);
    transform.rotation = Quaternion.Slerp(transform.rotation, target,
    Time.deltaTime);
}
```

Come potete notare, le uniche tre variabili che dichiariamo sono due *float*, che esprimono due angoli di rotazione o orientamento su Z e X, e un **Quaternion**, che come suggerisce il nome indica l'angolo “da raggiungere”. Non ci sono inizializzazioni in **Start**.

All'interno di **Update**, quindi ad ogni frame di gioco, **ricaviamo eventuali movimenti del mouse** e li utilizziamo come angoli di rotazione, ed ecco il senso delle prime due istruzioni:

```
rotazioneZ = Input.GetAxis("Mouse X");  
rotazioneX = Input.GetAxis("Mouse Y");
```

per cui la terza istruzione serve a costruire il **Quaternione** partendo da quei dati (assegnando angolo 0 alla coordinata Y), con:

```
target = Quaternion.Euler(rotazioneX, 0, rotazioneZ);
```

L'istruzione più interessante è comunque l'ultima:

```
transform.rotation = Quaternion.Slerp(transform.rotation, target,  
Time.deltaTime);
```

per cui qui l'orientamento iniziale è quello attuale dell'oggetto, quello finale è il *target* dato dal movimento del mouse e il momento di valutazione dell'interpolazione è dato dal tempo impiegato dal frame precedente (un modo per rendere il tutto indipendente dal frame rate).

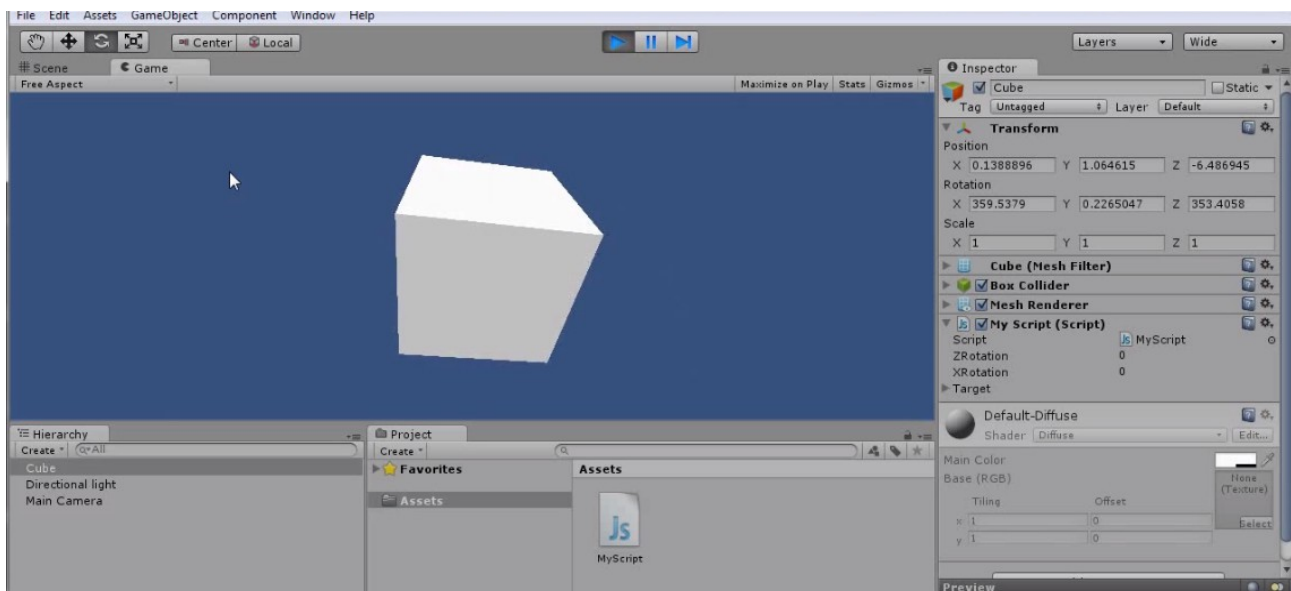
Il punto, comunque, è che **questo risultato viene assegnato a *transform.rotation***, per cui **al frame successivo** il *transform.rotation* primo parametro di **Quaternion.Slerp** sarà quello appena calcolato e da lì si calcolerà, in base al tempo trascorso (*Time.deltaTime*) il nuovo orientamento per arrivare al *target*, poi di nuovo si assegnerà questo valore a *transform.rotation* e si passerà al frame successivo, e così via fino a quando *transform.rotation* e *target* non coincideranno... in pratica, **l'oggetto sta “inseguendo se stesso”**, per così dire, variando la velocità di rotazione in base al *Time.deltaTime*.

NOTA: avviando ora il gioco, **le rotazioni probabilmente saranno troppo piccole** per essere apprezzate, per cui riscriviamo le prime due istruzioni di **Update** aggiungendo, alla fine di ciascuna istruzione prima del punto e virgola, un valore a moltiplicare, ad esempio * **50.0** o qualcosa del genere, in modo da “**esagerare**”, in un certo senso, **l'angolo di destinazione**.

A questo punto salviamo lo script per applicare le modifiche, torniamo alla schermata principale del programma e premiamo **Play**, muovendo il mouse per osservare il risultato.

Perché, quando il mouse si ferma, **il cubo torna nella posizione** (o, meglio, **nell'orientamento**) **iniziale**?

Per capirlo, diamo un'occhiata all'Inspector durante la fase di gioco: come potete vedere, quando non muovo il mouse i valori **RotazioneX** e **RotazioneY** vengono riportati a 0.0, per cui in quel momento il target nell'algoritmo è l'orientamento iniziale (0.0 per tutti gli assi) e **Quaternion.Slerp** ruota l'oggetto, nel tempo, per farlo arrivare a quel **target**. Ecco spiegato questo “ritorno”.



Questo tutorial è stato forse uno dei più pesanti, ma d'altronde **la rotazione nel 3D** è un argomento abbastanza complesso e padroneggiare gli strumenti che la riguardano è un bel traguardo; spero, quindi, di avervi presentato **Quaternion** e **Slerp** in maniera chiara, in modo da poter scrivere i vostri script o comunque leggere e comprendere quelli degli altri senza troppe difficoltà.

* * *

Algoritmo (e script JS) per mescolare le carte – I parte

Questo è il primo di due tutorial, nati da richieste di alcuni utenti del Forum *effe12.com*, sull'**algoritmo per mescolare un mazzo di carte** in **Unity 3D**.

In realtà, **l'algoritmo vero e proprio** lo vedremo già in questo primo capitolo, ma vedremo SOLO quello, nel senso che **a video non ci sarà nessun oggetto** e vedremo solo delle scritte nel **Debug Log**, che ci mostreranno l'esecuzione della procedura; nel prossimo capitolo, invece, vedremo come creare **un unico Prefab per le 52 carte** e disporre tali carte nella scena, ciascuna con una **Texture** immagine associata al suo valore, via codice, per cui poi rimescolando le carte vedremo davvero gli oggetti rimescolati nella scena 3D (cambiando, in realtà, solo le **Textures**).

Non vedremo direttamente l'effetto finale in un unico tutorial per due motivi: il primo è che magari vi interessa solo l'algoritmo per mescolare gli elementi in maniera casuale (e questo richiede molto poco tempo, quindi possiamo sbrigarlo subito); il secondo motivo è che per associare le **Textures** immagini alle variabili farò uso di nuovi elementi, come la struttura dati **Dizionario**, per cui per evitare di mettere troppa carne al fuoco preferisco spezzare la trattazione in due tutorial.

Attenzione, l'algoritmo che sto mostrando qui non l'ho preso da libri o altri siti ma è il primo che mi è venuto in mente per risolvere il problema di mescolare un mazzo di carte in maniera casuale, per cui sicuramente troverete in giro altri algoritmi o funzioni, magari migliori di questo, quindi metto le mani avanti dicendo che questo è solo uno dei tanti modi per affrontare il problema!

Altra piccola premessa: per forza di cose, all'inizio vi dirò di scrivere certe istruzioni “al buio”, ma in seguito riprenderemo tutto e, quando avrete il quadro completo, vi sarà tutto chiaro.

Fatte tutte le doverose premesse, possiamo finalmente iniziare.

Visto che in questa scena non mostreremo nulla a video, ci basta la semplice **Main Camera**, presente di default in ogni progetto vuoto, come questo.

Creiamo uno script **JavaScript**, chiamato ad esempio *CardShuffle*, quindi associamolo mediante trascinamento **alla Main Camera** e apriamolo in **MonoDevelop**, per editarlo.

Associandolo alla **Main Camera**, lo avvieremo (ossia, avvieremo il metodo **Start** di tale script, poi a seguire **Update**) sin dall'inizio del gioco.

Prima di iniziare a scrivere, ragioniamo su qual è il problema e come vogliamo affrontarlo.

Diciamo di avere un set di 52 elementi, ossia un vettore o **ARRAY** di 52 oggetti, e di volerli mescolare in maniera casuale.

Questi elementi possono essere di qualsiasi tipo: interi, stringhe o oggetti di altre classi, magari create ad hoc, ma per comodità decidiamo di fare un **array di stringhe** con dei “codici” particolari.

I semi delle carte francesi, quelle da poker per intenderci, sono: Cuori, Quadri, Fiori, Picche; elencati in questo modo – anche questo importante, nel secondo tutorial – possono essere ricordati come **CQFP** secondo la frase “Come Quando Fuori Piove”.

Decidiamo quindi di creare **un array di 52 stringhe** secondo questo codice: C1 (asso di cuori), C2, C3 e così via fino a C11, C12 e C13, ossia Jack, Donna e Re di cuori; a seguire, da **Come Quando Fuori Piove**, avremo Q1, Q2, eccetera, fino a P13 finale..

Qui viene il bello: **non creeremo queste variabili una per una**, neanche per sogno!

Le creeremo, invece, mediante **4 cicli for**; in seguito, potremo anche recuperarle altrettanto velocemente e, in generale, questo approccio consente di “scalare” l'algoritmo, adattandolo a tipi di mazzi, semi o quantità di carte molto diverse... **l'algoritmo è abbastanza “astratto”**, in un certo senso, per adattarlo a varie esigenze (cosa molto importante in programmazione).

Parliamo ora dell'algoritmo per mescolare le carte: l'approccio che utilizzeremo è quello di prendere **due oggetti ArrayList**, ossia **liste linkate** (vedremo tra un attimo cosa sono), il primo con tutte le carte ordinate come detto prima, il secondo con le carte mescolate (che sarà, quindi, **il mazzo finale**).

Una **Lista** è, come suggerisce il nome, **un insieme di elementi**, collegati uno dietro l'altro (**Linked**, appunto); a differenza di un **Array**, può essere modificata durante l'esecuzione, anche rimuovendo oggetti intermedi: in questo caso, l'oggetto precedente all'elemento **X** eliminato si collegherà direttamente a **X+1**, saltando **X**, che verrà tolto dalla lista.

Ecco quindi l'idea: facciamo **un ciclo for** con **52 iterazioni** dove, a ogni iterazione, generiamo un **numero casuale** (compreso tra 0 e 52 meno il numero di iterazioni, perché ad ogni ciclo la quantità di carte tra le quali cercare diminuirà), quindi **usiamo questo numero come indice** per prelevare un oggetto (quindi, un codice-carta) **dall'ArrayList originale** e metterlo in quello nuovo, inizialmente vuoto, che rappresenta il mazzo finale, mescolato, quindi alla fine restituiamo quest'ultimo oggetto.

Procediamo quindi con l'implementazione di quanto detto.

La prima cosa da fare, **fuori da qualsiasi funzione**, è creare **due** variabili di tipo **ArrayList**: la prima per il mazzo “già ordinato” (che andrà inizializzato, per prima cosa, in **Start**), il secondo per il mazzo “mescolato”, da restituire in output.

Scriviamo quindi:

```
var mazzoOriginale : ArrayList = new ArrayList();  
var mazzoMescolato : ArrayList = new ArrayList();
```

Da notare che non stiamo specificando **la quantità di elementi**: **ArrayList** dispone di un metodo, **Add**, che semplicemente aggiunge l'elemento passato come parametro in coda alla lista, indipendentemente dalla lunghezza o dalla quantità di elementi già presenti, per cui, come detto, è buono per vari casi, non solo per il mazzo di 52 carte.

Abbiamo detto che ci serve, in particolare, **una funzione di inizializzazione del mazzo originale**, con i nomi-codici delle carte, quindi una funzione di *shuffling* (il mescolamento delle carte) per

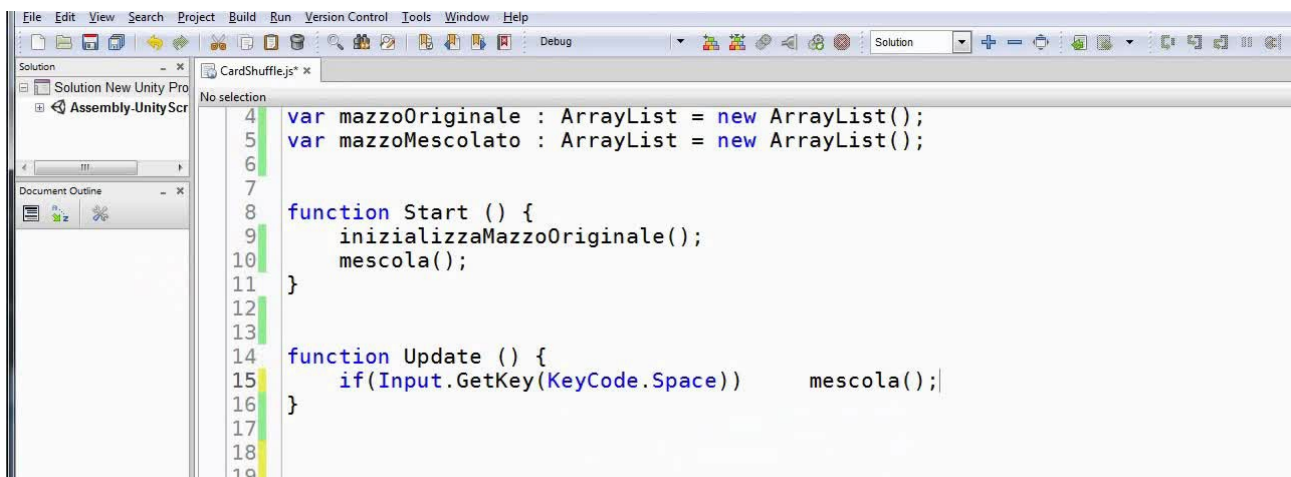
creare il mazzo mescolato; in **Start**, quindi, non scriviamo il codice di questi algoritmi, ma per fare le cose con ordine scriviamo solo due chiamate a funzioni, che definiremo subito dopo:

```
inizializzaMazzoOriginale();  
mescola();
```

Prima di definire queste funzioni, in **Update** scriviamo semplicemente:

```
if (Input.GetKey(KeyCode.Space)) mescola();
```

per cui, quando premeremo la **barra spazio** durante il gioco, **mescoleremo** nuovamente il mazzo, giusto per provare varie volte l'algoritmo.



Adesso, fuori da qualsiasi funzione, creiamo i blocchi (ancora vuoti, in questo momento) delle due funzioni “*inizializzaMazzoOriginale*” e “*mescola*”, con:

```
function inizializzaMazzoOriginale() {  
}  
  
function mescola() {  
}
```

quindi ora ci dedicheremo alle istruzioni da mettere nei corpi di queste funzioni.

Iniziamo dal corpo di *inizializzaMazzoOriginale()*: per il momento scrivo direttamente il codice, poi lo commenterò con voi che, vedendolo tutto in un colpo, capirete subito come funziona.

Ecco le istruzioni da inserire all'interno del blocco di *inizializzaMazzoOriginale()*:

```
for(var C : int = 0; C<13; C++) mazzoOriginale.Add("C" +
(C+1).ToString());
for(var Q : int = 0; Q<13; Q++) mazzoOriginale.Add("Q" +
(Q+1).ToString());
for(var F : int = 0; F<13; F++) mazzoOriginale.Add("F" +
(F+1).ToString());
for(var P : int = 0; P<13; P++) mazzoOriginale.Add("P" +
(P+1).ToString());
```

Le istruzioni sembrano strane perché negli **Array** e negli **ArrayList** il primo elemento ha indice 0, non 1; abbiamo, in particolare, che per ogni for aggiungiamo 13 elementi (for da 0 a minore di 13), di tipo **Stringhe**, definite mediante la lettera che identifica il seme e, concatenato come stringa, un numero, creato aggiungendo 1 all'indice del ciclo for (che parte da 0 e arriva a 12), per cui le stringhe saranno C1, C2, ... fino a C13, poi Q1, Q2 e così via; alla fine, avremo quindi $4 \cdot 13 = 52$ **oggetti stringa aggiunti**, uno per uno, in coda all'**ArrayList**, che quindi è una **lista linkata** di 52 nodi di tipo **String**.

Questa è l'inizializzazione di *mazzoOriginale*.

Se volete vedere, in Debug Log, la forma stringata di questa lista, scrivete nella funzione:

```
var out : String = "";
for (var i : int = 0; i<52; i++) out = out + " " + mazzoOriginale[i];
Debug.Log(out);
```

salvate lo script per apportare le modifiche e avviate il gioco: nella zona del **Debug Log** (in basso, nella schermata dell'Editor di **Unity**) vedrete il contenuto della stringa out, definita **concatenando le stringhe-nodi dell'ArrayList mazzoOriginale**.

Passiamo quindi all'**algoritmo di mescolamento** vero e proprio, all'interno del blocco di *mescola()*.

Qui, prima di procedere dobbiamo scrivere due istruzioni preliminari che nascono... dall'esperienza.

La prima istruzione è:

```
mazzoMescolato = new ArrayList();
```

ossia, dobbiamo ricreare **l'array delle carte mescolate**, che tecnicamente è vuoto, per renderlo nuovamente vuoto... perché?

Perché tra poco gli **aggiungeremo in coda le carte** prese dal mazzo originale e, se non lo **svuoteremo**, la prossima volta che mescoleremo le carte, chiamando questo algoritmo, gli aggiungeremo in coda le nuove carte, quindi avremo array di 52, 104, 156, 208 carte e così via, e tra l'altro le nuove carte saranno sempre sul fondo, mai nei primi 52 posti, per cui avremo pure l'impressione che qualcosa non vada nell'algoritmo, perché vedremo sempre le prime 52! Al primo avvio, quindi, questa istruzione è superflua, ma è indispensabile per le chiamate successive.

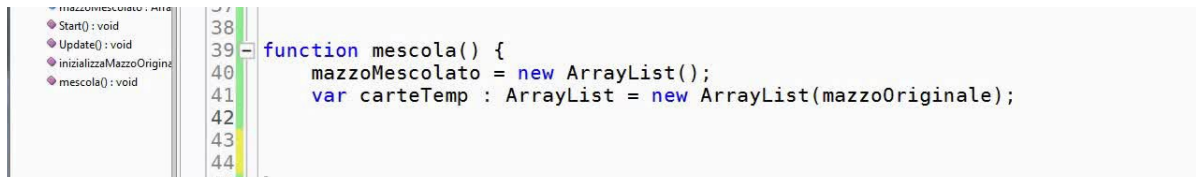
La seconda istruzione è

```
var carteTemp : ArrayList = new ArrayList(mazzoOriginale);
```

che serve a creare una copia temporanea dell'array originale, con le carte ordinate, dal quale rimuovere gli elementi presi casualmente, in modo da **non modificare l'array originale**, e qui la cosa da notare è l'utilizzo di *new ArrayList(mazzoOriginale)* per creare la **copia** dell'array anziché l'utilizzo dell'uguale, infatti scrivere:

```
var carteTemp : ArrayList = mazzoOriginale;
```

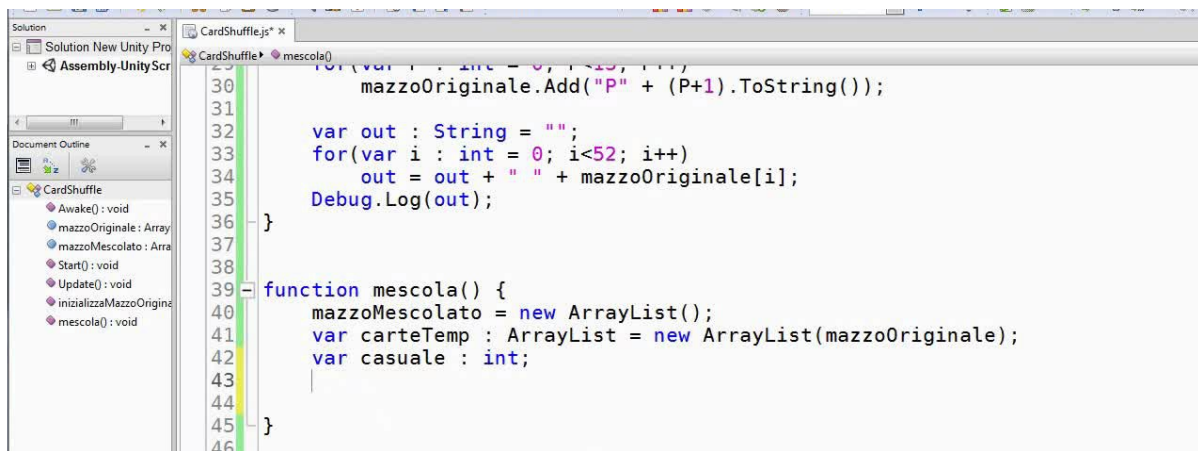
assegnerebbe a *carteTemp* le stesse locazioni di memoria di *mazzoOriginale*, ossia farebbe una “**copia riferimento**”, puntando alle locazioni di memoria dell'array *mazzoOriginale*, non una “**copia valore**”: un errore subdolo, che alla seconda iterazione di *mescola* ci porterebbe ad un errore a runtime (durante l'esecuzione). Lo risolviamo mediante *new ArrayList* passando, come parametro al costruttore, **l'array da copiare**.



```
38  
39 = function mescola() {  
40     mazzoMescolato = new ArrayList();  
41     var carteTemp : ArrayList = new ArrayList(mazzoOriginale);  
42  
43  
44
```

Prima del ciclo **for** di estrazione del **valore casuale** e suo inserimento nel **nuovo mazzo**, dichiariamo una variabile intera per il **numero casuale**, con:

```
var casuale : int;
```



```
30  
31     mazzoOriginale.Add("P" + (P+1).ToString());  
32  
33     var out : String = "";  
34     for(var i : int = 0; i<52; i++)  
35         out = out + " " + mazzoOriginale[i];  
36     Debug.Log(out);  
37  
38  
39 = function mescola() {  
40     mazzoMescolato = new ArrayList();  
41     var carteTemp : ArrayList = new ArrayList(mazzoOriginale);  
42     var casuale : int;  
43  
44  
45  
46
```

Adesso, il ciclo **for**: come detto, ci servono **52 iterazioni** e, ad ogni iterazione, dovremo generare un numero casuale compreso tra 0 e (52 – il numero di iterazioni fatte), quindi dovremo rimuovere l'elemento con questo indice dal **mazzoOriginale** (che farà “scalare” automaticamente le altre voci, in quanto lista linkata) e inserirlo in coda al **mazzoMescolato**.

Per generare un numero tra 0 e (52 - le iterazioni), faremo andare il ciclo **for** a passo di gambero, ossia all'indietro: inizieremo da 52 e arriveremo a 0, decrementando il valore dell'indice ad ogni iterazione.

Scriviamo quindi (sempre all'interno della **funzione mescola()**, dopo la dichiarazione di **casuale**):

```
for(var i: int = 52; i>0; i--) {  
    casuale = Random.Range(0,i); // Genera l'intero casuale tra 0 e i  
    mazzoMescolato.Add(carteTemp[casuale]);  
    carteTemp.removeAt(casuale);  
}
```

Attenzione all'utilizzo delle **parentesi quadre** per l'*Add*, perché accediamo a un elemento dell'array come suo **campo**, e a quello delle **parentesi tonde** di *removeAt*, che invece è una **funzione**, quindi lì passiamo un parametro.

Bene, **il mazzo è mescolato!**

Se volete una prova di quanto detto, potete scrivere, sempre dentro *mescola()* dopo il ciclo *for*, il blocco di codice del *Debug* scritto in *inizializzaMazzoOriginale*, sostituendo ovviamente *mazzoOriginale* con *mazzoMescolato*, salvare lo script e avviare il gioco.

Premendo la **barra spazio durante l'esecuzione**, **rimescolerete** le carte, visto che tale evento, come definito all'inizio in *Update*, richiamerà la funzione *mescola()*.

* * *

Algoritmo (e script JS) per mescolare le carte – II parte

Eccoci alla seconda e ultima parte del **tutorial sull'algoritmo per mescolare le carte in Unity e Javascript**.

In realtà, abbiamo visto l'**algoritmo vero e proprio** nel capitolo precedente, mostrando i codici delle carte mescolate nella finestra **Debug Log**; in questo tutorial, invece, disporremo dei **Planes** con le **immagini delle carte** associate a tali codici e, cosa molto più importante, parleremo della struttura dati nativa **Dictionaries**, i dizionari... prima, però, un piccolo ripasso dello script così come l'abbiamo lasciato nel capitolo precedente.

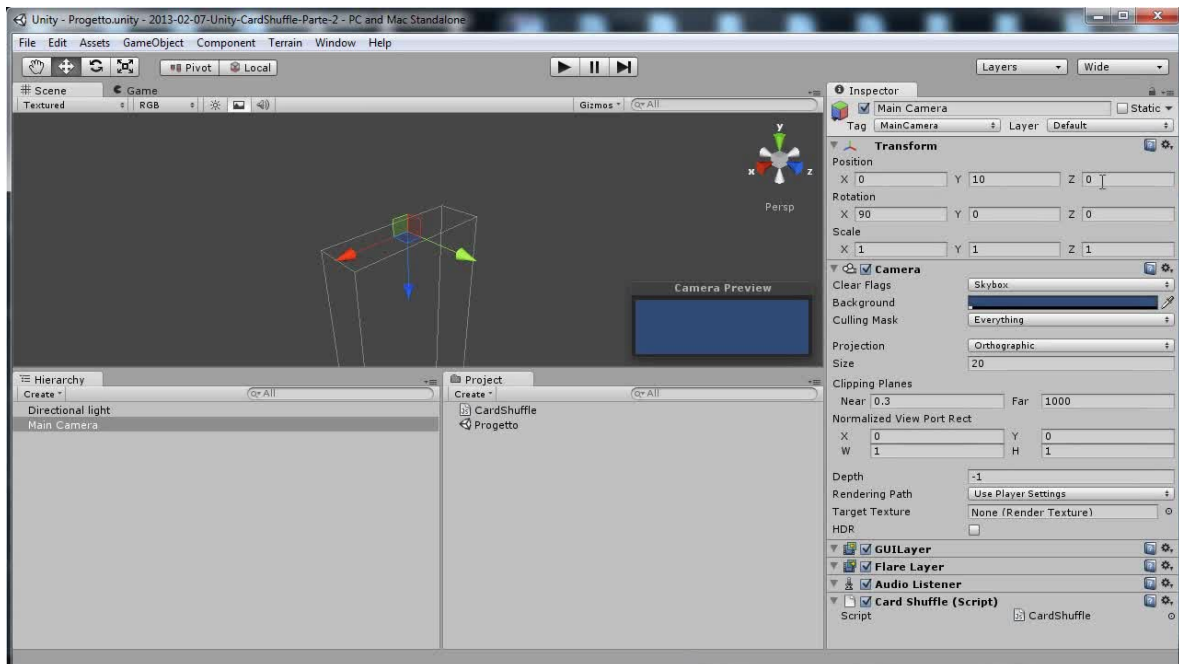
Avevamo, in particolare, due variabili pubbliche, **mazzoOriginale** (da inizializzare in **inizializzaMazzoOriginale**, chiamato da **Start**, e da mantenere inalterato) e **mazzoMescolato**, ricreato ad ogni chiamata della funzione **mescola**, dove si faceva uso di un array temporaneo, **carteTemp**, con copia valore di **mazzoOriginale**; infine, in **Start** le uniche due chiamate a funzioni erano a **inizializzaMazzoOriginale** e a **mescola**.

Prima di procedere con le modifiche, pensiamo un attimo a cosa vogliamo fare: far apparire dei **Planes** texturizzati associando, **a ciascun plane, un'immagine**, in base al codice della carta.

Dobbiamo quindi agire su due fronti: nell'Editor principale di **Unity** dobbiamo creare almeno il **Prefab della carta**, mentre nel codice dovremo creare **un array di Texture**, definire delle **funzioni** per disporre le istanze del **Prefab** nella scena e associare loro le immagini.

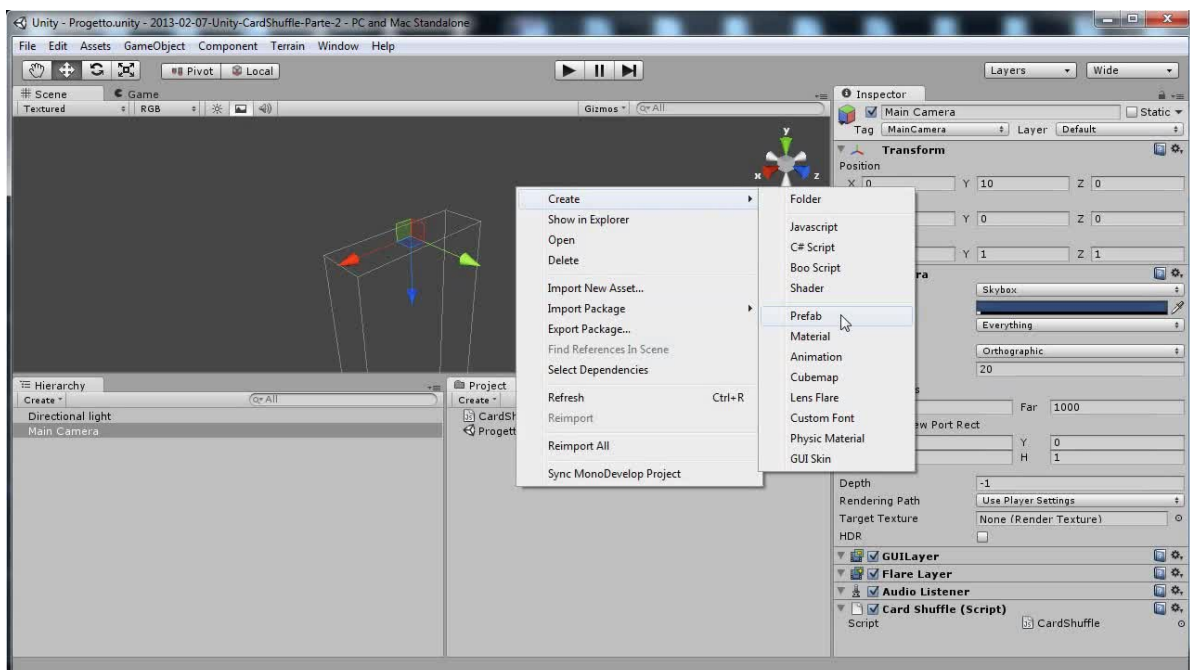
Iniziamo dalla scena **Unity**, nell'editor principale del programma.

Disponiamo la **Main Camera** in questo modo: Position X 0, Y 10, Z 0; Rotation X 90, Y 0, Z 0; modalità **Orthographic** con **Scale** 20.

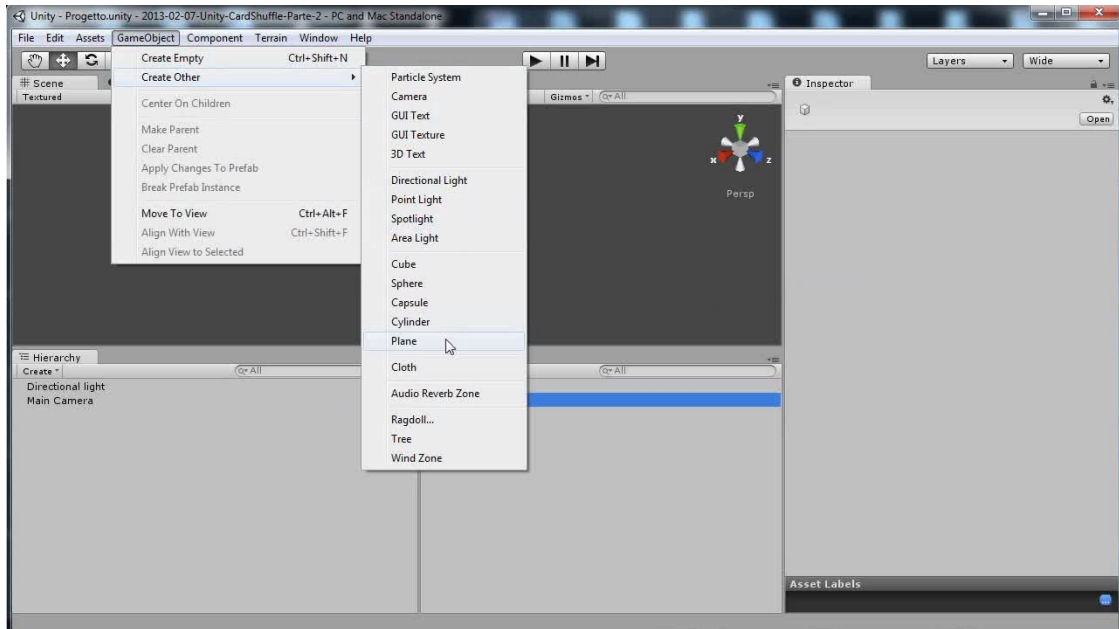


In questo modo, **inquadreremo la scena dall'alto** (altezza 10 unità) in modalità **ortografica**, senza effetti di prospettiva.

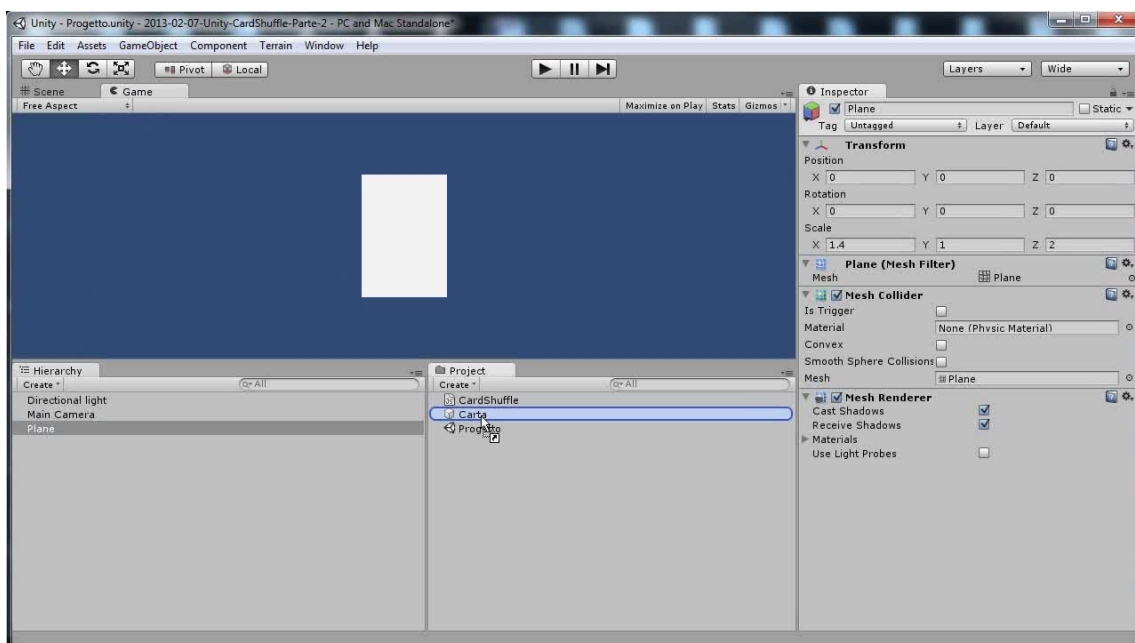
Ora creiamo un **Prefab** vuoto, con click destro nella scheda **Project** e **Create | Prefab**, chiamandolo ad esempio “**Carta**”, in quanto rappresenta il prototipo della carta dal quale otterremo poi le istanze per disporre gli oggetti nella scena.



Tale **Prefab** è, comunque, **vuoto**; per fornirgli almeno la geometria iniziale, facciamo così: inseriamo **un Plane vuoto nella scena**, con **Game Object | Create Other | Plane**, quindi posizioniamo tale oggetto in 0,0,0 (l'origine) e impostiamo, ad esempio, Scale X 1.4 e Scale Z 2 (magari le proporzioni non sono quelle reali, corrette... sto andando a occhio, qui).



Come potete vedere dalla schermata **Game**, abbiamo il **Plane** rettangolare di fronte alla telecamera. Facciamo click col tasto sinistro su tale oggetto nel pannello **Gerarchia** e, tenendolo premuto, trasciniamolo sul **Prefab “Carta”** nel **Project**, rilasciando per associare al **Prefab** tale geometria.



A questo punto il **Prefab Carta** ha la geometria del **Plane** e un componente **Renderer**, che gestirà **Materials** e **Textures**, presente di default, per cui possiamo cancellare l'oggetto **Plane** dalla **Gerarchia**, rimuovendolo dalla scena: creeremo e gestiremo le istanze via codice.

Torniamo quindi in **MonoDevelop**, con lo script **CardShuffle** pronto per essere modificato.

Dai tutorial precedenti su **Unity**, sappiamo che per creare delle **istanze da un Prefab via codice** ci serve un oggetto particolare di tipo **Transform**; visto che le carte da creare sono 52, definiremo un array di **Transform**, più una variabile **Transform** a parte per identificare il **Prefab Carta** (ok, potremmo creare un solo array di 53 elementi e mettere il **Prefab Carta** come primo elemento, con indice 0, ma per evitare di fare confusione creiamo due variabili distinte).

All'inizio dello script, **fuori da ogni funzione**, scriviamo quindi:

```
var planesCarte : Transform[] = new Transform[52];  
var PrefabCarta : Transform;
```

Sappiamo già che ci serviranno anche **52 oggetti Textures**, per identificare le immagini da associare alle varie **carte**, quindi scriviamo:

```
var arrayTextures : Texture[] = new Texture[52];
```

A questo punto, ci serve un modo per associare **i codici delle carte**, rappresentati come sappiamo con stringhe nella forma “**SEME-NUMERO**”, ad esempio C3, P2, F1, eccetera, alle varie **Textures**; nel capitolo precedente vi ho fatto inizializzare il mazzo con l'ordine **CQFP**, secondo “*come quando fuori piove*”, con i numeri da 1 a 13, quindi **l'associazione tra i codici e le Textures** seguirà quest'ordine... ma **come implementare** questa associazione?

Utilizzeremo una struttura dati nativa chiamata **Dizionario**.

Potete pensare a un dizionario come ad **un array a due dimensioni**, ossia un **vettore di coppie**, ma la cosa importante è che questo speciale array identifica, nella coppia, **un elemento “chiave”** e un **elemento “valore”**, per cui si tratta, in effetti, di **un array di coppie chiave-valore**.

Perché **la chiave** è importante? Perché deve essere **unica**, in tutto il **Dizionario**, per poter identificare e **recuperare il valore** associato.

Il valore, invece, può essere replicato, cioè chiavi diverse possono avere valori identici, ma non possono esserci due chiavi uguali nello stesso Dizionario.

Parliamo ora dei tipi di dati di chiavi e valore: il bello è che **possiamo usare dati di qualsiasi tipo**.

Nel nostro caso, quindi, **per le chiavi useremo delle Stringhe**, in quanto identificheremo le **Texture** con i codici stringa nella forma **SEME-NUMERO**, mentre per i **valori** useremo degli oggetti Texture... ovvero, **le Texture** definite un attimo fa nell'array **arrayTexture**!

Il **dizionario** va creato **specificando i tipi delle chiavi e dei valori**, per cui, sempre fuori da qualsiasi funzione (nello “spazio globale” dello script), definiamo il sesto e ultimo oggetto globale:

```
var dizionarioTextures : System.Collections.Generic.Dictionary.<String,
Texture> =
    new System.Collections.Generic.Dictionary.<String, Texture>();
```

Questa dichiarazione fa un po' paura ma in realtà stiamo specificando il tipo di dato **Dictionary** con la **forma estesa**, che prende tutta la catena gerarchica, perché **esistono altri tipi di Dictionary** nella libreria nativa, quindi dobbiamo dire di prendere proprio quello che deriva da **System, Collections, Generic**; l'altro punto che può generare confusione è la definizione dei tipi di dati presi dal dizionario mediante lo schema **Template**, quindi con **<String, Texture>**, ma ci si fa presto l'abitudine e comunque riguarda solo questa riga... tutto qui.

Adesso dobbiamo associare, in fase di inizializzazione, il codice della carta alla variabile **Texture** immagine relativa, quindi dobbiamo spostarci in **inizializzaMazzoOriginale** e modificarne il contenuto.

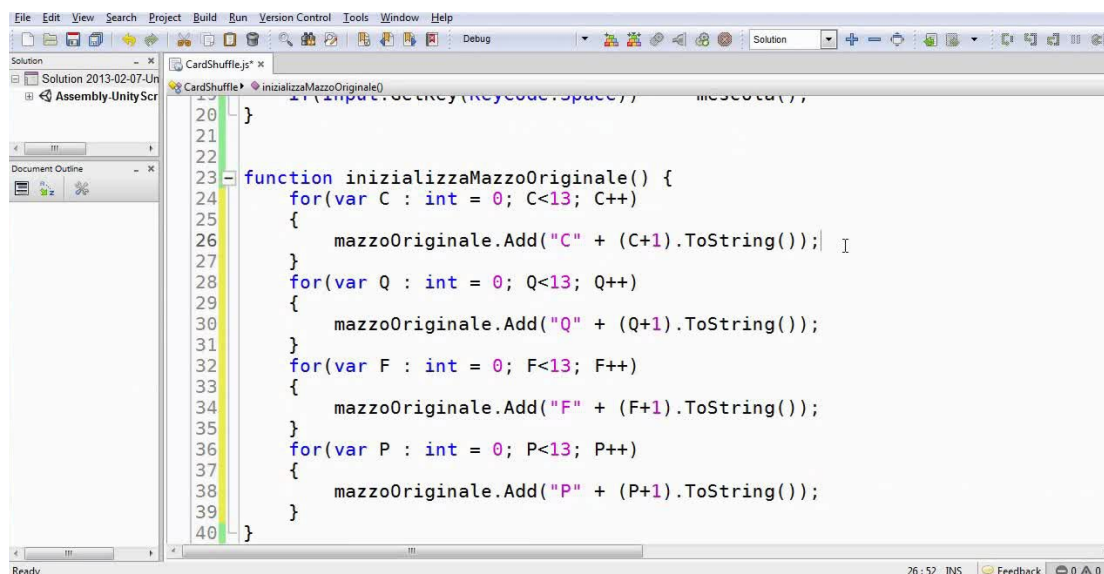
In particolare, per ogni ciclo **for** dobbiamo definire due istruzioni, non più solo una come nello script originale, quindi ad esempio **il primo ciclo for andrà cambiato da:**

```
for(var C : int = 0; C<13; C++) mazzoOriginale.Add("C" +
(C+1).ToString());
```

a:

```
for(var C : int = 0; C<13; C++){  
    mazzoOriginale.Add("C" + (C+1).ToString());  
    // Seconda riga vuota, per mettere la nuova istruzione  
}
```

e questa modifica va fatta anche **per gli altri tre cicli for**, per cui aggiungiamo il blocco di parentesi graffe con dentro l'istruzione di inizializzazione del codice stringa e una riga vuota, per la nuova istruzione.



Parliamo quindi di questa nuova istruzione: ad ogni iterazione, nel **Dizionario *dizionarioTextures*** aggiungeremo, proprio come con un **arrayList**, una nuova **coppia chiave-valore**, con la chiave costituita dalla **stringa-codice** della carta e, per il valore, l'elemento corrispondente nell'array di **Textures...** su questo punto c'è da fare una piccola riflessione.

I quattro cicli for, nell'ordine **CQFP**, partono ogni volta da zero e arrivano a 12 (incluso).

L'array di Textures, che segue sempre l'ordine CQFP, parte invece da 0 e arriva a 51 (incluso).

Nel ciclo for del C, quindi, i codici C possono essere mappati tranquillamente con gli indici da 0 a 12 del ciclo, ma già nel ciclo Q non possiamo fare la stessa cosa perché finiremmo col prendere nuovamente le prime 13 locazioni dell'array di Textures. **Ci serve, quindi, un offset**, uno scostamento di 13 passi ad ogni ciclo successivo.

Come **nuova istruzione del ciclo *for*** del codice **C** scriviamo:

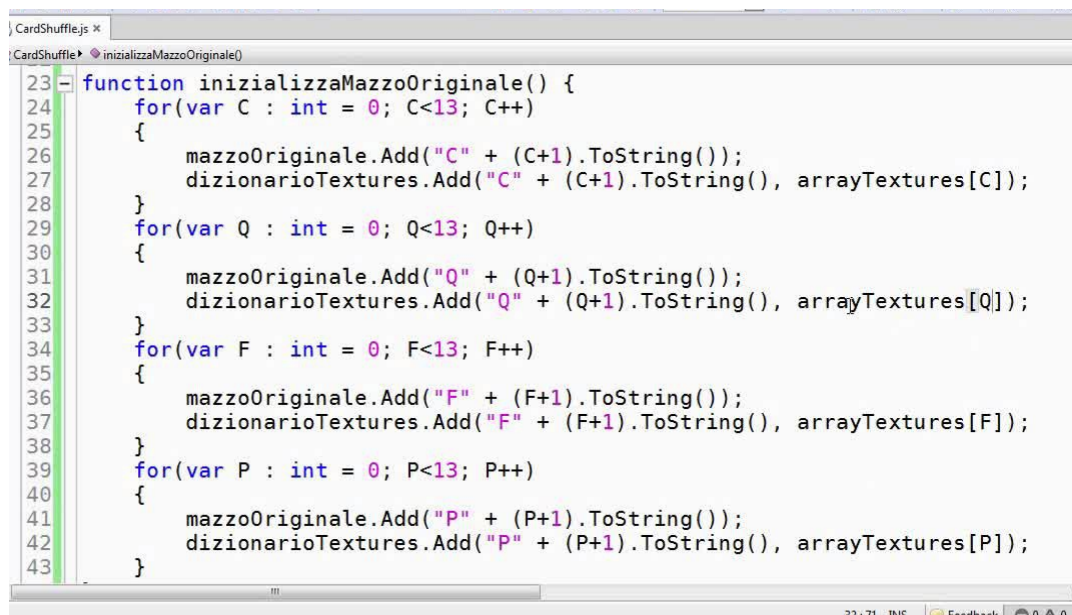
```
dizionarioTextures.Add("C" + (C+1).ToString(), arrayTextures[C]);
```

Come potete vedere, **il primo argomento** è identico a quello dell'istruzione precedente, in quanto si tratta di inizializzare lo stesso codice stringa per la carta.

Vi consiglio, a questo punto, di procedere così: copiate questa istruzione così com'è anche negli altri 3 cicli ***for***. Fatto questo, per ciascun ciclo ***for*** sostituite il primo argomento di ***dizionarioTextures.Add*** con l'argomento dell'***Add*** dell'istruzione sopra, in quanto, come visto, si tratta dello stesso codice stringa.

Avremo quindi, per gli altri tre cicli ***for***:

```
dizionarioTextures.Add("Q" + (Q+1).ToString(), arrayTextures[C]);  
dizionarioTextures.Add("F" + (F+1).ToString(), arrayTextures[C]);  
dizionarioTextures.Add("P" + (P+1).ToString(), arrayTextures[C]);
```

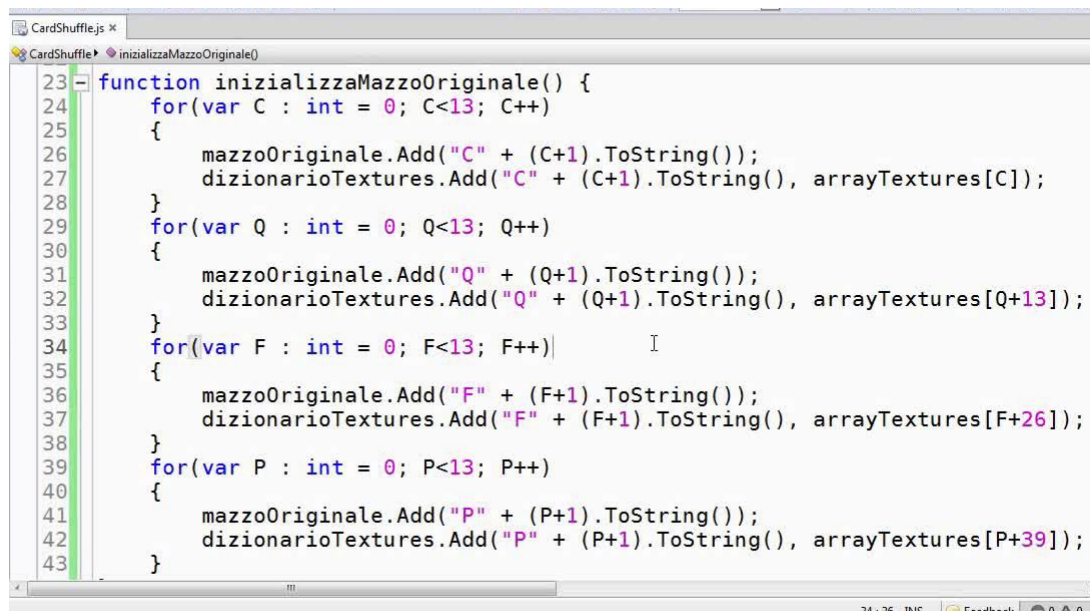


Passiamo quindi al secondo argomento di queste nuove istruzioni.

Come detto, si tratta di **aggiungere un offset** di valore 13 per ogni ***for*** successivo, in modo da **spostarci nell'array di Textures**.

Sostituiamo quindi le tre istruzioni in questo modo:

```
dizionarioTextures.Add("Q" + (Q+1).ToString(), arrayTextures[Q+13]);  
dizionarioTextures.Add("F" + (F+1).ToString(), arrayTextures[F+26]);  
dizionarioTextures.Add("P" + (P+1).ToString(), arrayTextures[P+39]);
```



Diamo un'occhiata a quello che succede.

Nel primo ciclo, definiamo le carte di codice C e inseriamo, nei primi 13 posti del dizionario, i codici stringa C e i primi 13 elementi dell'array di Textures.

Nel secondo ciclo, definiamo le carte di codice Q e inseriamo, nei successivi 13 posti del dizionario (perché *Add* aggiunge in coda, quindi partiremo proprio dall'indice 13), i codici stringa Q e i successivi 13 elementi dell'array di Textures, in quanto il ciclo Q riparte da 0 e arriva a 12, quindi aggiungendo 13 all'indice Q accederemo al 14esimo elemento dell'array di Textures.

Discorsi analoghi possono essere fatti per F e P, dove l'offset numerico vale 26 (cioè 13*2) e 39 (cioè 13*3)... in effetti, per rendere un po' più astratto questo codice, in modo da estenderlo a mazzi con numeri di carte differenti, potremmo pensare ad **una variabile offset** con il numero di carte per seme, nel nostro caso 13, e scrivere, anziché 13, 26 e 39, offset, offset*2 e offset*3 rispettivamente... ma per questo esempio lasciamo le cose così come sono.

Con queste modifiche, **la funzione *inizializzaMazzoOriginale* è pronta.**

Restano l'istruzione per disporre le istanze del **Prefab** nella scena, l'istruzione per **aggiornare le immagini** di tali istanze (e parlo di “istruzione” perché, in entrambi i casi, si tratta di un semplicissimo **ciclo for**) e un'operazione alquanto noiosa ma che, purtroppo, va fatta, ossia l'associazione delle immagini alle variabili nell'array, secondo lo schema CQFP, nell'editor di **Unity**.

* * *

Dobbiamo definire due nuove funzioni: la prima per disporre fisicamente le istanze del **Prefab** nella scena, la seconda per **aggiornare** le loro immagini.

In particolare, **le istanze vanno disposte solo una volta**, all'inizio del gioco, per cui all'interno della funzione **Start**, tra *inizializzaMazzoOriginale()* e *mescola()*, inseriamo una nuova istruzione:

```
disponiCarte();
```

ossia una chiamata a una nuova funzione, *disponiCarte*.

Quello che vogliamo è **disporre le istanze del Prefab di fronte alla telecamera**, quindi ad esempio dalla posizione X -50 in poi, con coordinate Y e Z 0 (ho fatto delle prove, per questo prima ho dato indicazioni precise su dimensioni carte, posizione, orientamento e *Orthographic Scale* della telecamera).

Scriviamo quindi (nello “spazio globale” dello script):

```
function disponiCarte() {  
    for(var i: int=0; i<52; i++) planesCarte[i] =  
        Instantiate(PrefabCarta, Vector3(-50+(i*5),0,0),  
        Quaternion.identity);  
}
```

Per ogni iterazione, quindi, definiamo un oggetto dell'array di **Transform** *planesCarte*, associandogli un'istanza di oggetto realizzata con tre parametri: il **Transform** *PrefabCarta*, che poi nell'Editor di **Unity** assoceremo all'oggetto **Prefab Carta** del **Project**; un **Vector** di **posizioni** che utilizza l'indice dell'elemento come offset di spostamento su X, per cui la prima carta sarà a X -50, la seconda -45, la terza -40 e così via, in quanto a ogni iterazione abbiamo *i*5*, quindi l'offset è di 5

unità su X; infine, come terzo parametro, un **Quaternion** di base, visto che *Instantiate* vuole un orientamento predefinito, per cui gli forniamo l'orientamento standard con *Quaternion.identity*.

```
rdShuffle.js* x
rdShuffle Awake()
52     var casuale : int;
53     for(var i: int = 52; i>0; i--) {
54         casuale = Random.Range(0,i); // Genera l'intero casuale tra 0 e i
55         mazzoMescolato.Add(carteTemp[casuale]);
56         carteTemp.RemoveAt(casuale);
57     }
58     var out : String = "";
59     for (var j : int = 0; j<52; j++) out = out + " " + mazzoMescolato[j];
60     Debug.Log(out);
61 }
62
63 function disponiCarte() {
64     for(var i: int=0; i<52; i++)
65         planesCarte[i] = Instantiate(
66             PrefabCarta,
67             Vector3(-50+(i*5), 0, 0),
68             Quaternion.identity);
69 }
```

La funzione *disponiCarte* viene quindi chiamata solo la prima volta e si occupa di disporre solo i *planes* nella scena, non gestisce l'aspetto e questi *planes* non vengono spostati quando si rimescolano le carte: a essere “spostate”, per così dire, sono solo le **Textures**.

Dobbiamo ricordare, infatti, che il **blocco dati dei planes**, con le geometrie definite nel **Prefab**, e il blocco dati delle **Textures**, sono distinti, separati, con delle associazioni tra un oggetto *plane* e un oggetto **Texture**, per cui, anziché spostare fisicamente i *planes*, possiamo cambiare le associazioni tra immagini **Textures** e *plane*, mantenendo ferme le geometrie e cambiando solo gli aspetti.

Per far ciò, ci serve una nuova funzione, chiamata *aggiornaImmagini*, da richiamare ogni volta che si mescolano le carte (e poiché le carte vengono mescolate già durante l'inizializzazione, verrà chiamata in automatico anche la prima volta).

Per prima cosa, quindi, all'interno di *mescola()*, come ultima istruzione prima della chiusura del blocco funzione, scriviamo:

```
aggiornaImmagini();
```

in modo da chiamare tale funzione ogni volta che il mazzo viene mescolato.

Fuori da qualsiasi funzione, in coda allo script, scriviamo quindi:

```
function aggiornaImmagini() {  
    for(var i:int=0; i<52;i++)  
        planesCarte[i].renderer.material.mainTexture =  
            dizionarioTextures[ (mazzoMescolato[i]).ToString() ];  
}
```

Questo ciclo **for** fa 52 iterazioni e, ad ogni iterazione, prende un elemento dell'array di oggetti (quindi delle istanze di carte create e disposte in precedenza) e gli cambia la Texture, accedendo a quell'informazione con **.renderer.material.mainTexture**.

La **Texture** da associare alla carta viene quindi **recuperata dal dizionario** utilizzando, tra parentesi quadre, **la chiave di ricerca**, che sappiamo essere univoca e rappresentata dal codice stringa...

... ma come facciamo a sapere qual è il codice della carta di posto **i**?

Semplice: accediamo all'oggetto di indice **i** dell'array di stringhe **mazzoMescolato**, quindi **mazzoMescolato[i].ToString()** ci restituisce il codice (stringa) della carta.

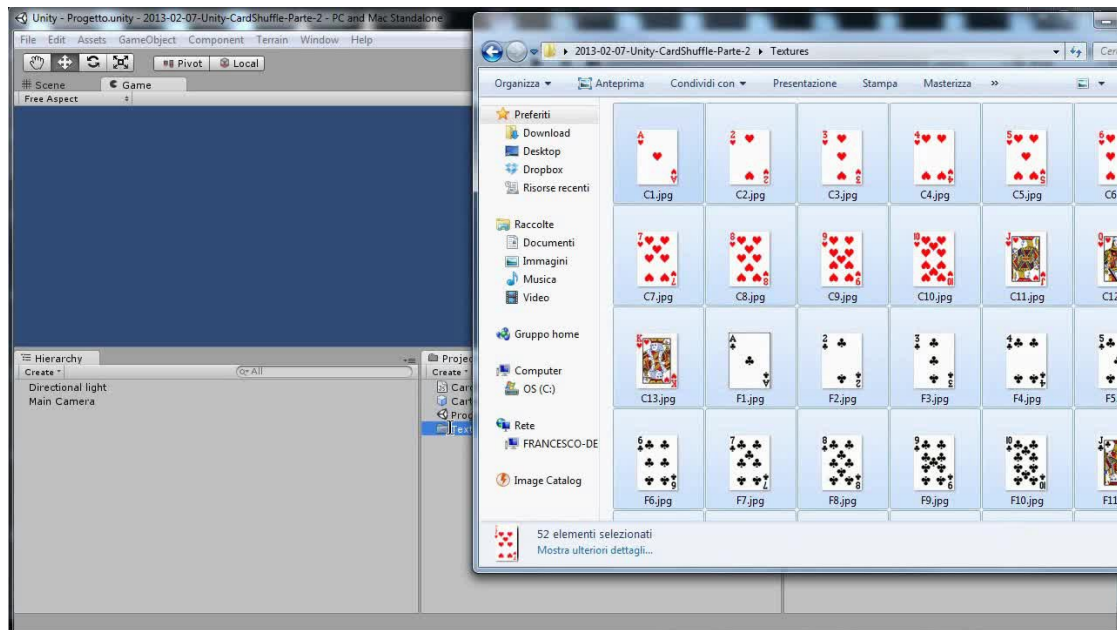
Rivedendo l'istruzione con queste componenti, abbiamo che prendiamo il codice della carta con **mazzoMescolato[i]**, quindi prendiamo la **Texture** dal dizionario usando questo codice carta come chiave e associamo la **Texture** così recuperata all'oggetto di posto **i** di **planesCarte**.

Con lo **script** abbiamo finito.

In realtà non è così complicato come può sembrare, abbiamo impiegato un bel po' di tempo per spiegarlo da zero ma rivedendolo con calma noterete che in effetti è abbastanza semplice e lineare; tuttavia, non abbiamo ancora finito col **progetto**, in quanto mancano degli elementi fondamentali: le associazioni tra le variabili dello script e gli oggetti di scena.

Per quanto riguarda **le immagini da usare come Textures**, dovreste procurarvi (cercando online) 52 immagini di carte, cambiandone i nomi per renderli nel formato C1, C2, C3, eccetera.

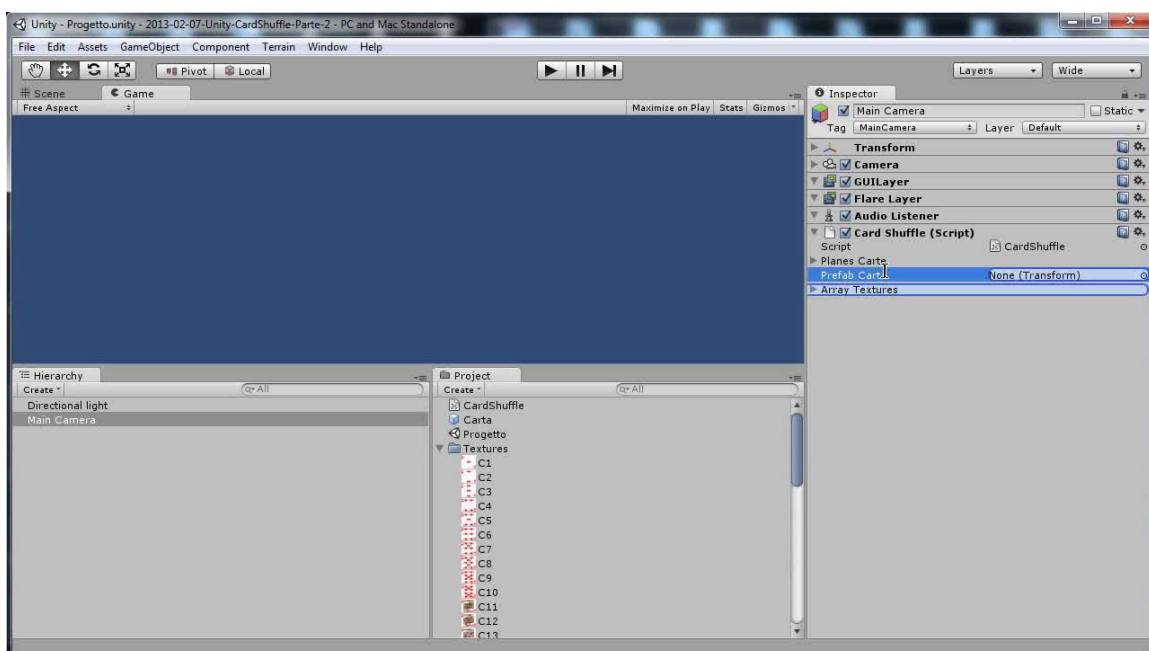
Create quindi una cartella “**Textures**” nel **Project** dell'editor principale di **Unity** e trascinate lì i file delle immagini presi dalla cartella su disco, in modo da **generare gli Assets** (immagini) delle carte.



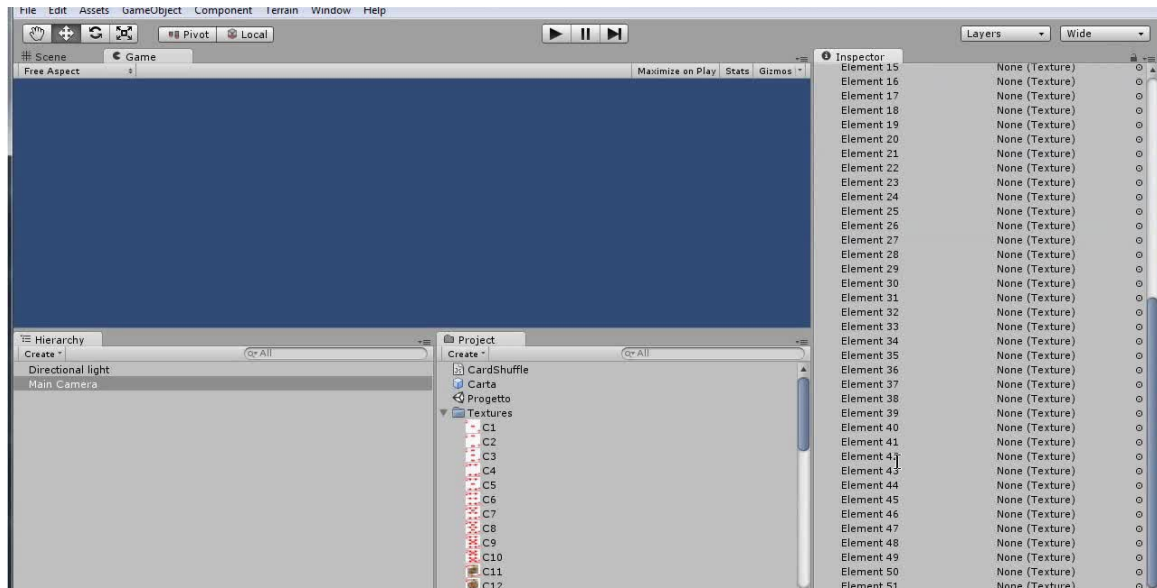
Fatto questo, selezionate l'oggetto **Main Camera**, provvisto dello **script CardShuffle**, e date un'occhiata al **componente script**, nell'Inspector.

Le variabili pubbliche da inizializzare nell'editor, mediante drag-and-drop come visto in molti tutorial precedenti, sono due: **PrefabCarta**, da associare al **Prefab** della carta, e **ArrayTextures**.

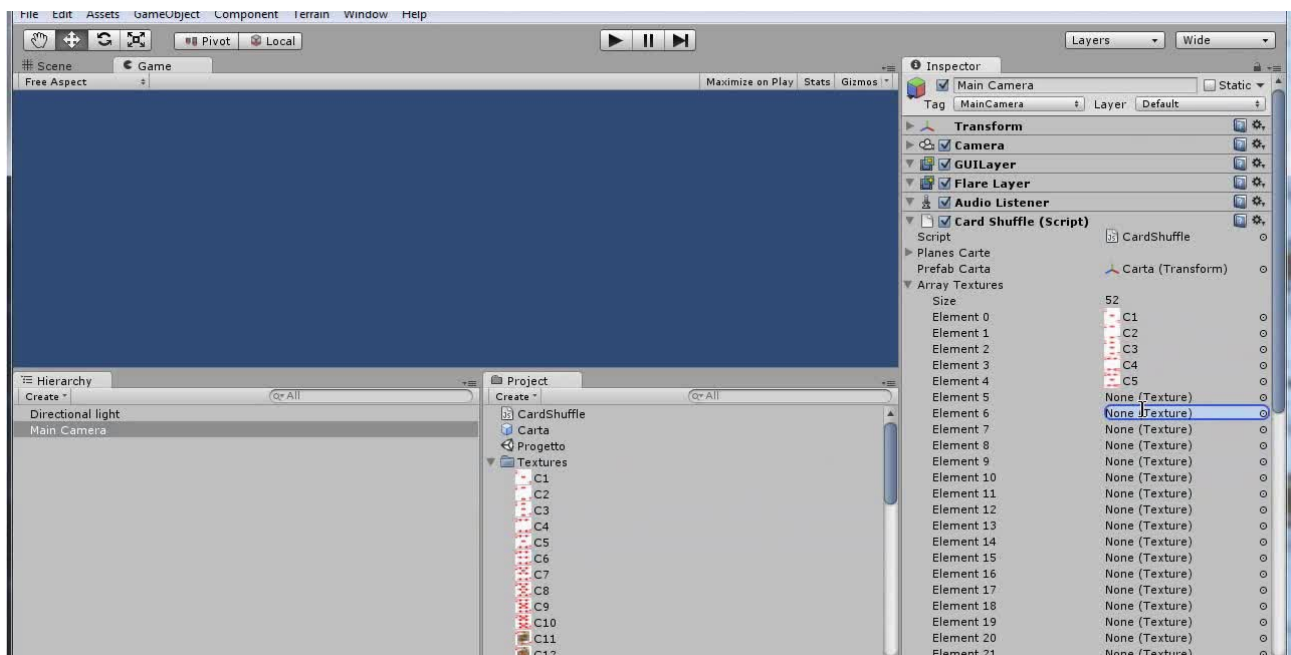
Facciamo quindi click **SENZA RILASCIARE** il mouse sul **Prefab “Carta”** nel **Project** e trasciniamolo nello slot **PrefabCarta** dello script **CardShuffle**, nell'Inspector.



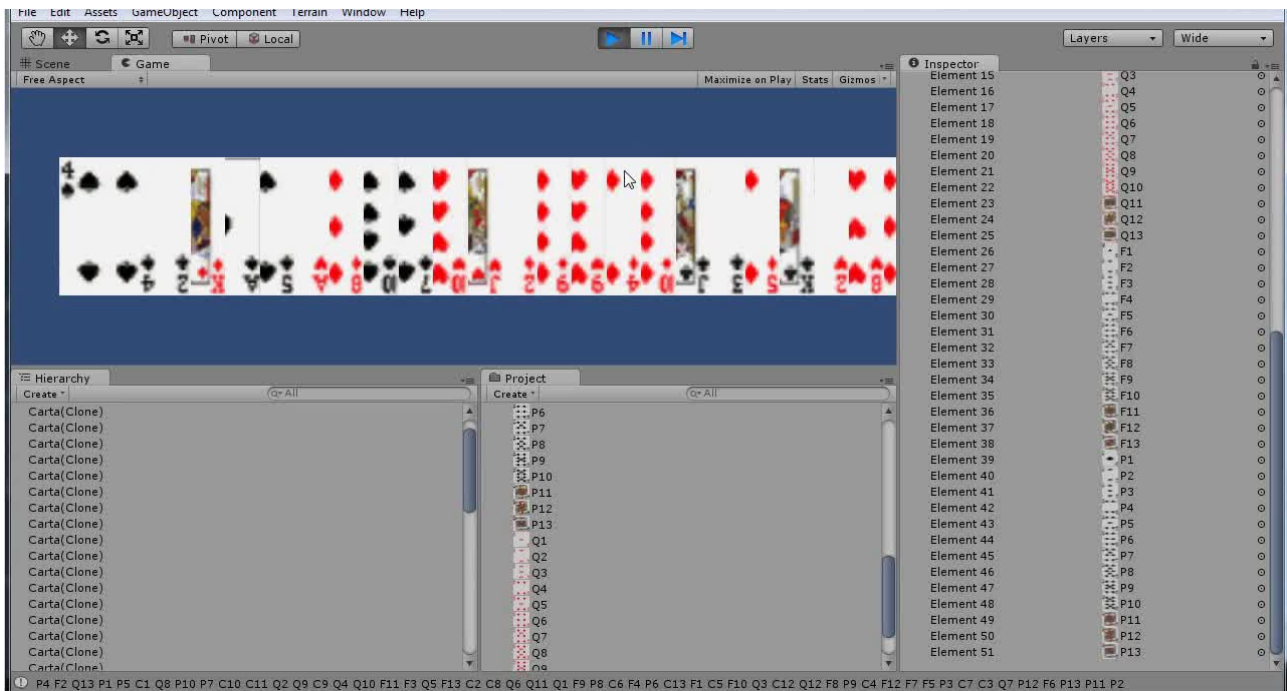
Facendo click sulla freccia accanto allo slot *ArrayTextures* nel componente **script**, notiamo con orrore che ci sono 52 slot numerati e, sì, **dobbiamo associare le immagini delle Textures**, caricate nella cartella all'interno del *Project*, ai vari slot!



Questa fase di click-e-trascinamento manuale richiede un po' di tempo (nell'immagine seguente, un “istante” di questa fase, con alcune textures assegnate), ma il punto importantissimo che non deve sfuggirvi è che gli elementi vanno inseriti nell'array **secondo l'ordine CQFP**, infatti nel codice dello script noi **inizializziamo il Dizionario codici-textures** secondo quest'ordine, nei quattro cicli **for** all'interno di *inizializzaMazzoOriginale*, quindi le **Textures** DEVONO seguire quest'ordine.



Fatto questo lavoro salviamo lo script, salviamo il progetto e avviamo il gioco: adesso avremo le immagini delle carte disposte di fronte alla telecamera secondo l'ordine degli elementi nell'array, con i codici visualizzabili nel **Debug Log** in basso; ad ogni pressione della barra spazio, che richiama la funzione *mescola*, rimescoleremo sia i dati nell'Array che le **associazioni carte-Textures**, visto che alla fine di ogni chiamata *mescola* viene invocata anche la funzione *aggiornaImmagini*.



* * *

Muri a spinta – I parte

Questo è il primo di due tutorial riguardanti la **registrazione di animazioni in modo oggetto** e il rilevamento delle **collisioni per tag** in **Unity 3D**.

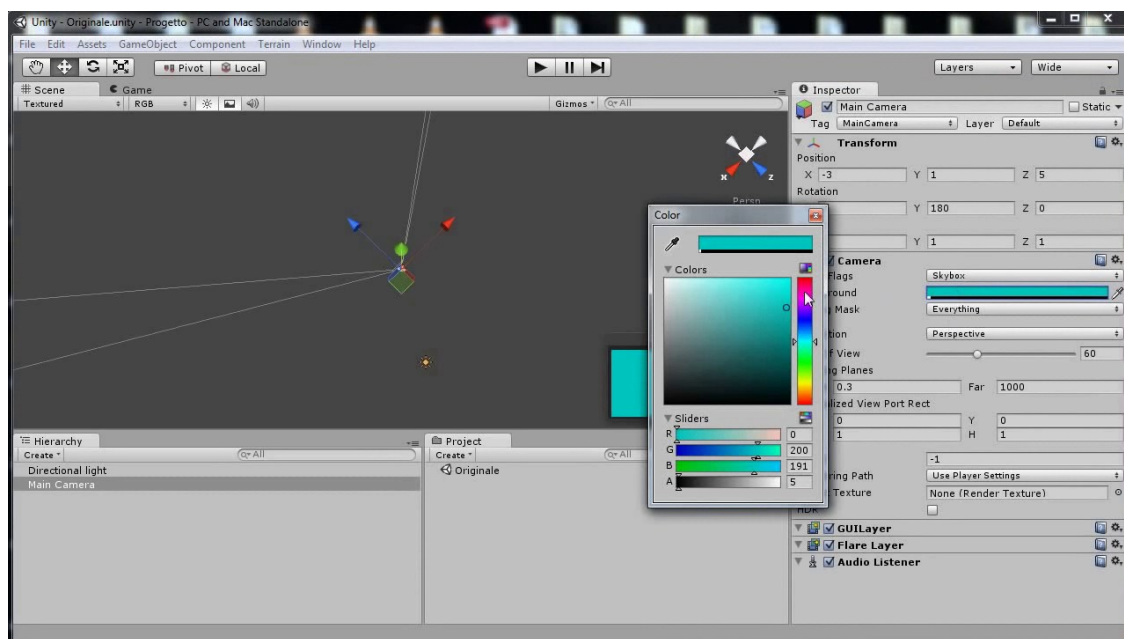
Ho chiamato questi due tutorial “**Muri a spinta**” in riferimento alle pareti di **Wolfenstein 3D**... non mi riferisco a *Castle Wolfenstein* o *RtCW* o altri, usciti di recente, ma proprio a *Wolfenstein 3D* del 1992; comunque, lì, camminando e premendo **il tasto per aprire le porte** vicino a certe pareti, queste si spostavano, mostrando dei **passaggi segreti**... è proprio quello che stiamo per fare qui.

Per realizzare in proprio questo tutorial si dovranno avere degli **assets** di gioco: la **texture** delle pareti e i **tre oggetti mesh** creati e **uv-mapped in Blender 3D** (in questo tutorial non tratteremo modellazione e UV Mapping in Blender 3D, per i quali vi rimando ai vari tutorial che ho pubblicato su sito e canale Youtube, oltre agli ebook), con le **mesh** che dovranno essere **UV-Mapped**, in modo da vedere l'immagine texture **mappata correttamente** quando verrà applicata alle mesh (magie dell'importer **blend** di Unity 3D).

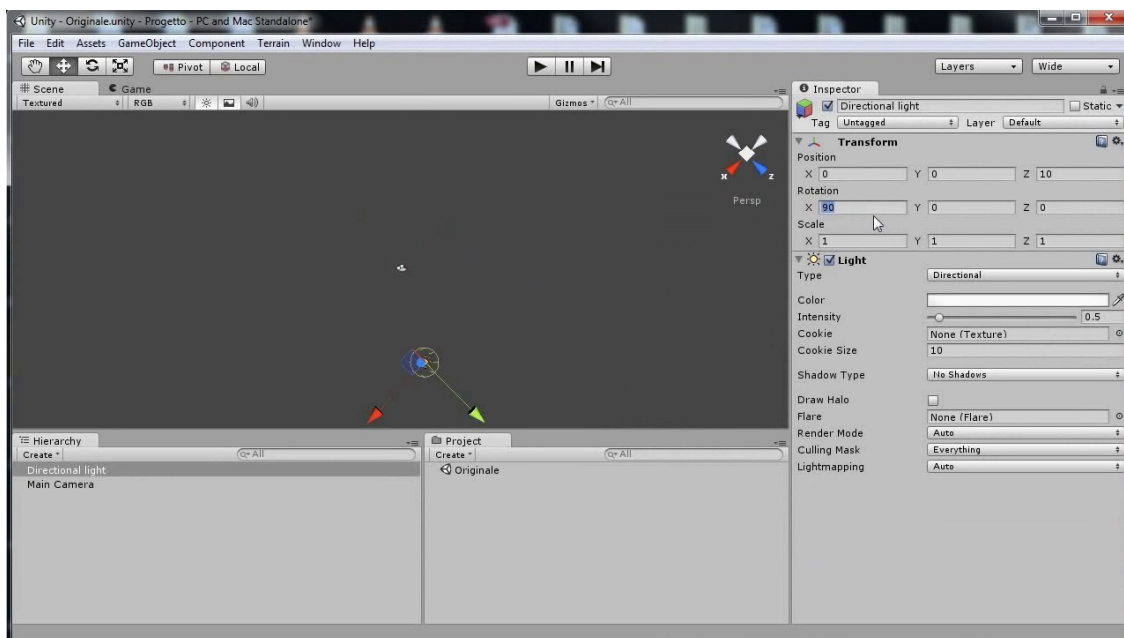
Animazione in modo oggetto e gestione delle collisioni via script con tag sono argomenti diversi, quindi ho deciso di dividerli in due capitoli: in questo, vedremo **l'animazione**.

Iniziamo con un nuovo progetto e una scena completamente vuota, cioè con la sola telecamera virtuale nella scena.

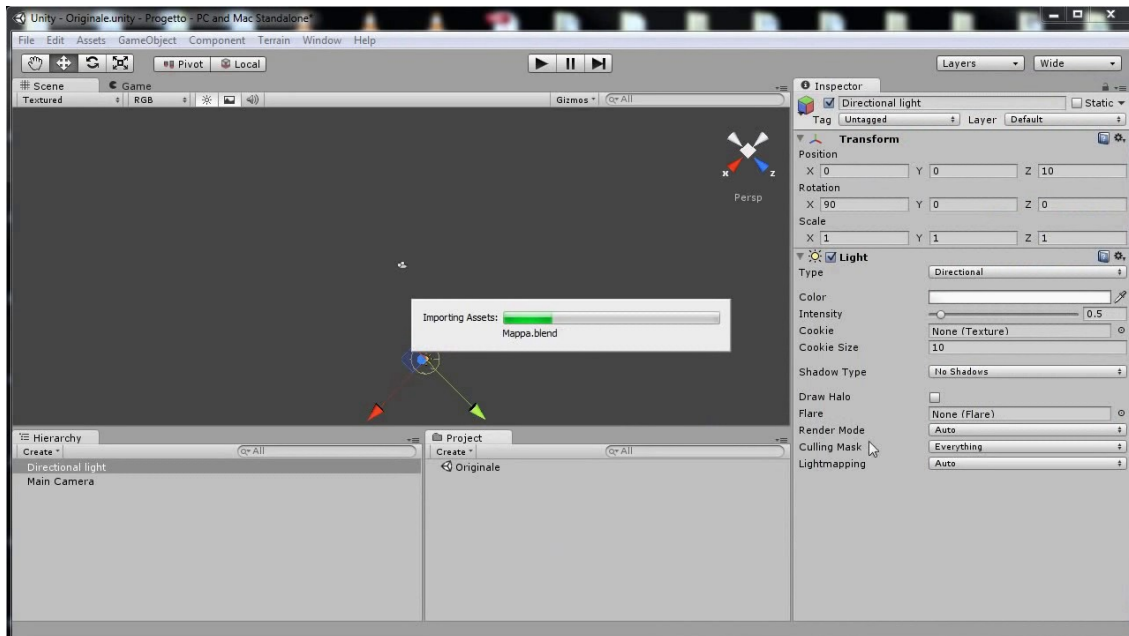
Per prima cosa ho cambiato il colore di **Background** dall'Inspector dell'oggetto **Main Camera**, riprendendo un po' quello di **Wolfenstein 3D** (se non lo conoscete, vi consiglio di cercare qualche screenshot su Google, per farvi un'idea del gioco, prima di continuare).



Ho aggiunto, inoltre, **una fonte di luce direzionale**, orientata verso il basso (**Position** non è importante, mentre **Rotation** XYZ valgono rispettivamente 90, 0, 0), in modo da illuminare oggetti e Textures.

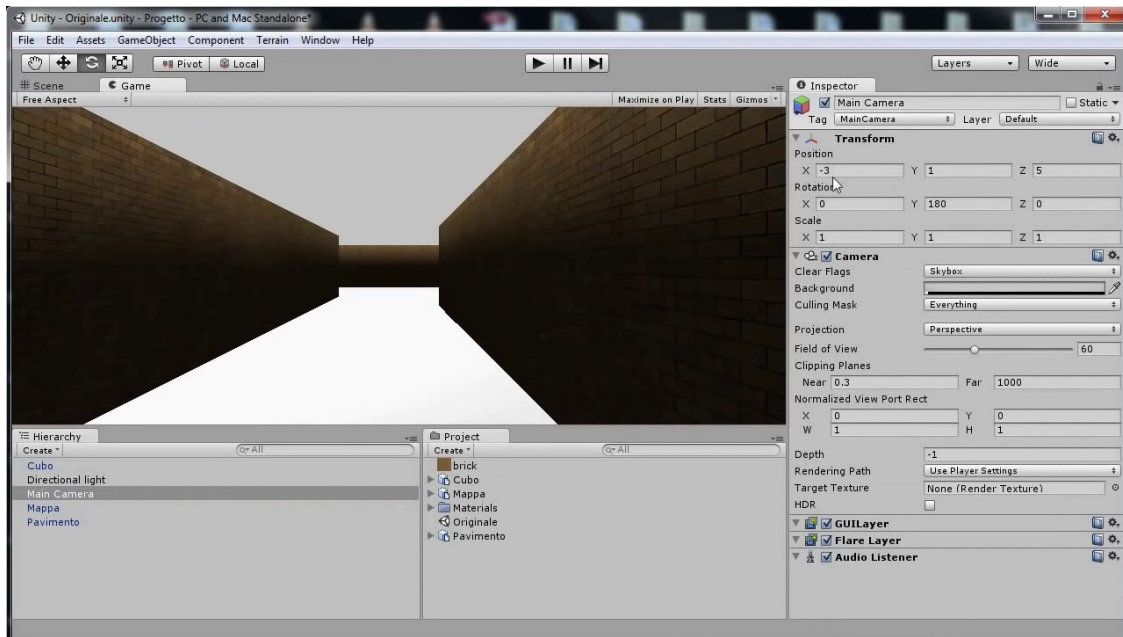


Adesso possiamo **importare**, nella scena 3D, gli **assets** di gioco, ossia trasciniamo dalla cartella su disco alla scheda **Project** i tre file blend e il file della texture. Insieme ai quattro elementi, che verranno creati come **Prefabs**, verranno creati anche altri elementi per informazioni **geometriche** e **materiali**, ma per il momento ignoriamoli.

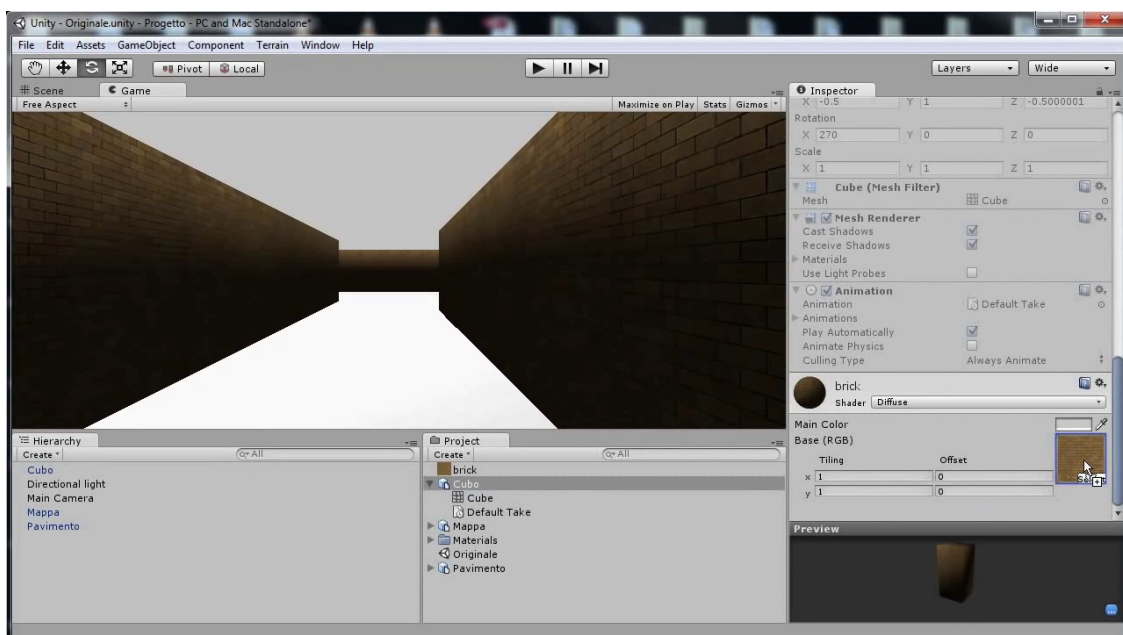


I Prefabs **“Cubo”**, **“Mappa”** e **“Pavimenti”**, ossia quelli delle tre mesh dello scenario, sono nel progetto, ma non nella scena corrente, quindi trasciniamoli uno per uno nella **Gerarchia**; siccome ho posizionato l'elemento **“Cubo”**, quello del muro a spinta, proprio dove deve trovarsi nella mappa, lo troveremo nella giusta posizione anche nella scena del gioco... anzi, in generale **le posizioni dei vari elementi** dovrebbero essere quelle corrette, per cui poi dovremo selezionare la **Main Camera** e riposizionarla e riorientarla con questi valori: Position XYZ -3, 1, 5; Rotation XYZ 0, 180, 0.

Selezioniamo l'oggetto **Main Camera** e diamo un'occhiata alla schermata **Preview** oppure selezioniamo la vista **Game** per vedere la scena inquadrata; **se l'inquadratura va bene**, lasciamola così, altrimenti sistemiamo posizione e orientamento dell'oggetto.



Le tre **mesh** inserite nella scena dovrebbero essere già texturizzate, visto che abbiamo importato, nel **Project**, anche l'immagine usata nel file **.blend** come **Texture UV-Mapped**; in caso di problemi, selezioniamo uno per uno i **Prefab** nel **Project** e, cliccando ma tenendo premuto, per selezionare e trascinare, impostiamo l'oggetto **Texture Brick** come **Texture** dei vari **Prefab**, nei loro **Inspectors**.



La scena è impostata, per cui possiamo occuparci finalmente dell'**animazione del muro a spinta**.

Quello che vogliamo è che il **Cubo si sposti lungo l'asse X**, rivelando la stanza segreta, quando la **Main Camera** è a **contatto** con esso (verificando quindi una **collisione**) e l'utente **preme il tasto** per **aprire le porte**, ad esempio la barra spazio.

Ci serve, quindi, una semplice **animazione**, in modo **oggetto**, ossia senza deformare la struttura o forma della mesh: **un semplice spostamento sull'asse X**, e questa animazione deve essere effettuata a comando, ossia **via script**, ma di questo parleremo nel prossimo capitolo: qui, vediamo **come registrare l'animazione** da mettere poi in esecuzione.

Se avete qualche base di animazione con software 2D o 3D, come **Blender** o **3D Studio**, e conoscete i concetti di **keyframes**, allora quello che segue vi apparirà subito chiaro e il tutto sta nel sapere dove si trovano finestre e comandi; per tutti gli altri, due parole ora, prima di proseguire.

Immaginate **il tempo**, dall'avvio del gioco, come una retta, con istante di partenza 0.

Aniché pensare al tempo in secondi, pensate ai **frame o fotogrammi**, quindi potremmo avere 25, 30, 60 **fotogrammi al secondo**.

Questo è il tempo: **una successione di frames**; dall'altra parte, abbiamo le **caratteristiche degli oggetti**, ad esempio la posizione di un elemento espressa con tre valori: le coordinate X, Y e Z.

In Computer Grafica, **le animazioni** vengono effettuate **registrando i valori delle proprietà** degli oggetti in certi **fotogrammi-chiave**, detti appunto **keyframes**, e lasciando al software di animazione il compito di calcolare i valori intermedi (ossia, di interpolare i valori) tra i vari frames che ci sono tra due fotogrammi chiave (nell'animazione tradizionale, dove i disegnatori devono realizzare uno per uno questi elementi tra un keyframe e l'altro, questi frames sono detti **in-betweens**, ossia “elementi di mezzo”).

Il software, durante l'esecuzione del gioco, mostrerà quindi la **successione di frames nel tempo**, e poiché ad ogni frame verrà calcolato il nuovo valore per il parametro registrato e l'elemento verrà

aggiornato di conseguenza (ossia: il cubo sarà in una nuova posizione), **la successione di frames a video** (possibilmente con frame rate maggiore di 25, per rendere il movimento fluido) costituirà **l'animazione**.

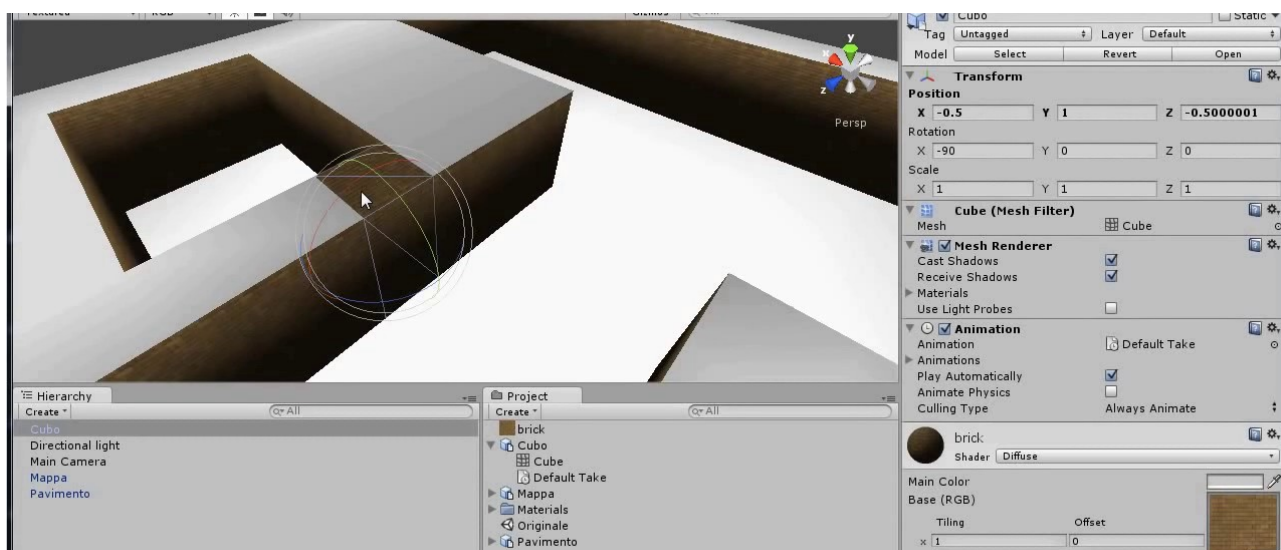
I **keyframes** vengono quindi scelti in base a **cambiamenti particolari** nelle caratteristiche di un elemento e **l'animazione è il passaggio** da uno stato speciale all'altro.

Nel nostro caso, **il cubo ha due stati speciali**: il primo è la posizione originale, che vediamo all'inizio, mentre l'ultimo è la posizione finale, quando torna fermo, dopo l'animazione, per liberare il passaggio.

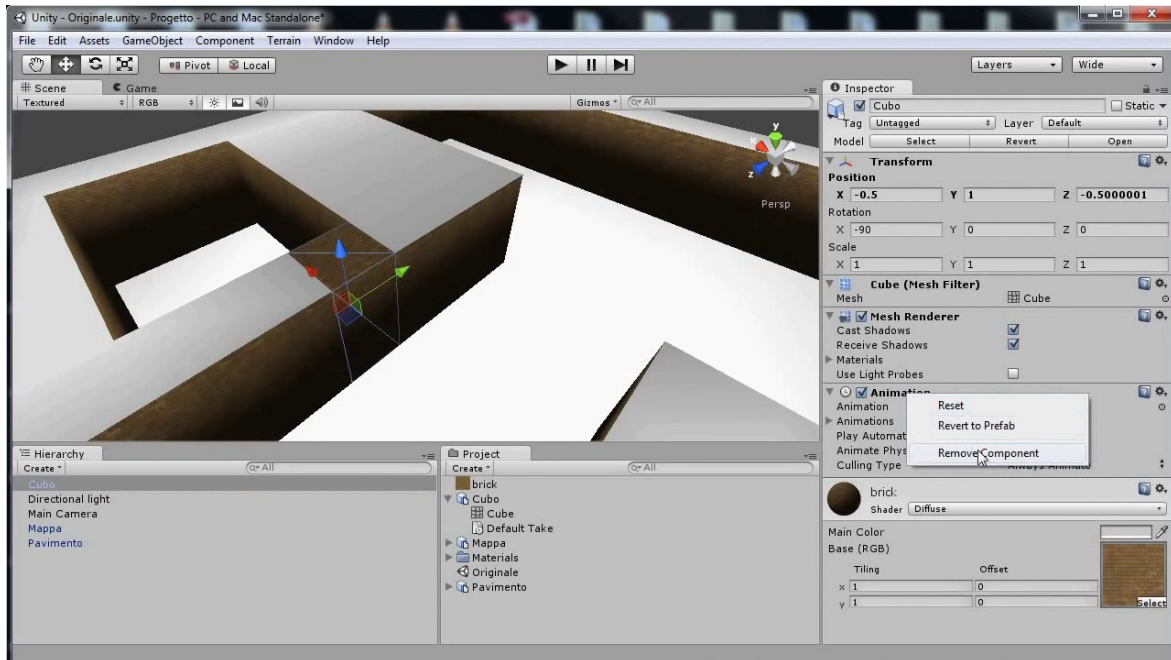
Queste due posizioni saranno quindi **due keyframe** e lo spostamento sarà l'animazione calcolata in automatico da **Unity**: noi dobbiamo definire solo **le coordinate per i keyframes**, **Unity interpolerà** i valori per noi (ossia: creerà gli in-betweens, animerà l'oggetto... a questo punto, sono tutti modi per esprimere lo stesso concetto).

Per nostra fortuna, l'implementazione di tutto questo discorso teorico è abbastanza semplice, in **Unity**, per cui dobbiamo solo sapere **dove si trovano le finestre e i comandi** per registrare le animazioni nella scena e come dire al programma di **eseguire le animazioni solo a comando**, non automaticamente all'avvio del gioco.

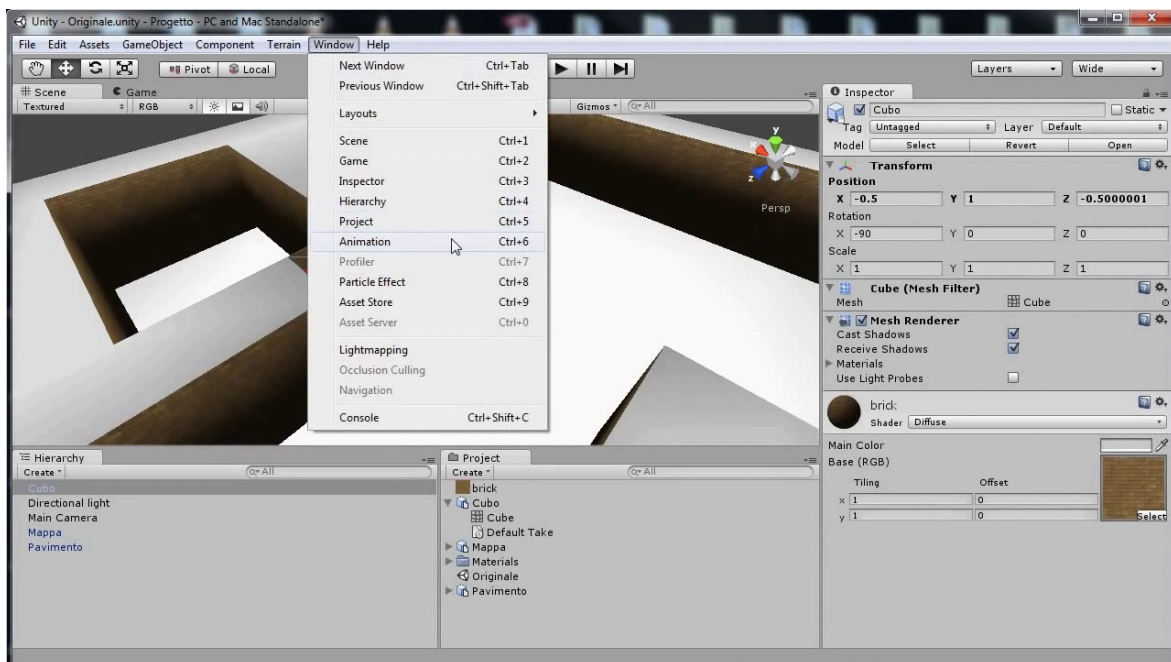
Per prima cosa, selezioniamo il **cubo** da spostare, nella **Gerarchia**.



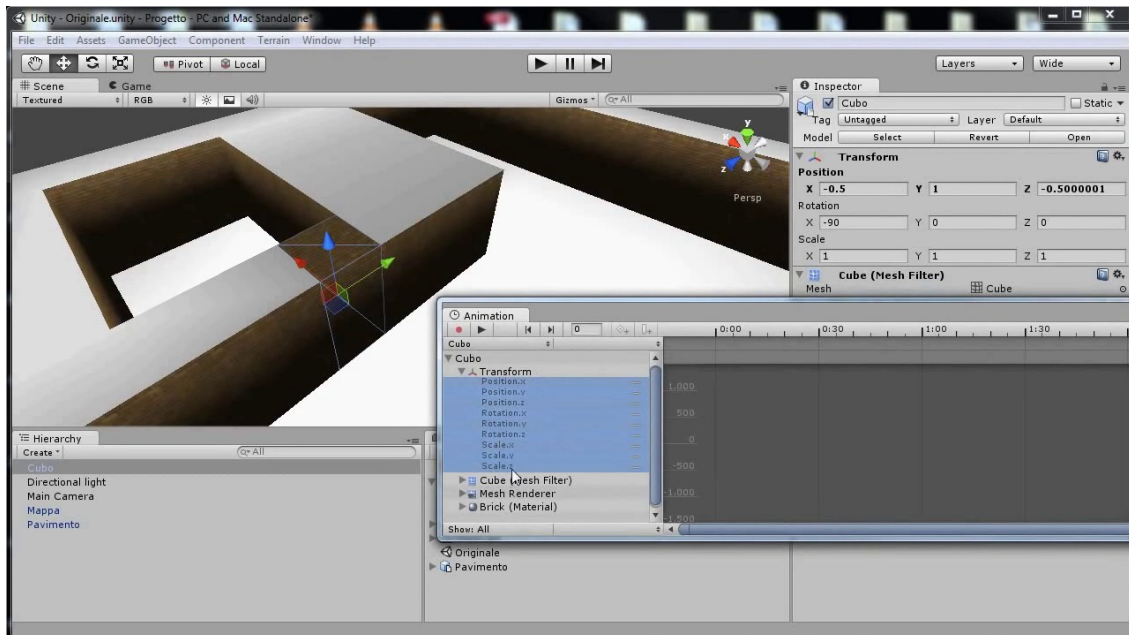
Con tale elemento selezionato, nell'Inspector **rimuoviamo il Component *Animation*** presente di default con click destro e ***Remove Component*** su tale elemento nell'Inspector.



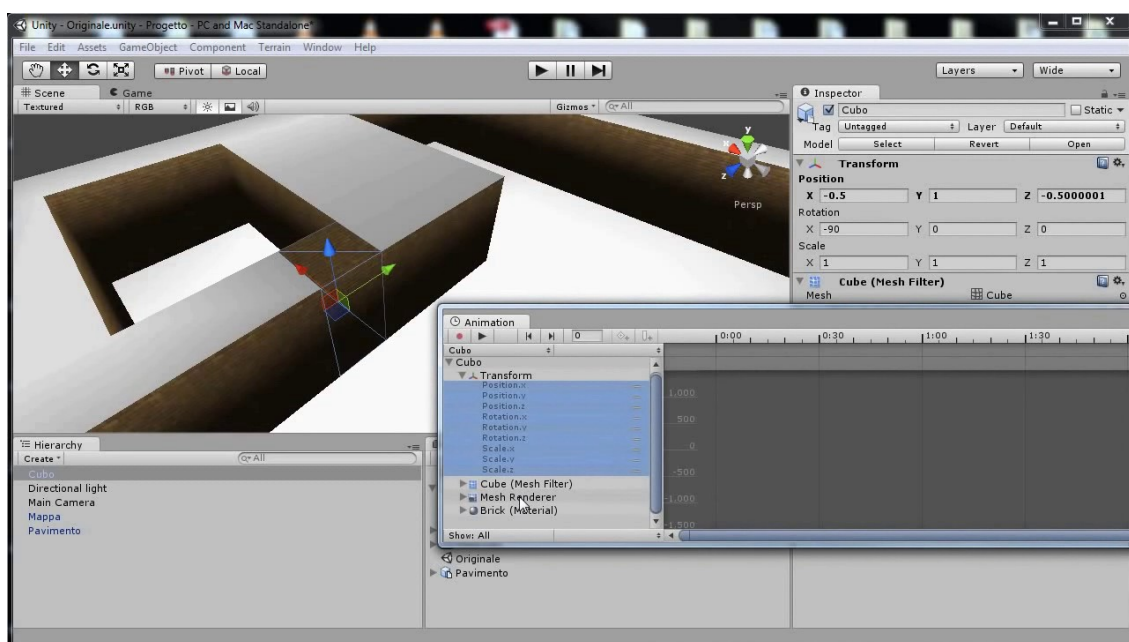
Adesso apriamo la finestra delle animazioni scegliendo, dal menù **Window**, **Animation** (shortcut CTRL 6).



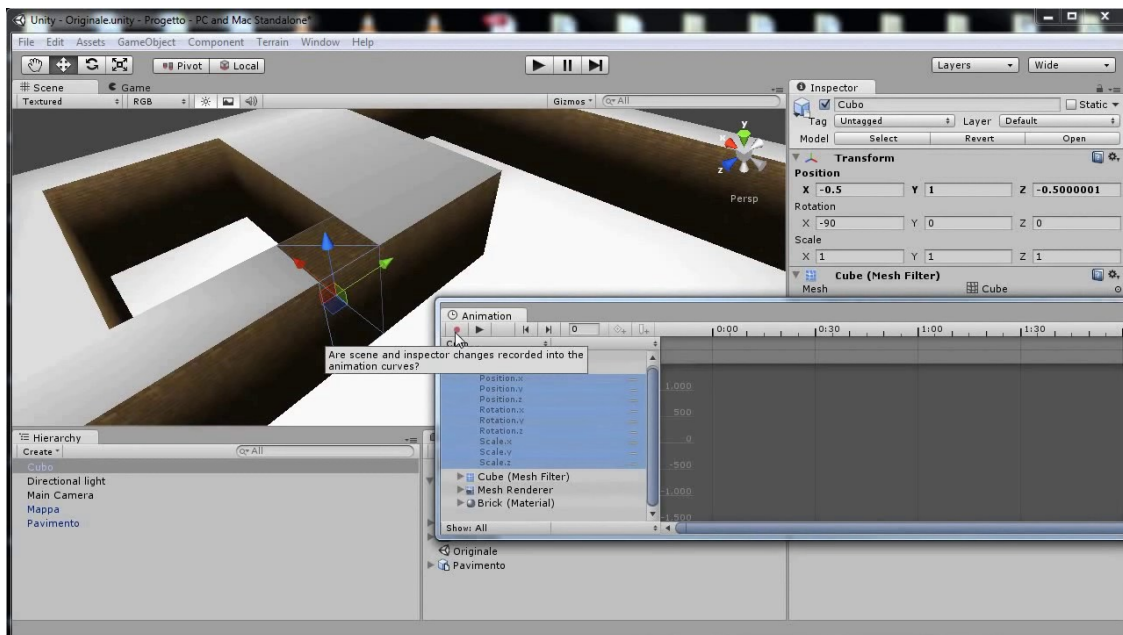
In questa finestra sulla sinistra abbiamo, per l'oggetto selezionato, i canali animabili, ossia quelli per i quali possiamo **registrare i keyframes**, mentre sulla destra abbiamo una rappresentazione della **barra del tempo**, con il tempo espresso in **frames**, nella parte superiore.



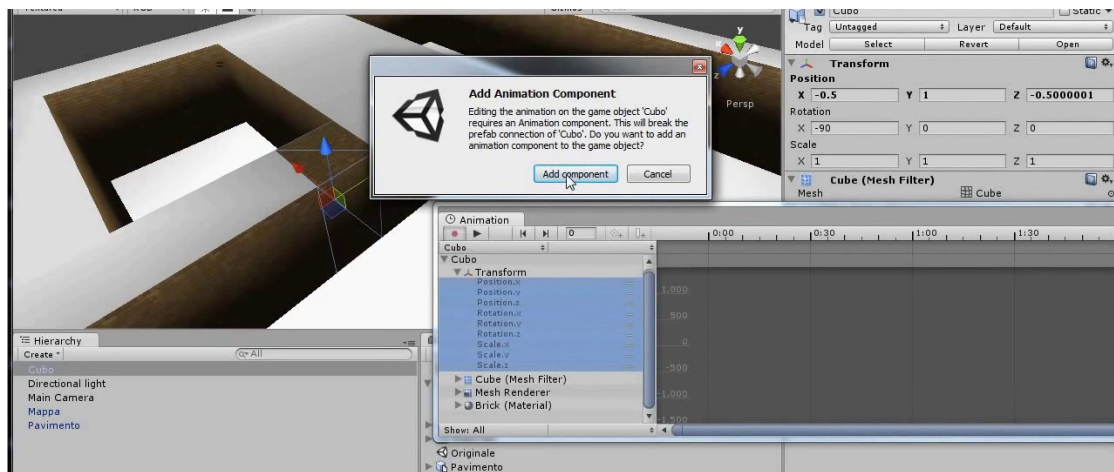
All'inizio, **non abbiamo curve** che rappresentano le **animazioni**, in quanto non abbiamo **registrato** ancora alcuna operazione. Sappiamo che vogliamo modificare le informazioni su **Position**, quindi per prima cosa **clicchiamo su tale blocco** a sinistra, nella finestra.



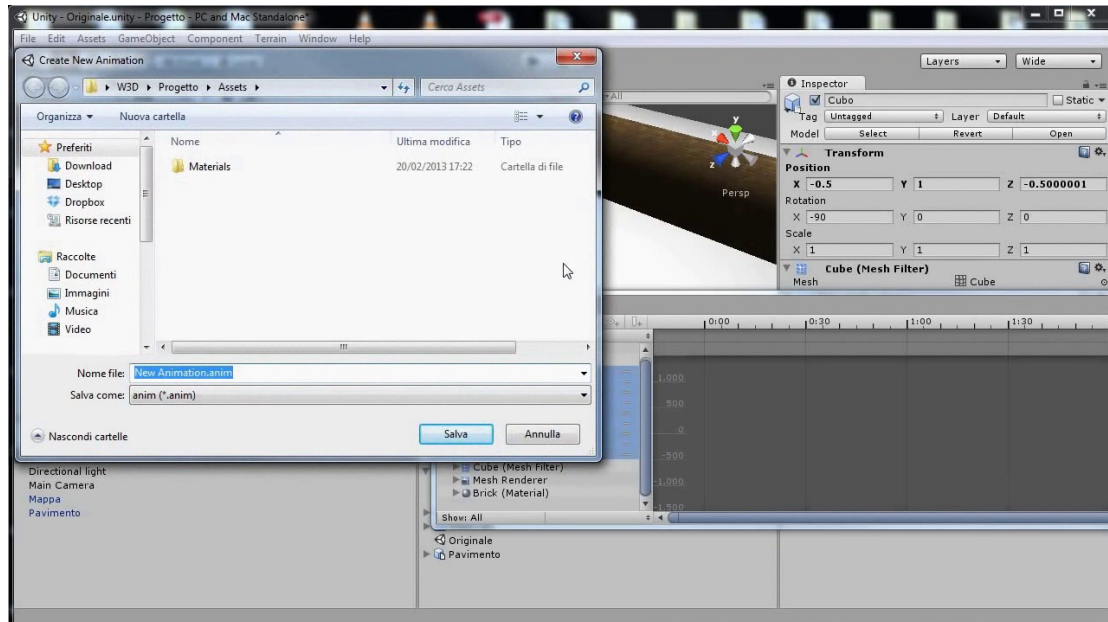
Per registrare i vari valori, entriamo in modalità **“Record”** (registrazione), in quanto in tale modalità potremo **fissare i keyframes**, quindi clicchiamo sul **cerchio rosso** (un pulsante di registrazione): **Unity** ci dirà che deve creare un componente di animazione diverso da quello di default e che questa operazione farà perdere il collegamento con il **Prefab**, con tutte le conseguenze del caso.



Unity ci chiederà, poi, di **salvare le informazioni** di questo componente di animazione come **file .anim** su disco... può sembrare strano, in quanto il salvataggio delle informazioni di un Component come file a parte su disco non avviene spesso in Unity, ma questo è uno di quei casi, per cui scegliamo un nome per tale animazione e salviamo il file.

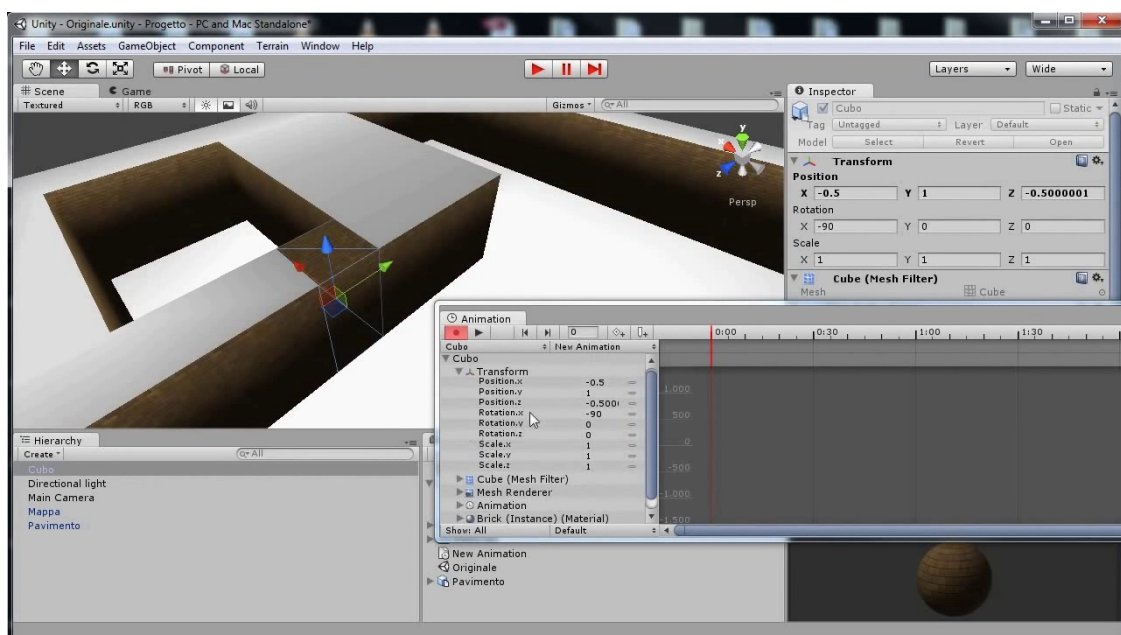


Nell'immagine seguente: **salvataggio del file .anim** su disco (all'interno della cartella *Assets* del progetto corrente).



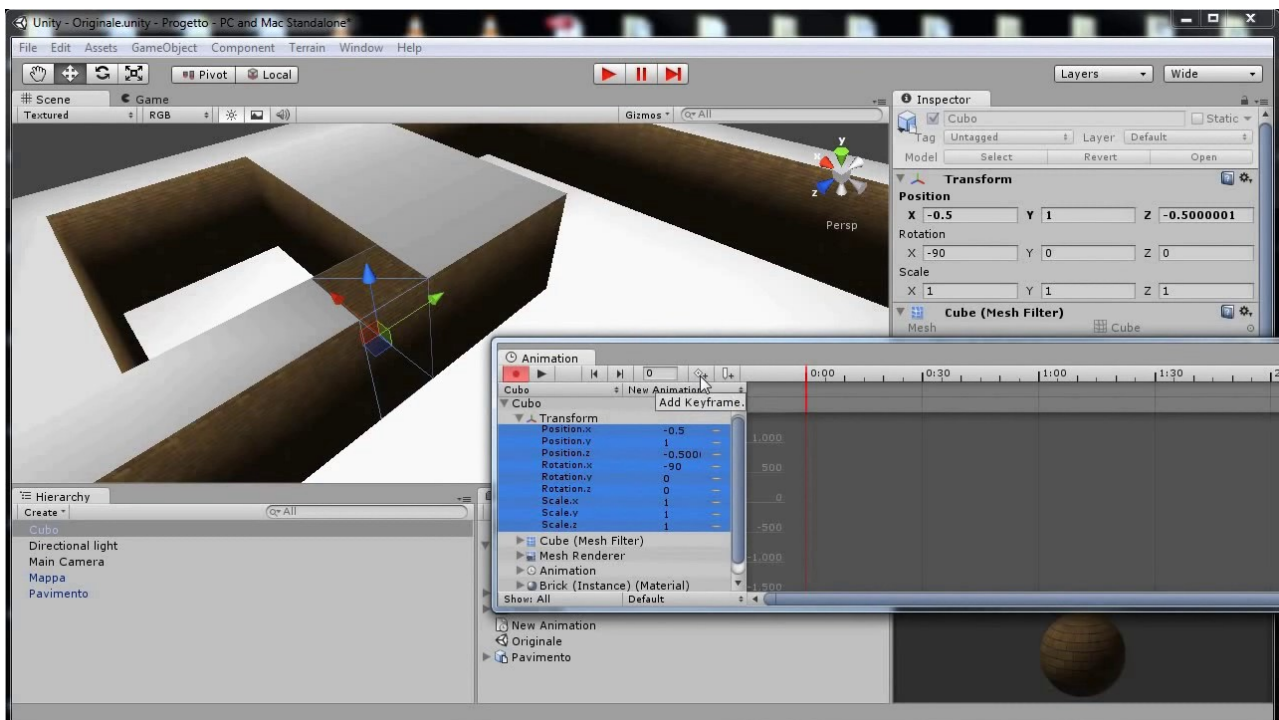
Effettuato il salvataggio, saremo di nuovo nella scena con la finestra **Animation** aperta e in modalità **Registrazione**, come si può constatare dando un'occhiata ai pulsanti **Play**, **Record** ecc, che ora appaiono **rossi**, con il tasto **Record** “premuto”.

Nella finestra **Animation**, il valore 0 in alto a sinistra ci dice che siamo **al primo frame** della **Timeline**, la barra del tempo, e questo ci va più che bene.



Ci va anche bene **la posizione del cubo** nella scena, visto che all'inizio deve trovarsi proprio lì, per mimetizzarsi col resto della parete, per cui **non ci resta che registrare questi valori al frame 0**.

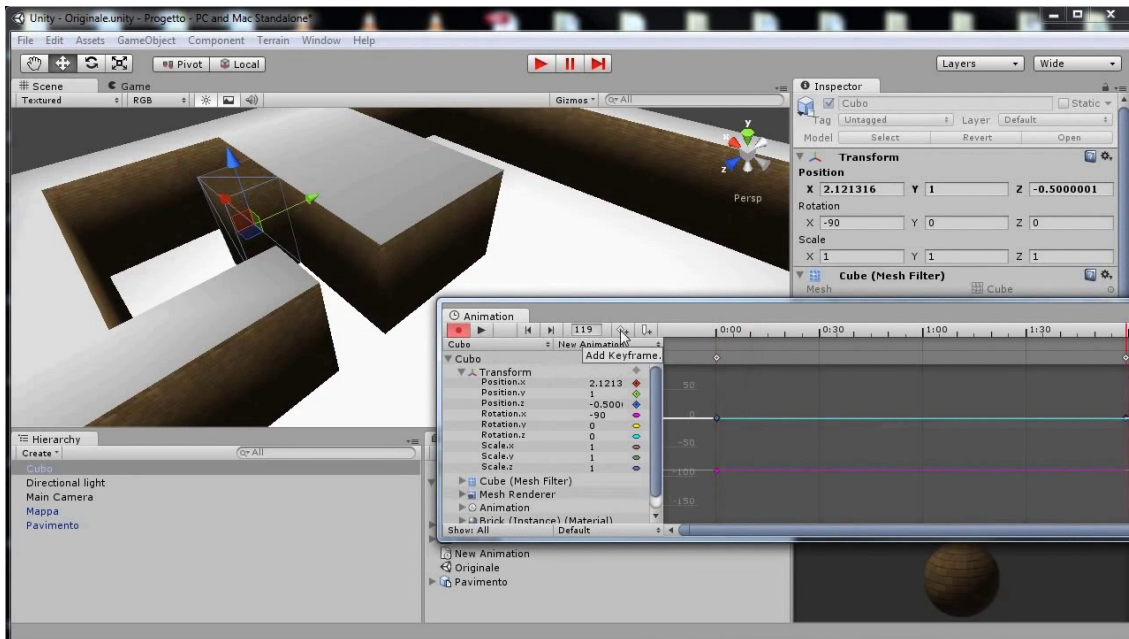
Selezioniamo quindi il canale **Transform** nella finestra **Animation** e, sempre in questa finestra, clicchiamo sul pulsante **Add Keyframe**, a destra del campo con il frame corrente, indicato nell'immagine seguente: fatto!



Nella finestra **Animation** appariranno dei punti che indicano i **keyframe** con i vari valori per i canali selezionati. In questo spazio potete spostarvi e zoomare come nella scena 3D, ruotando la rotellina o cliccandola e trascinando.

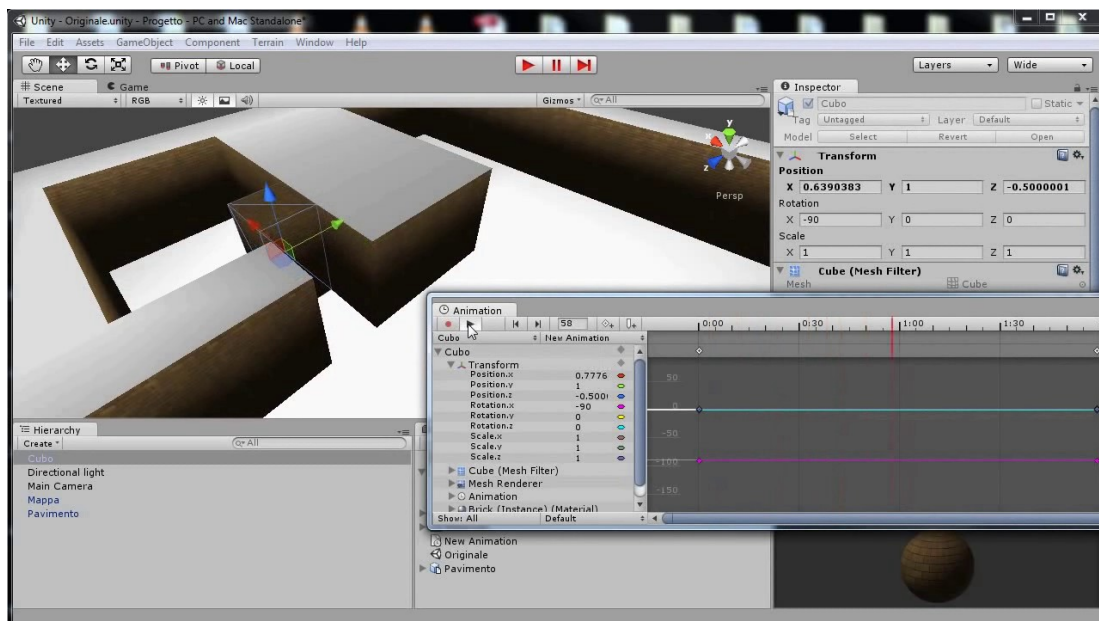
Adesso spostiamoci al **120esimo frame della Timeline** scrivendo 119 (si parte da 0) nel campo **frame number** della finestra **Animation**, quindi nella scena 3D spostiamo sull'asse X il cubo, posizionandolo dove vogliamo farlo finire al termine dell'animazione, per liberare il passaggio.

Come avrete intuito, tutto quello che resta da fare ora è **cliccare nuovamente sul pulsante Add Keyframe** per aggiungere un **keyframe** per il cubo in questa posizione: **Unity** farà il resto, creando l'interpolazione tra i valori frame per frame, ossia realizzando in effetti l'animazione.

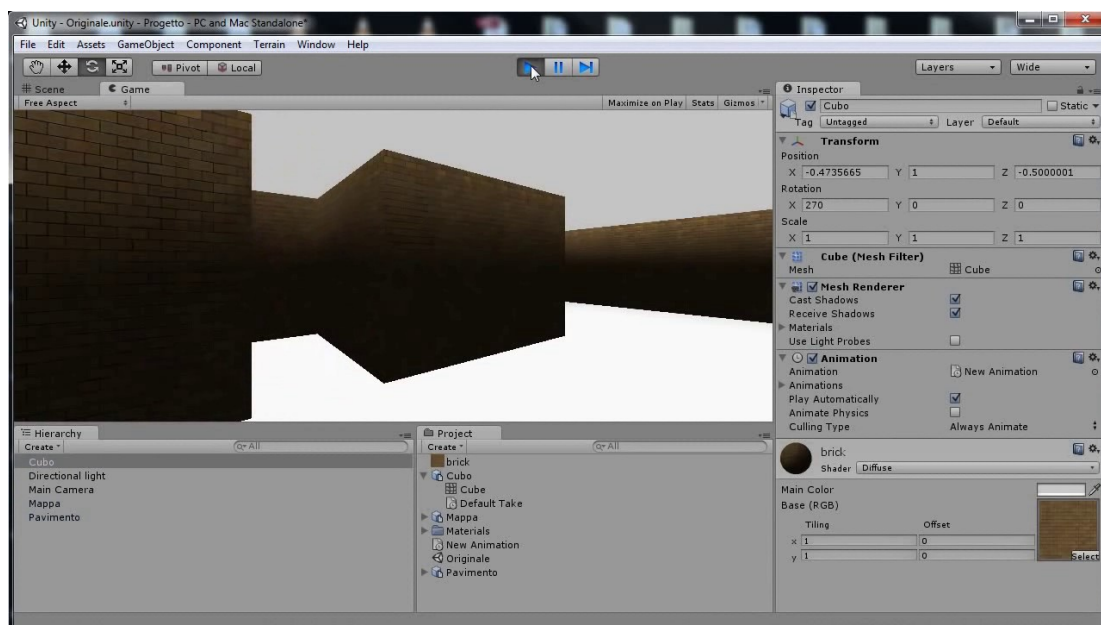
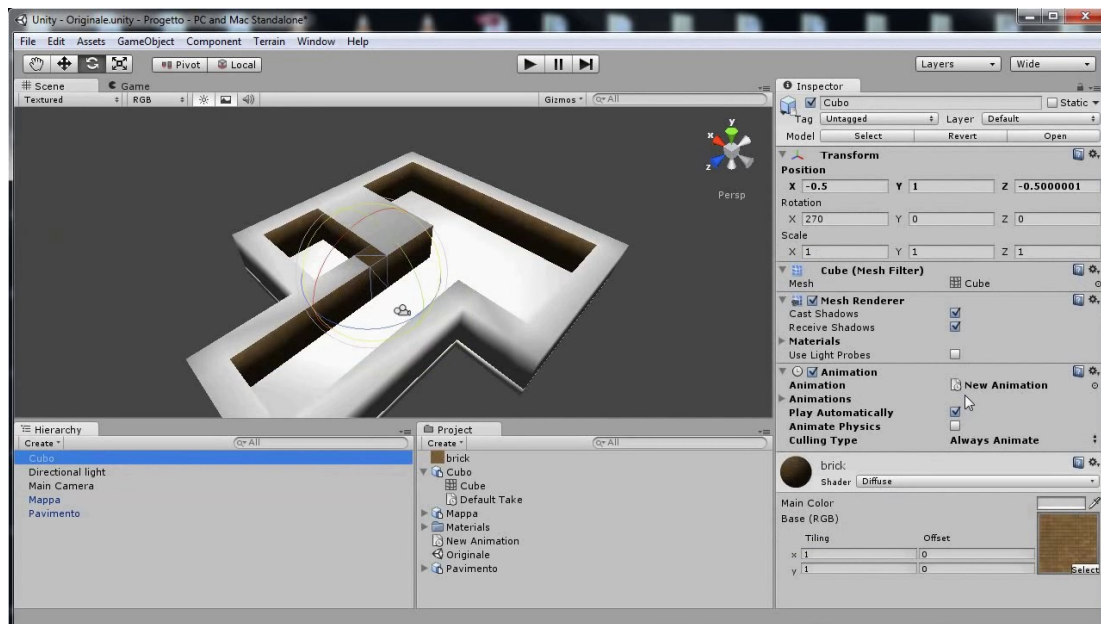


A questo punto possiamo anche uscire dalla modalità **Record** cliccando sul **cerchietto rosso**.

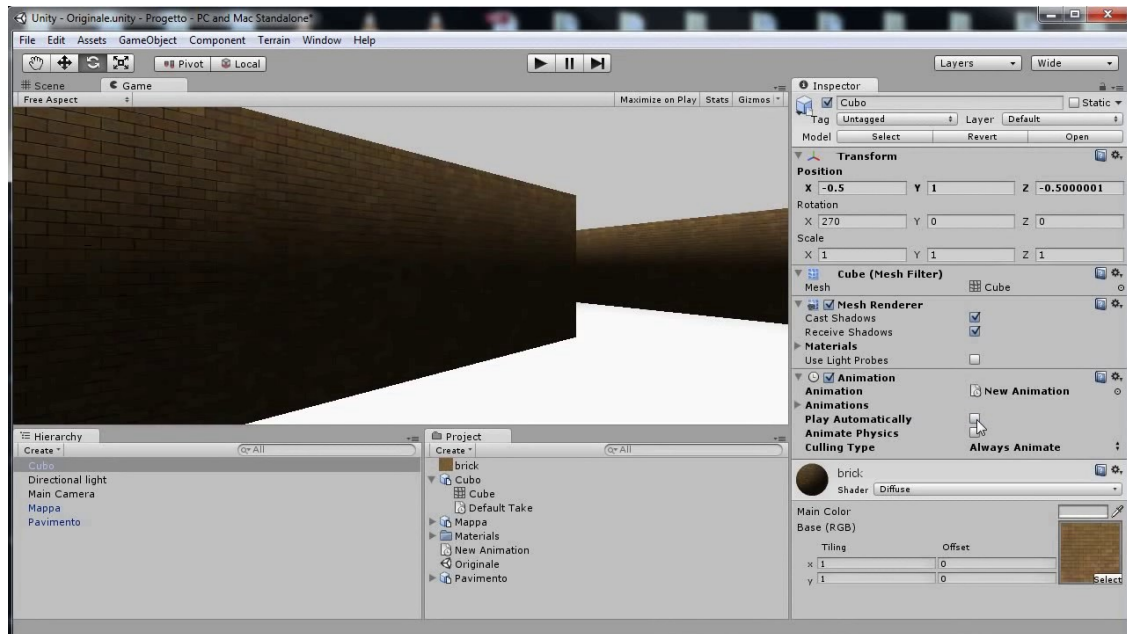
Possiamo **verificare l'avvenuta registrazione dell'animazione** cliccando, sempre nella finestra **Animation**, sul tasto **Play**: il cubo si sposta, con un timing che ci va bene (nota: l'anteprima dell'animazione verrà ripetuta qui di continuo, ma noi la effettueremo solo una volta, nel gioco).



Possiamo **chiudere la finestra Animation** e, tornando alla 3D View, possiamo andare in modalità **Game** e premere **Play** per vedere il risultato in tempo reale: di default, un'animazione così creata viene eseguita subito, quindi dovremmo vedere il muro rientrare. Nel caso questo non dovesse succedere, selezionate il **Cubo** e verificate, nell'Inspector, che nello slot **“Animation”** del nuovo componente **Animation** sia presente, come nel mio caso, l'animazione creata e salvata su disco poco fa; **tale Animazione è un oggetto a tutti gli effetti**, infatti è nel **Project**, per cui se quello slot dovesse essere vuoto **trascinate manualmente** l'oggetto dal **Project** su quel campo.



L'animazione c'è, ma **parte subito all'avvio del gioco**, mentre noi vogliamo farla partire a **comando** (con la barra spazio) e a certe condizioni (la collisione), quindi **per togliere l'esecuzione automatica** deseleggiamo la casella **Play Automatically**, appunto, nell'Inspector del **Cubo**, nella sezione **Animation...** fatto!



Con questo primo capitolo ci fermiamo qui: abbiamo visto come accedere alla finestra **Animation** di un oggetto, come creare un **Animation Component**, **registrare** una semplice animazione in modo oggetto e visualizzarla, togliendo tra l'altro la riproduzione automatica; nel prossimo capitolo, passeremo allo **script per riprodurre l'animazione** a comando.

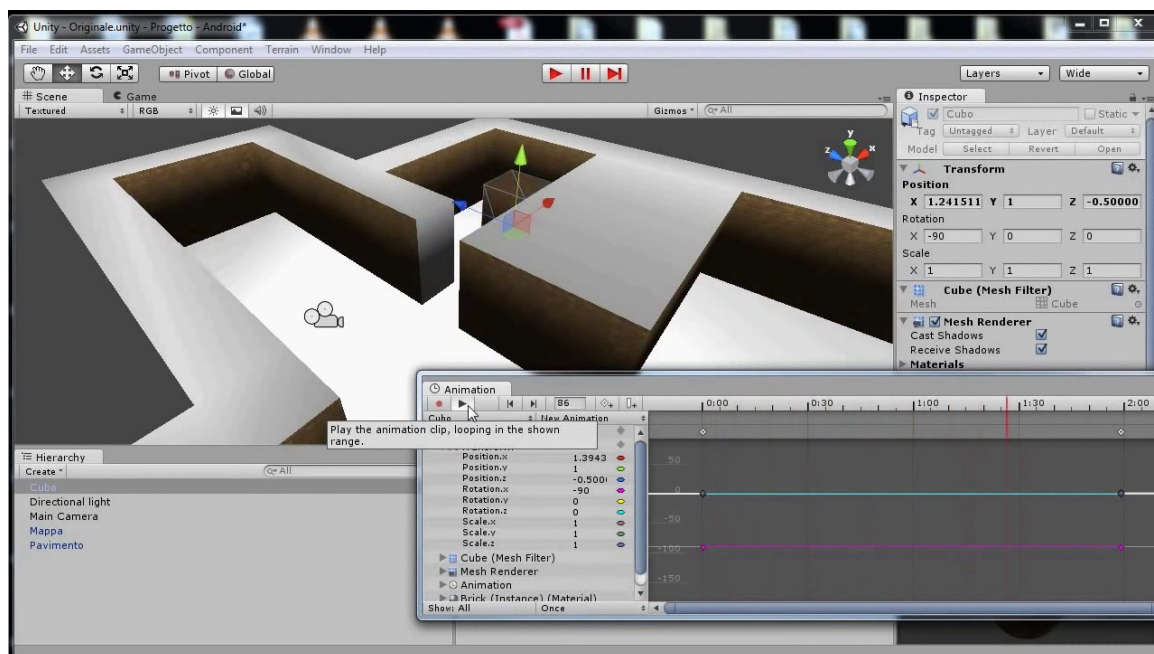
Muri a spinta – II parte

In questo secondo capitolo del tutorial sui muri a spinta vedremo **come rilevare la collisione** tra la telecamera e gli oggetti di scena e come **avviare, mediante script**, la riproduzione **dell'animazione** creata nel tutorial precedente. Tratteremo, tra le altre cose, **Colliders**, **Triggers** e **Tag**.

Un piccolo riepilogo: nella scena abbiamo un pavimento, alcune pareti, una telecamera, una fonte di luce direzionale e un muro particolare, dotato di un'animazione in modo oggetto (ossia, senza deformare la struttura, solo le caratteristiche di posizione, rotazione e scaling).

Le mesh sono dotate di **Materials** e in alcuni casi, o in alcune parti, di **Texture UV Mapped**.

Abbiamo disattivato la **riproduzione automatica** dell'animazione (deselezionando **Play Automatically** nell'Inspector), quindi all'avvio non succederà nulla, né possiamo **muovere la telecamera**.



Per ottenere l'effetto voluto dobbiamo implementare una serie di **caratteristiche**: per prima cosa, dobbiamo far **muovere la telecamera** a comando nella scena, quindi dobbiamo impedirle di **oltrepassare le pareti** (rilevando le collisioni) e infine dobbiamo **avviare l'animazione** del muro a spinta se la telecamera lo tocca e se l'utente preme la barra spazio.

Iniziamo dall'**animazione della telecamera**: anziché realizzare uno script di movimento, utilizziamo l'**FPS Controller** nativo di **Unity 3D**; vi spiego subito perché scegliamo questa strada.

C'è un problema dato dalle **collisioni**: per evitare che la telecamera passi attraverso le pareti e per farle rilevare la collisione, in particolare col muro a spinta, dovremmo inserire dei componenti **Colliders** e **Rigid Body**, tuttavia in caso di collisione tra **Rigid Body** questi verranno respinti, generando dei **rimbalzi**, ineliminabili persino settando a 0 la Bounciness dei **Physics Materials** dei **Colliders** (e le componenti *Collider* e *Rigid Body* sono distinte, separate)... per evitare di farvi perdere tempo con strade inutilmente complicate o che portano a vicoli ciechi, quindi, vi risparmio i tentativi con gli script e **vi dico subito la soluzione** che adotteremo (e ora sapete anche perché).

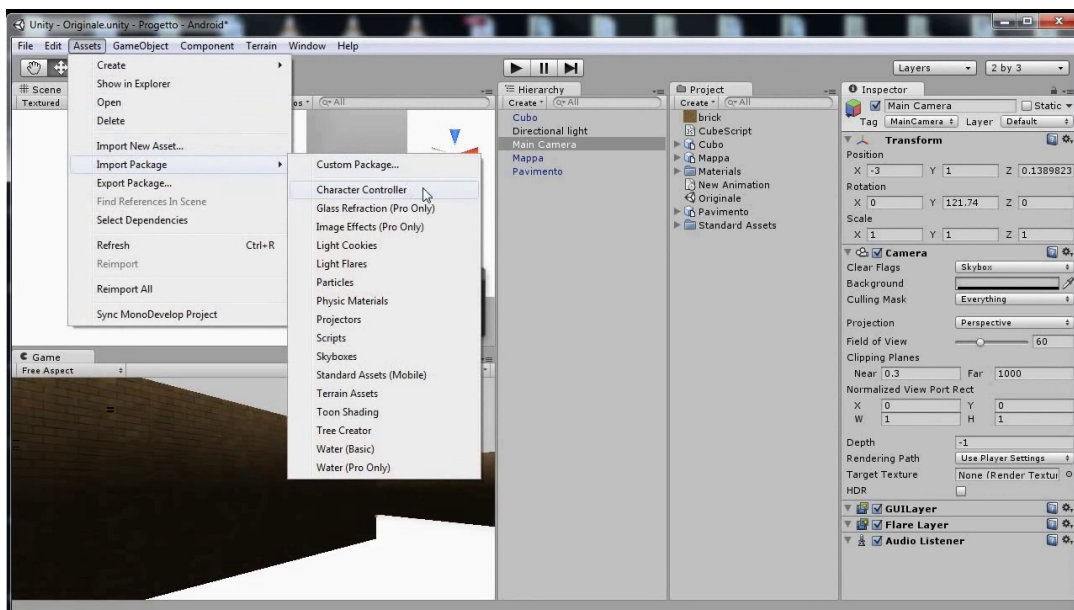
La soluzione è la seguente: **la telecamera avrà un FPS Controller** e un **Mouse Look**, elementi nativi di **Unity**, che tra l'altro ci risparmieranno un bel po' di codice; non solo: l'**FPS Controller** **rileva** automaticamente le **collisioni** con i **Rigid Body**, quindi il muro a spinta avrà un **Rigid Body** senza gravità e un **Box Collider**, per rilevare la collisione, informazione questa da trattare via codice, mentre le restanti pareti avranno un **Mesh Collider**, ossia un componente per rilevare le collisioni modellato grosso modo sulla forma della mesh importata.

Gli oggetti **Collider** sono infatti come delle “**gabbie**” da mettere intorno agli elementi per rilevare le collisioni; tali elementi possono avere varie **forme**: per il muro a spinta useremo un *Box*, ossia un parallelepipedo, mentre per una palla potremmo usare una *Sphere*, per la telecamera l'**FPS Controller** userà un *Capsule* e per oggetti più complessi, come tutte le pareti, useremo un generico *Mesh Collider*... una piccola nota: si usano dei *Colliders* anziché le mesh originali per ridurre la complessità dei calcoli, in quanto questi *Collider* hanno in genere molti meno vertici o spigoli degli oggetti che ingabbiano, quindi il motore di gioco farà meno calcoli per rilevare le collisioni.

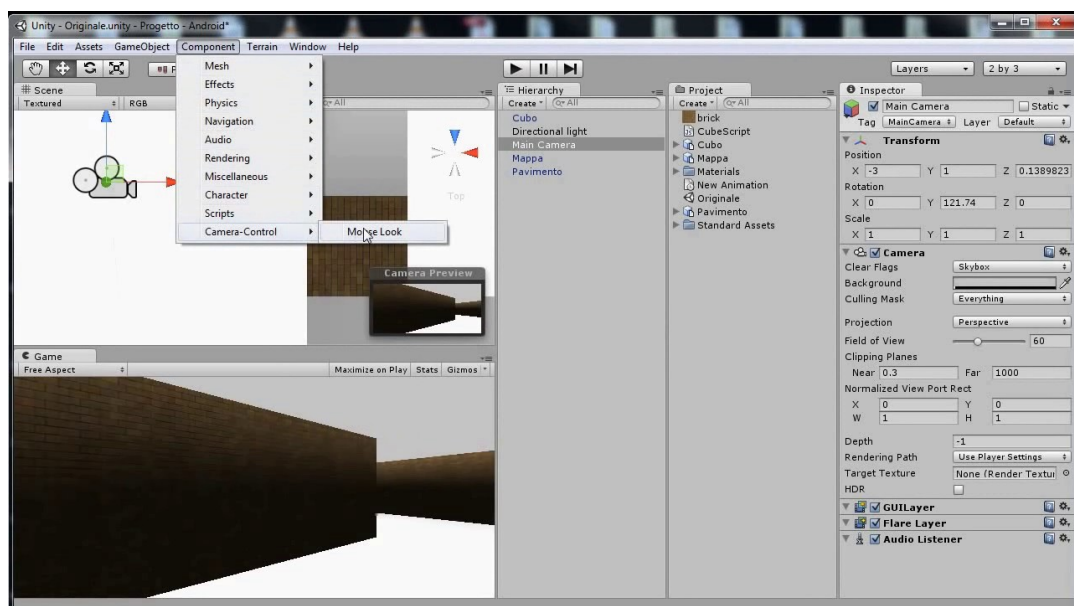
Un'ultima nota, prima di continuare: ho parlato dell'**FPS Controller**, del **Mouse Look** e di altri elementi che sto per utilizzare nei dieci tutorial sulle basi di **Unity**, quindi qui non li descriverò nuovamente da zero, ma li utilizzerò direttamente.

Possiamo finalmente passare alla pratica.

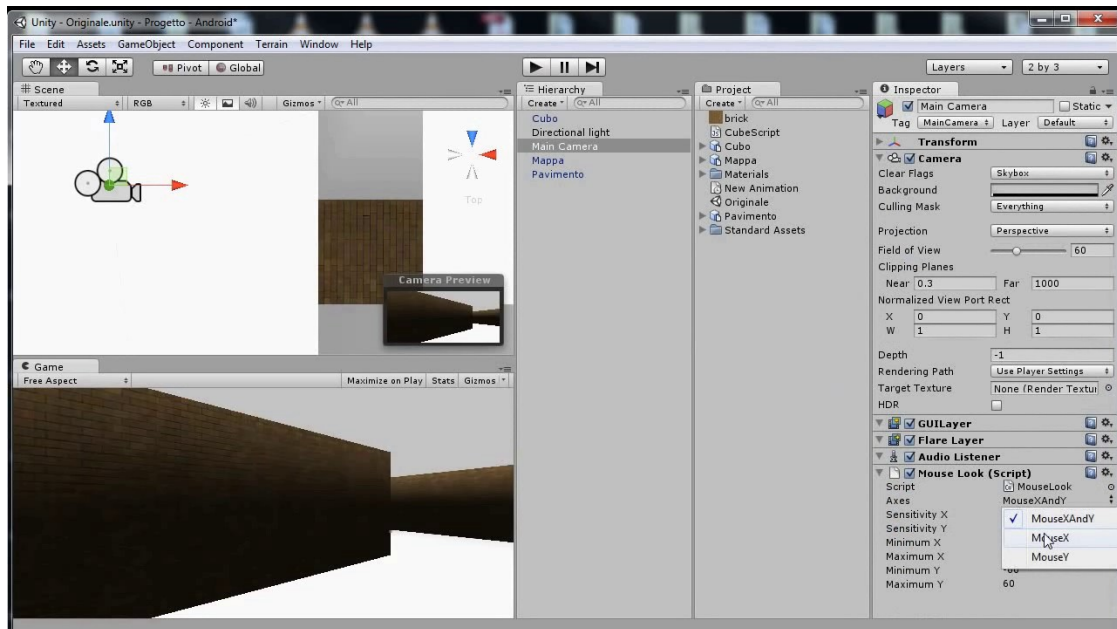
Per prima cosa, importiamo nel pacchetto il **Package Character Controller** da *Assets | Import Package | Character Controller*, quindi confermiamo l'importazione di tutti gli elementi.



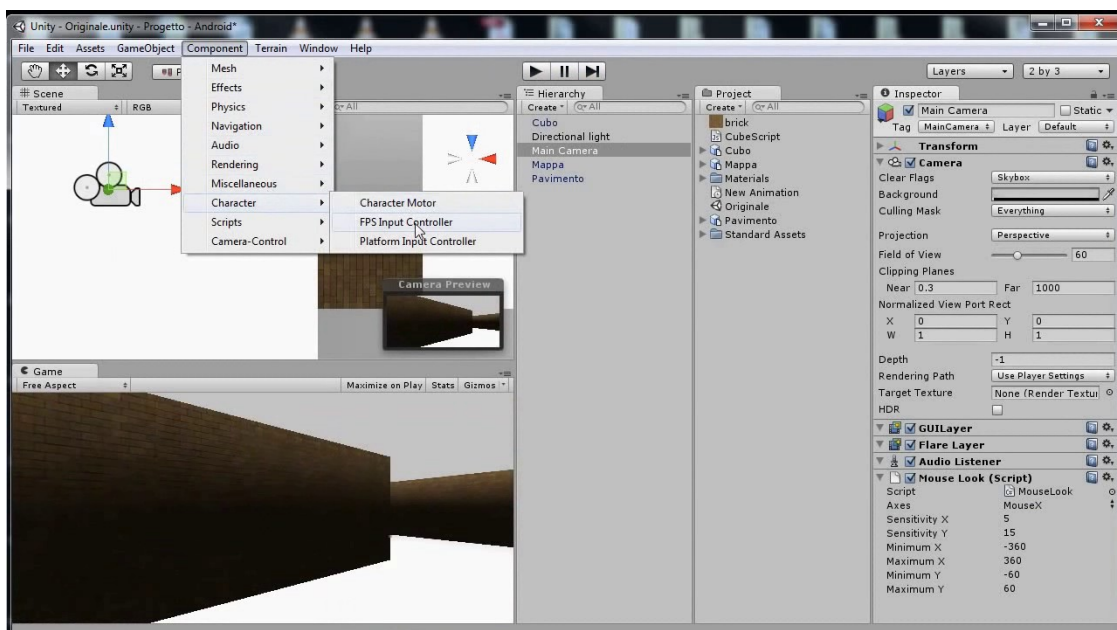
Fatto questo, con la telecamera selezionata, scegliamo **Components | Camera-Control | Mouse Look**, per iniziare dal componente più semplice, ossia l'orientamento mediante mouse.



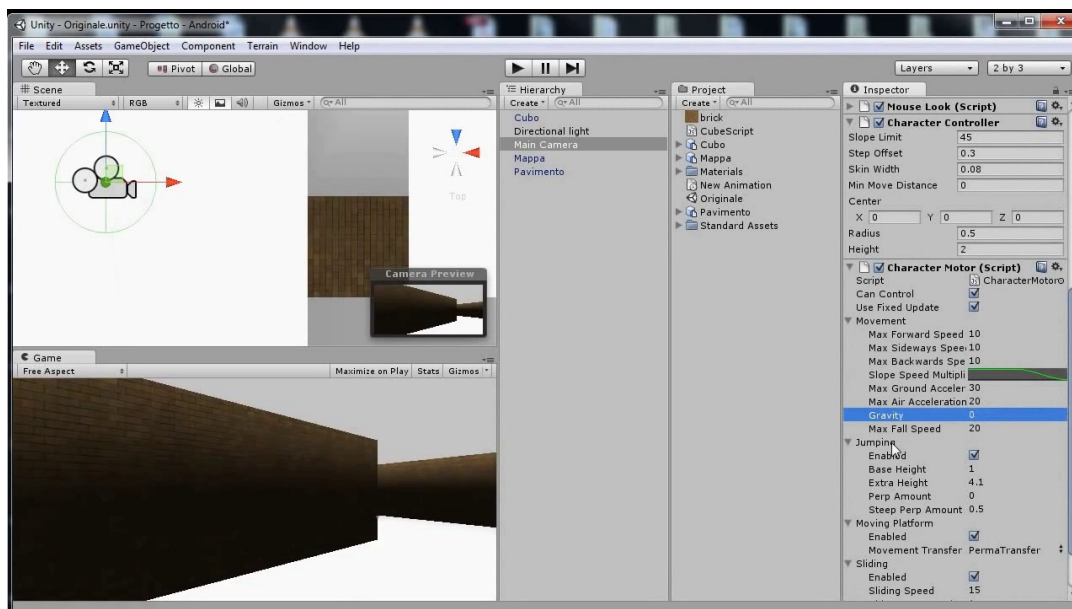
Nella scheda di tale **Component**, nell'Inspector della telecamera, scegliamo **MouseX** per **Axes**, anziché X e Y (beh, stiamo facendo Wolfenstein, non Quake!) e abbassiamo ad esempio a 5 la **sensibilità per l'asse X**.



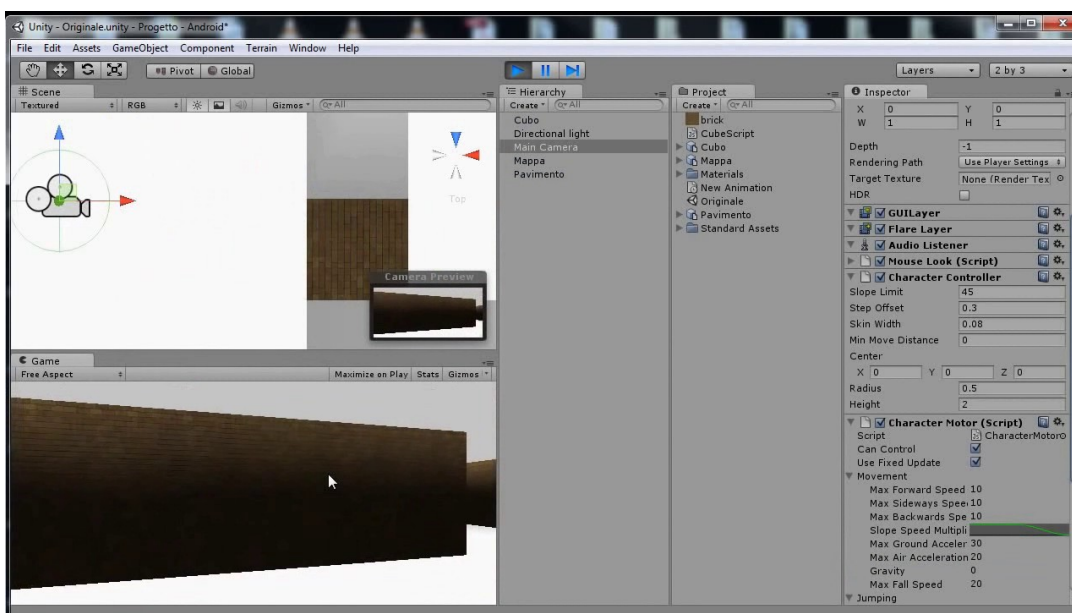
Sempre con la **Main Camera** selezionata aggiungiamo un **Component** per il movimento con **Component | Character | FPS Input Controller**.



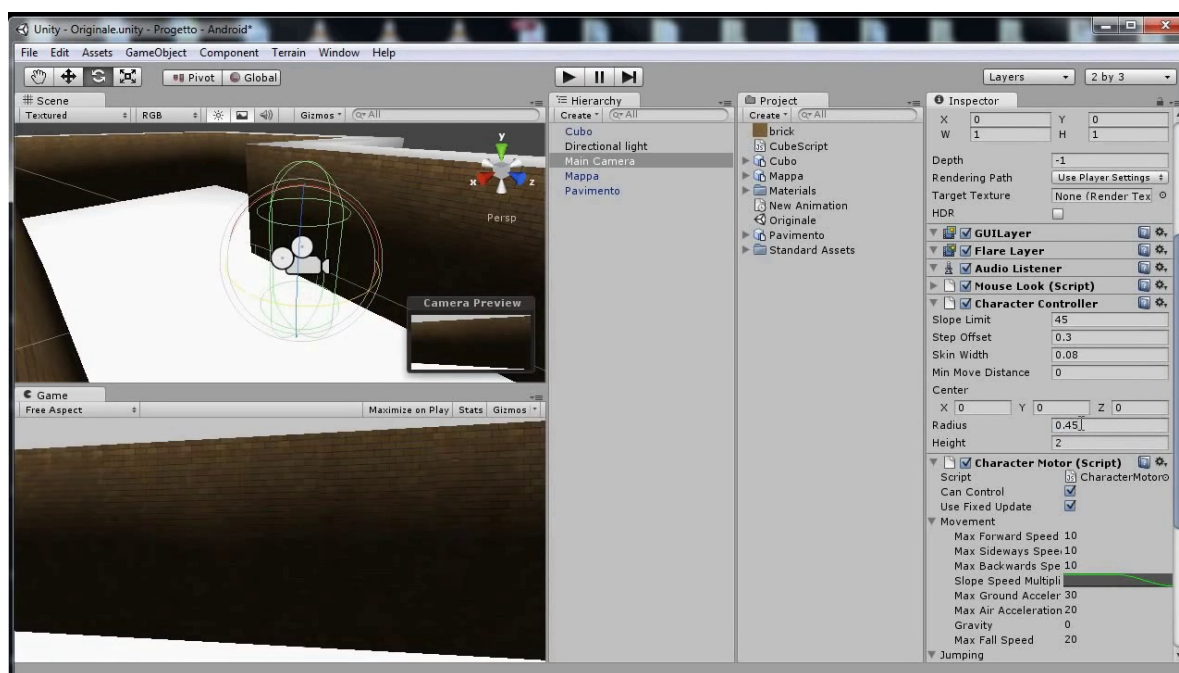
L'unica cosa da fare, per questo elemento, è **portare a 0 il parametro Gravity**, nell'Inspector: il pavimento della scena non sarà un **Rigid Body** e, dal momento che non ci sono salti o altri effetti, solo un movimento all'interno dello scenario su un solo piano, anziché impostare le proprietà fisiche di altri oggetti togliamo l'effetto della gravità per la telecamera.



Fatto questo, premiamo **Play** per verificare che sia tutto a posto con la telecamera: dovrebbe essere tutto ok, ad eccezione del fatto che possiamo passare attraverso i muri, ma questo lo risolveremo a breve; notate, tra l'altro, che nell'editor abbiamo ora una specie di **capsula** che ingabbia la telecamera: si tratta del **Collider**, di tipo **Capsule**, creato in automatico dall'**FPS Controller**, che rileva le collisioni con altri **Collider** o **Rigid Body**.



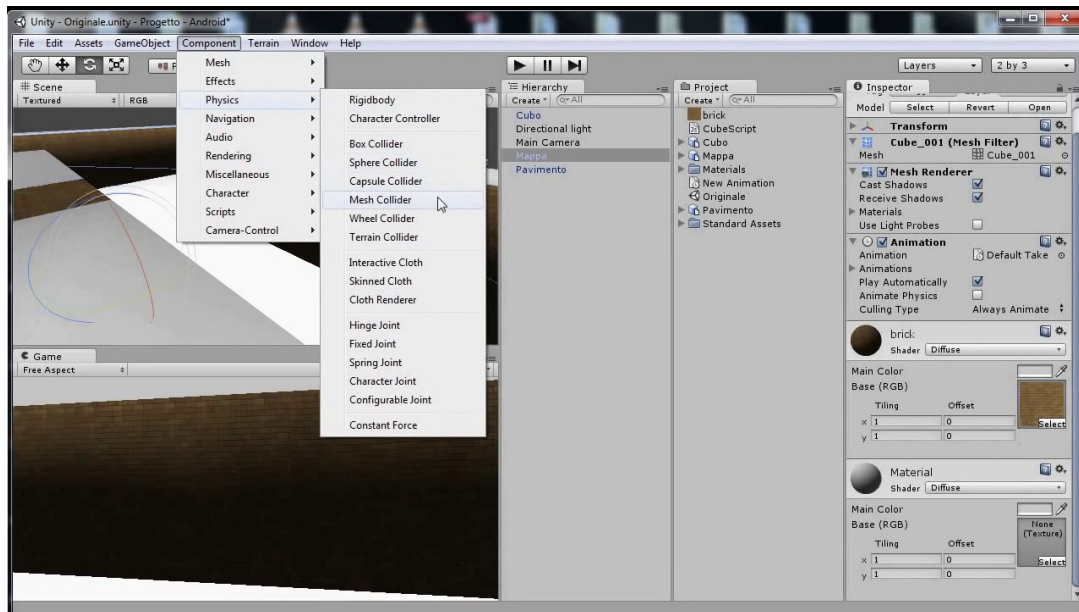
Nel fare i test, prima di realizzare il tutorial, ho visto che l'apertura nel muro, è troppo piccola per le impostazioni di default del Collider, ma per fortuna per risolvere il problema è sufficiente impostare, nel modulo **Character Controller** all'interno dell'**Inspector**, un valore leggermente più basso di quello di default per il campo **Radius**, ad esempio 0.45, scalando un po' il **Capsule Collider**, come visibile nella 3D View.



Come anticipato, per le pareti che non devono muoversi è sufficiente aggiungere un **Component** di tipo **Collider**, per implementare le collisioni con la **Telecamera**.

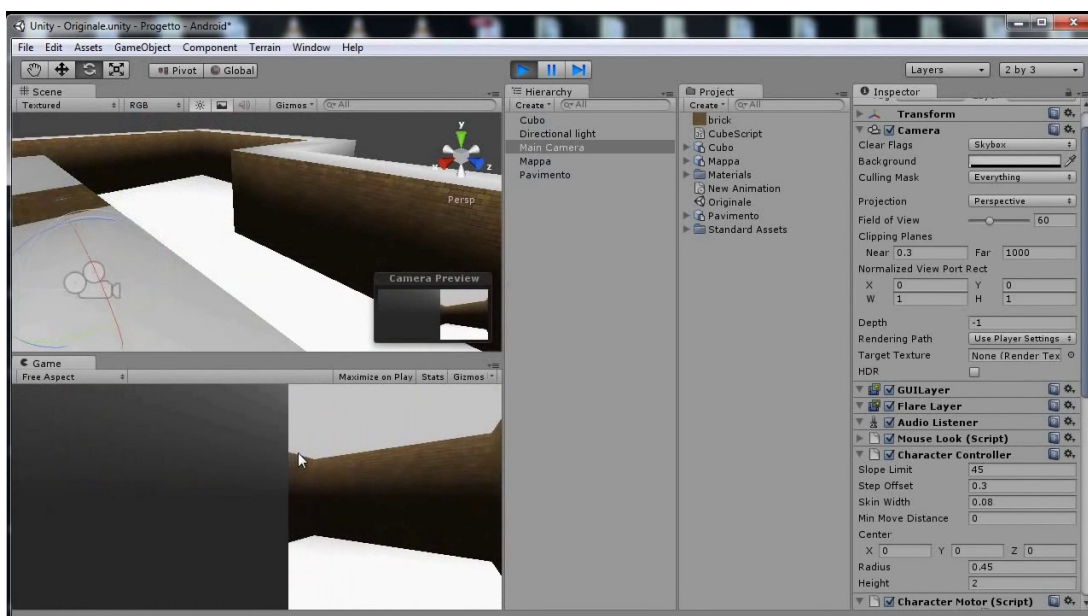
Dal momento che la **Main Camera** ha un **FPS Controller** che gestisce bene le collisioni (sostanzialmente, come se fosse un personaggio), non avremo brutti rimbalzi all'indietro, rotazioni varie e altri effetti quando toccheremo le pareti.

Selezioniamo quindi la mesh che rappresenta le pareti e scegliamo **Component | Physics | Mesh Collider**: fatto.

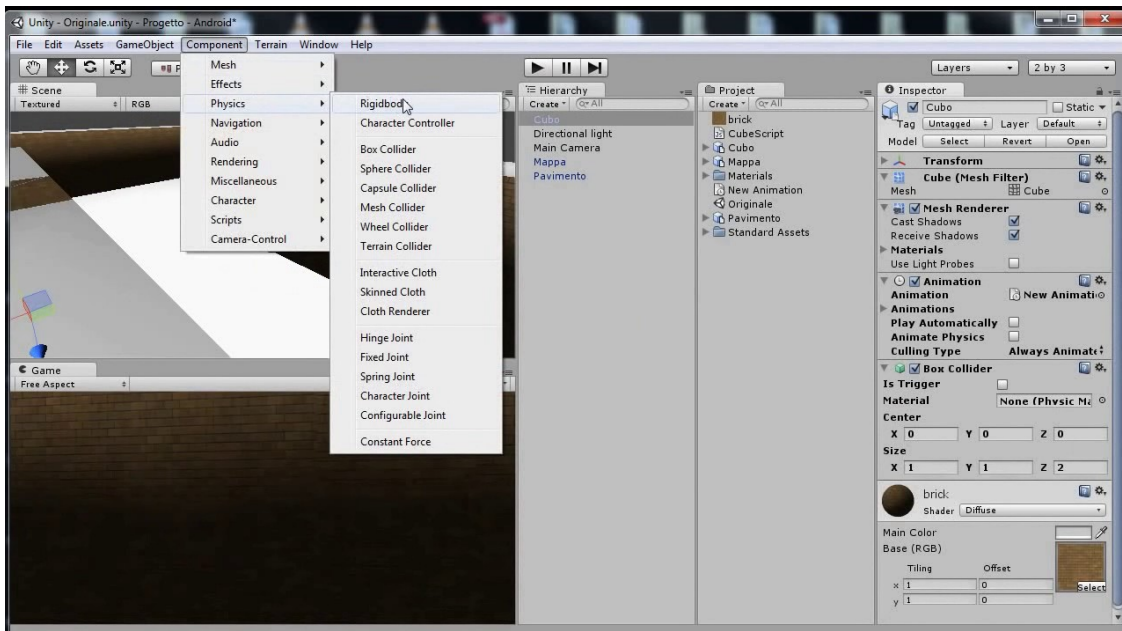
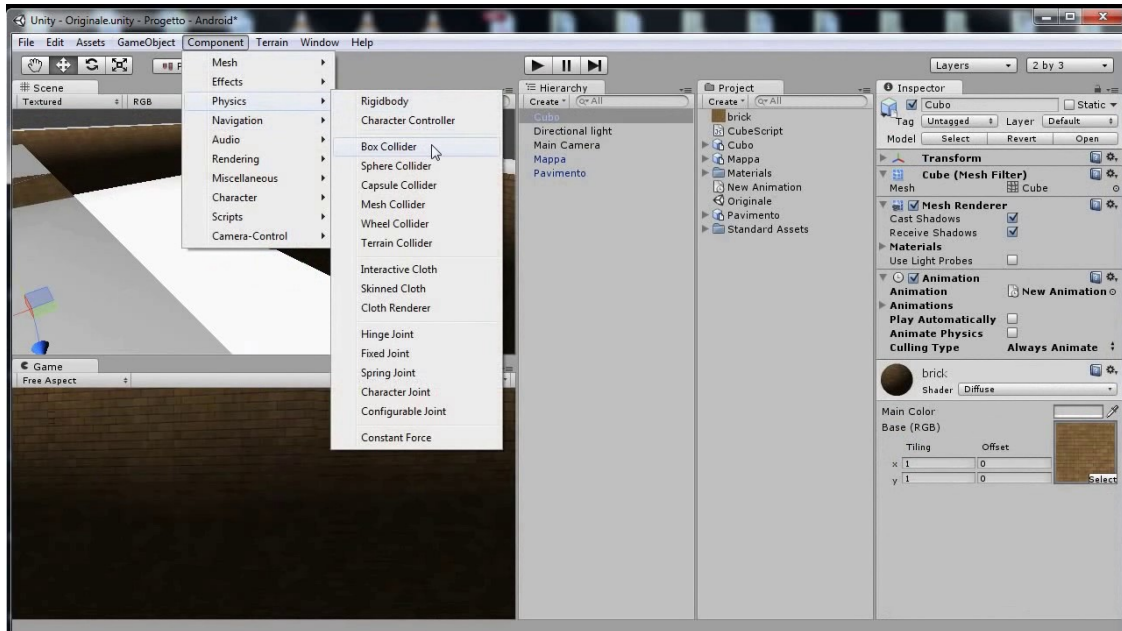


Per verificare che sia tutto ok, lanciamo di nuovo il gioco per verificare le collisioni con le pareti; faccio notare che non ci sono **Colliders** o **Rigid Body** o altro per il pavimento: per come abbiamo impostato movimenti ed effetti, **non ci servono**, e meno componenti e caratteristiche carichiamo, più fluida sarà l'esecuzione del gioco.

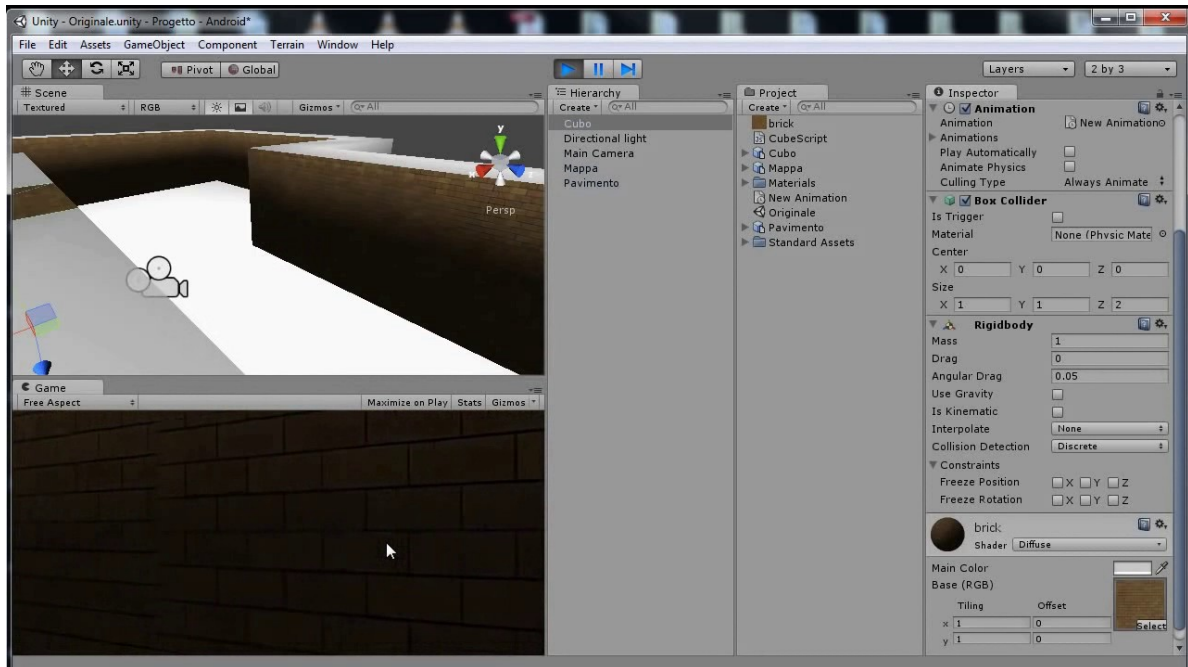
Ai lati della schermata di gioco probabilmente vedrete **attraverso le pareti**: il problema deriva dalla **dimensione del Capsule Collider**, con il raggio abbassato a 0.45... potete provare a variare le caratteristiche della telecamera (ad esempio, io ho portato il **Field of View** a 35, nell'Inspector della **Main Camera**), o altre soluzioni... oppure, potete evitare di fare un errore come il mio e progettare per bene il tutto, prima di mettere insieme i vari elementi!



Passiamo ora ai **Components “nativi”** per il muro a spinta: dobbiamo aggiungere per prima cosa un **Box Collider**, quindi con l'oggetto selezionato scegliamo **Component | Physics | Box Collider**; subito dopo, scegliamo **Component | Physics | Rigid Body**.



Come avrete intuito, nella scheda del **Rigid Body** dobbiamo deselezionare la casella **Gravity**, per evitare di far precipitare il muro attraverso il pavimento e, poi, nel vuoto, quindi deselezioniamo quella voce.



Con il setup degli elementi nella 3D View e delle loro componenti abbiamo finito.

Ora dobbiamo **creare uno script da associare al muro a spinta** per rilevare la collisione con la telecamera, verificare la pressione della barra spazio e per avviare l'animazione creata nel capitolo precedente; creiamo quindi uno script **JavaScript** nel **Progetto** e associamolo mediante trascinamento **all'oggetto del muro** a spinta, **non alla Main Camera**, quindi apriamolo in **MonoDevelop** per editarlo.

* * *

Eccoci quindi alla parte riguardante lo **script**, anche se non mancherà qualche ritocchino agli oggetti nell'editor principale.

Anziché metterci subito a programmare, **analizziamo la situazione**, visto che ci sono un paio di punti che ad una prima analisi possono passare inosservati, ma è meglio considerarli sin da subito.

Vogliamo **rilevare la collisione con altri elementi**, cosa che – ve lo anticipo subito – può essere fatta con varie funzioni native di **Unity**, come ***OnCollisionStay*** (che rileva le collisioni tra *Colliders*) e ***OnTriggerStay*** che, invece, viene attivato se un oggetto di tipo *Trigger* entra in collisione con altri elementi.

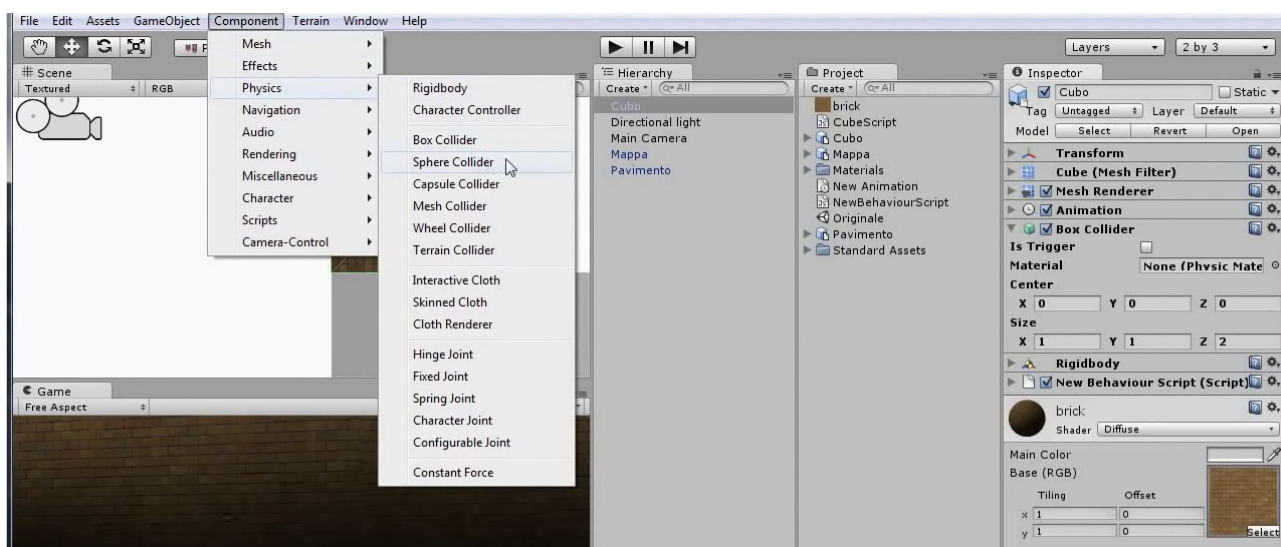
Si tratta comunque di **funzioni che vengono chiamate** automaticamente da **Unity** in caso di collisione.

OnTriggerStay funziona se l'oggetto che la possiede ha un *Collider* – e questo il nostro muro a spinta ce l'ha – ma tale *Collider* deve avere la **proprietà *isTrigger*** attivata, nell'Inspector; il problema è che attivando ***isTrigger*** escluderemo l'oggetto dalla **simulazione fisica**, quindi la telecamera potrà passargli attraverso!

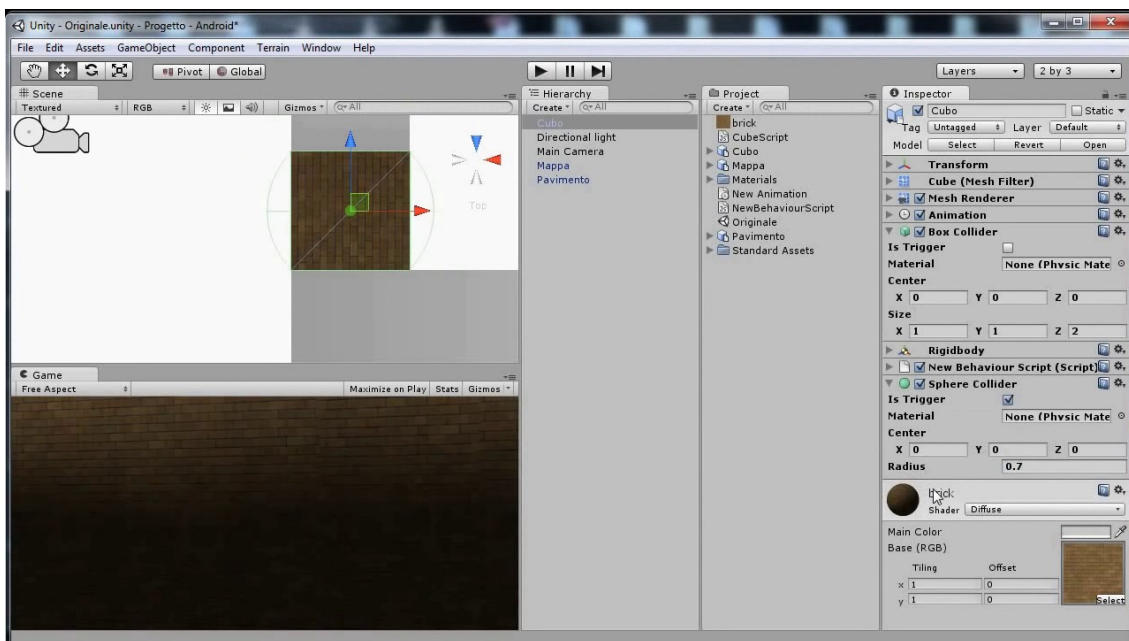
Possiamo aggiungere un secondo *Collider* al muro a spinta ma deve essere **di tipo diverso da Box Collider**, perché un oggetto non può avere più *Colliders* dello stesso tipo.

È una soluzione orrenda, per niente elegante, ma rilevare e gestire collisioni ed eventi con l'**FPS Controller** nativo, i **Collider** e i **Rigid Body** non è per niente semplice, quindi accontentiamoci.

Torniamo nell'Editor della scena di Unity, selezioniamo il muro a spinta e aggiungiamo un componente ***Sphere Collider*** con *Component | Physics | Sphere Collider*.



Nell'Inspector, impostiamo un **Radius** che possa far uscire il **Collider** dall'oggetto, in modo da farlo toccare con il **Collider** nativo del **Character Controller**, ad esempio 0.7 (osservate i risultati con i vari valori con la vista interattiva, nella 3D View, fino ad arrivare ad un valore che vi soddisfa), dopodiché selezioniamo la casella **isTrigger**, che renderà attive le funzioni **OnTriggerEnter**, **Stay** ed **Exit** del Collider, escludendo però le simulazioni fisiche (quindi non ci saranno “impatti” con questa sfera).



Adesso il muro a spinta rileva le collisioni degli oggetti che entrano nel **Trigger**; il problema è che già così ci sono **almeno due elementi** in collisione permanente col muro, ossia le altre pareti e il pavimento, quindi **OnTriggerStay** è sempre valido!

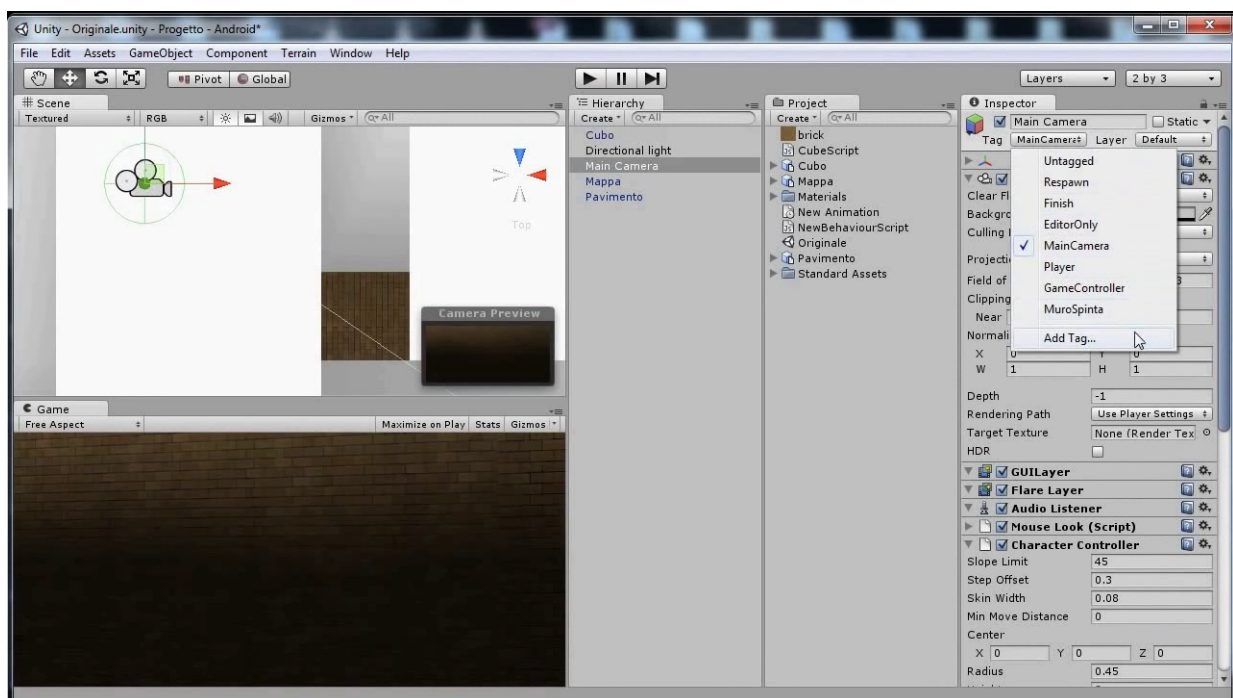
Dobbiamo quindi rilevare la collisione “filtrando” solo il contatto con il **Character Controller** della **Main Camera**... non solo: come secondo punto, dobbiamo far eseguire l'animazione solo se non è stata già eseguita.

Il primo punto può essere risolto in molti modi ma, per farvi conoscere un nuovo argomento, useremo i **tag**, mentre per il secondo punto basterà usare una **variabile flag**, ossia una variabile che ha lo scopo di memorizzare se una condizione si è già verificata o meno.

I **tag** sono, come dice il nome, dei “contrassegni” da dare agli oggetti, delle proprietà o caratteristiche extra.

Servono per filtrare gli elementi presi in considerazione, ad esempio nel caso della **collisione**: se un oggetto ha un **tag** specifico, possiamo dire alla funzione di collisione “se sei entrata in contatto con un oggetto provvisto di tale tag fai questo, altrimenti niente”.

I **tag** vanno specificati, per ogni oggetto, in alto nell'Inspector; ovviamente, possiamo creare **tag** personalizzati, ma per certi elementi, come la telecamera, ci sono dei **tag** di default, infatti guardando nell'Inspector dell'oggetto **Main Camera**, in alto, troviamo selezionato il **tag** “**MainCamera**” (e cliccando sul menù vediamo che possiamo aggiungerne altri con “**Add Tag**” e il gestore dei **tag**, ma questo è un altro discorso e lo tratteremo un'altra volta; qui ci va bene sapere che la telecamera ha il tag “**MainCamera**” di default).



Abbiamo quindi visto che **la telecamera ha un tag specifico** e abbiamo creato un nuovo **Collider** di tipo **Trigger** per il **muro a spinta**.

Possiamo tornare a **MonoDevelop**.

All'inizio dello script, che ricordo essere **associato al muro a spinta**, fuori da ogni funzione dichiariamo e inizializziamo la variabile *flag* che indica **se l'animazione non è stata ancora eseguita**, scrivendo:

```
var animMaiEseguita : boolean = true;
```

La funzione ***OnTriggerStay*** prende come parametro un oggetto di tipo *Collider* nel quale depositerà **informazioni sull'elemento** che è entrato in collisione.

Questa funzione viene **invocata ad ogni frame** durante il quale si verifica una **collisione** e, poiché come abbiamo visto il muro è sempre in collisione con altri elementi, vi suggerisco di provare altre soluzioni, ad esempio definire dei ***Colliders*** con altre forme, oppure se proprio non è possibile eliminare le collisioni con gli altri elementi... “andateci piano”, per così dire, con l'inserimento di muri a spinta nel gioco, visto che a ogni frame il programma verificherà le collisioni (inutilmente, tra l'altro).

Definiamo quindi la funzione nello script scrivendo:

```
function OnTriggerStay(col : Collider)
{
}
```

Adesso si tratta di **“filtrare” l'oggetto in collisione** basandoci sul suo **tag** e, se viene premuta la **barra spazio** e se l'animazione non è **mai stata eseguita**, avviarla.

Nel corpo della funzione, scriviamo quindi:

```
if ( (col.gameObject.tag=="MainCamera") && animMaiEseguita &&
Input.GetKey(KeyCode.Space) )
{
    gameObject.animation.Play();
    animMaiEseguita = false;
}
```

Quindi: entriamo nel blocco ***if*** solo se si verificano le seguenti tre condizioni:

- se l'oggetto col quale entriamo in collisione è provvisto del **tag** di nome ***MainCamera***, rilevando l'oggetto con ***col.gameObject*** e il suo **tag** con ***.tag*** (*col* è ovviamente il Collider parametro della funzione ***OnTriggerStay***);

- se **l'animazione non è mai stata eseguita** (scrivere, in un *if*, il solo nome di una variabile booleana, è una scorciatoia per chiedere “*se questa variabile è vera*”);
- infine, se l'utente **sta premendo la barra spazio** (può anche camminare e tenerla premuta: l'evento verrà rilevato, come avveniva in Wolfenstein 3D scivolando lungo le pareti con la barra spazio premuta).

Da notare che ***animation.Play*** è una funzione che accede al blocco ***Animation*** dell'oggetto di gioco e avvia l'animazione presente con ***Play***; ovviamente, possono esserci **più animazioni** in un blocco ***animation***, ad esempio un ciclo di camminata, un salto, eccetera, per cui la funzione ***Play*** può prendere come parametro **il nome dell'oggetto Project Animation** da eseguire, ma visto che qui ce n'è uno solo (creato nel capitolo precedente) possiamo scrivere ***Play*** senza parametri.

Non ci metto la mano sul fuoco, ma se il compilatore degli script di ***Unity*** è buono, e suppongo chiaramente che sia così, allora in realtà **non ci saranno tre controlli a vuoto**, a ogni frame, ma solo il primo, perché molti compilatori fanno il cosiddetto **cortocircuito dell'if con AND**: se la prima condizione, che qui è il contatto con un oggetto con **tag Main Camera**, non viene verificata, non si controllano le altre condizioni in **AND**, perché **basta un false** per rendere **false** tutta la catena di **AND**.

Salviamo lo script per applicare le modifiche, torniamo all'editor principale, premiamo ***Play*** e spostiamoci un po' in giro, quindi quando siamo a contatto col muro a spinta **premiamo la barra spazio** per osservare il risultato.

* * *

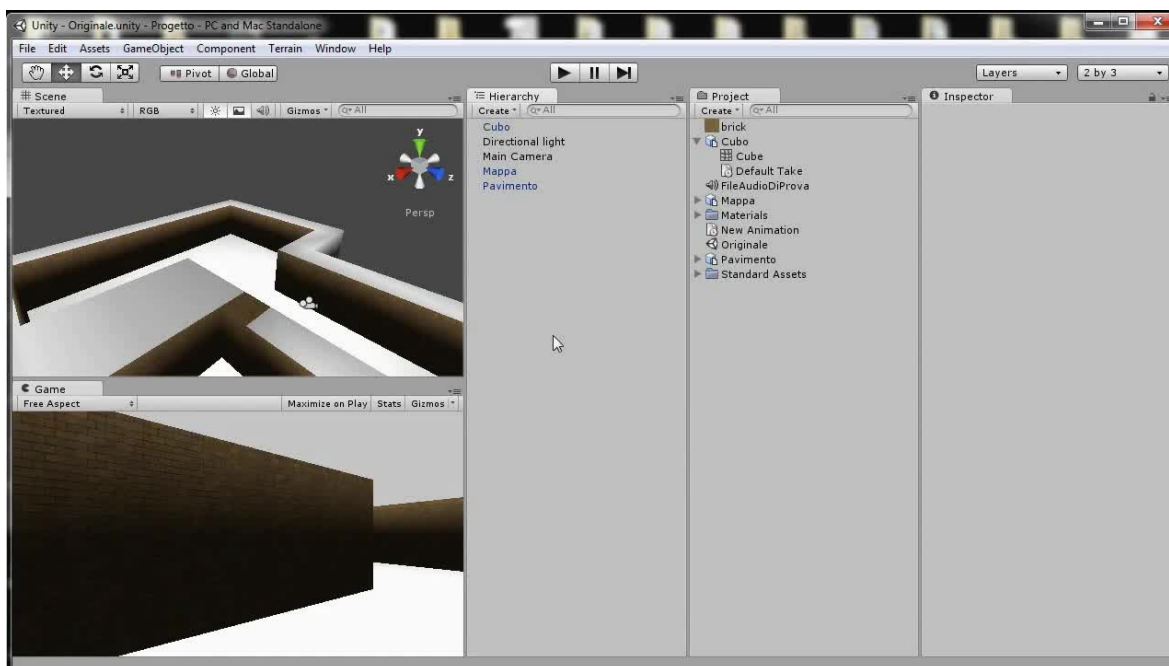
Come potete vedere, a livello di script non c'è molto da dire: in questo capitolo vi ho fatto conoscere la **funzione OnTriggerStay** e l'utilizzo dei tag per rilevare e filtrare le collisioni, quindi il discorso riguardava più che altro riuscire a trovare **una soluzione tra le varie incongruenze e incompatibilità** di *Colliders*, *Controllers* e *Rigid Body*.

* * *

Riprodurre dei file audio in seguito a delle collisioni (triggering)

In questo tutorial, nato dalla richiesta di un utente del Forum *effel2.com*, vedremo **come creare fonti sonore e riprodurre** l'audio a comando in tempo reale, **durante l'esecuzione** del gioco, in **Unity 3D**, in seguito a un **contatto** o collisione della telecamera con una determinata area del gioco.

Per i nostri esempi utilizzeremo la scena già vista nei due tutorial sui **Muri a Spinta in Unity 3D**, per cui – dopo aver importato il **Package Character Controller** da *Assets | Import Package*, bisogna dare alla telecamera un **Component | Character | FPS Input Controller** (che fornirà vari componenti alla telecamera, tra cui un **Capsule Collider**, che useremo in questo esempio per rilevare una collisione e attivare un evento), poi dovremo dare alla telecamera un **Component | Mouse Look**, sul solo **asse X**, per poter utilizzare il mouse per cambiare inquadratura; infine, bisognerà impostare **Gravity a 0** nella sezione **Character Motor Component** della telecamera, nell'Inspector, per non far cadere l'oggetto del vuoto (in questa sede, visto che non sono previsti salti, rimbalzi o altri effetti fisici, non c'è bisogno di rendere il pavimento un Rigid Body).



Fino a questo punto tutto è simile a quanto visto nei capitoli sui “muri a spinta”, salvo che lì abbiamo impostato anche l'animazione, in base a contatto e pressione di un tasto, di un pezzo di muro, cosa che qui non faremo, per cui in questo momento la telecamera può passare attraverso le pareti; comunque, qui non importa: ci interessava dare alla telecamera i **controller** di movimento per spostarci nella scena e parlare degli audio clip.

In **Unity**, la riproduzione di un **clip audio** (un file audio, in uno dei tanti formati supportati da **Unity**, come l'MP3, nel mio caso) è affidata a un **Component** detto **Audio Clip**, da associare quindi ad un oggetto.

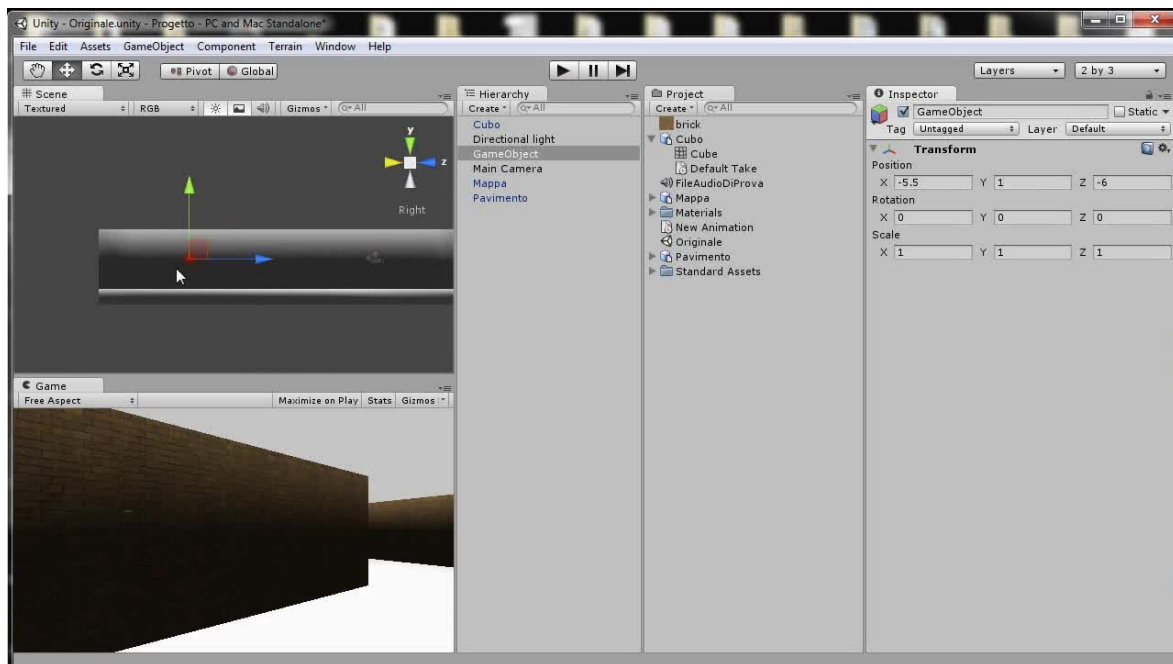
In questo particolare caso daremo l'**Audio Clip** a un oggetto **Empty** posizionato in un certo punto della scena; questo perché il tutorial nasce da una richiesta specifica: riprodurre l'audio quando il personaggio si avvicina ad un determinato oggetto.

Audio Clip è un **Component**, quindi può essere fornito a una qualsiasi mesh (ad esempio, una statua, in un museo virtuale, per far partire l'audio con il commento sonoro, oppure al fucile del protagonista, così quando sparerà verrà riprodotto il relativo effetto sonoro); io, con uno sforzo praticamente nullo di fantasia, userò una generica **Empty**: un oggetto invisibile.

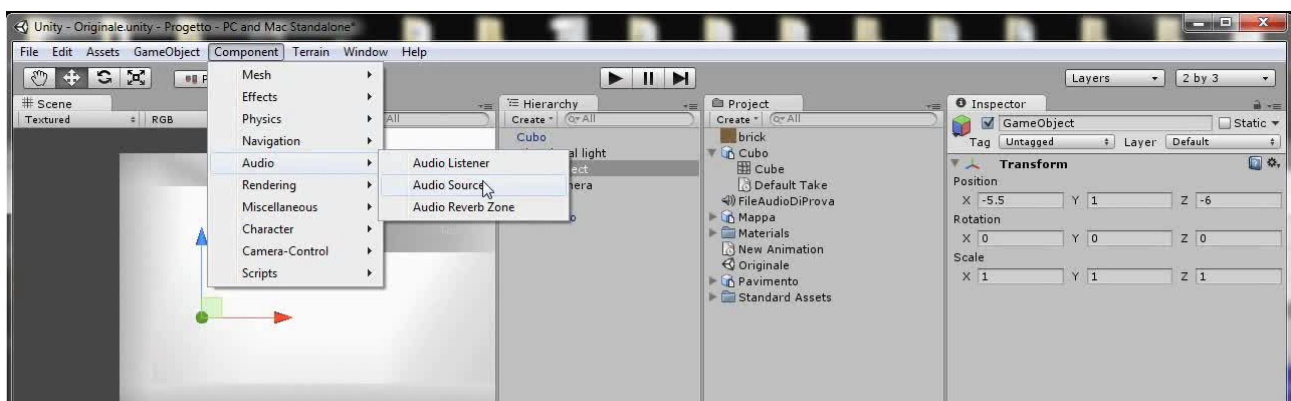
Le istruzioni **JavaScript** che consentono di riprodurre l'audio sono semplicissime ed anzi, in realtà, basta **una** riga di codice, come vedremo; se dovete riprodurre l'audio di uno sparo allora non vi serve molto altro, ma qui, come detto prima, vogliamo riprodurre l'audio quando il personaggio si avvicina ad un oggetto, quindi ci serve qualcos'altro.

“*Si avvicina ad un oggetto*” può essere letto anche “*entra in una determinata porzione di spazio*”, il che significa: **attiva un collider**, con “attiva” che in Inglese si dice “**triggers**”... se ricordate i tutorial sui muri a spinta, avete già capito qual è la soluzione: **ingabbieremo la Empty** (che rappresenta la statua, il quadro, l'idolo da raccogliere, qualsiasi cosa) in un **Box Collider** con opzione **isTrigger** attiva (in modo da escluderlo dalla simulazione fisica della scena, per cui potremo passarli attraverso) e, **in caso di collisione con la telecamera** (che rileveremo usando il tag “**MainCamera**”, anche questo visto nei due tutorial sui muri a spinta), **faremo partire l'audio**.

Per prima cosa, quindi, **aggiungiamo un oggetto *Empty***; in questa particolare scena, sto digitando valori precisi per posizionarlo in un angolo (-5.5, 1, -6); in particolare, la *Empty* dev'essere comunque sullo stesso piano del giocatore; non fatevi cioè ingannare da un posizionamento dalla vista superiore, perché potrebbe essere sopra o sotto il livello di gioco: **controllate sempre** con almeno una vista laterale!

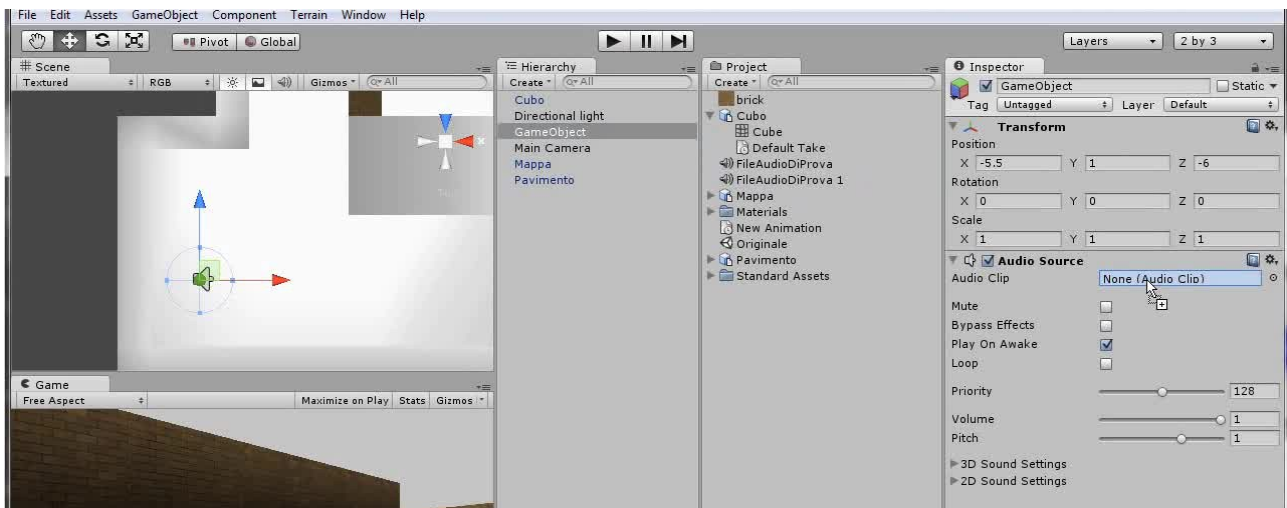


Diamo alla *Empty* un **Component | Audio | Audio Source** (immagine seguente), inizialmente senza clip associata, ovviamente; va detto che per sentire l'audio, **la telecamera** deve avere un **Component | Audio | Audio Listener** (ascoltatore), ma la *Main Camera* ne ha uno di default, come visibile nell'Inspector di tale oggetto.

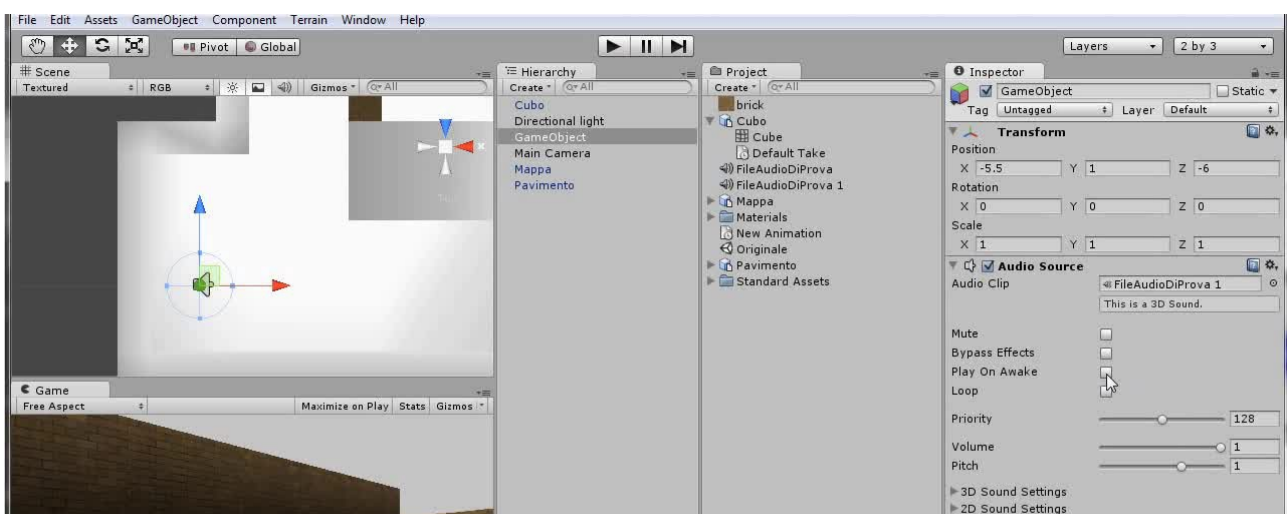


Importiamo quindi il file audio che vogliamo riprodurre nel progetto, **trascinandolo** da una cartella nella sezione **Project** dell'Editor oppure caricandolo con **Import Assets**, rendendolo così a tutti gli effetti un oggetto del progetto!

Impostiamo tale oggetto come **clip audio dell'Audio Source dell'Empty** mediante semplice trascinamento dell'audio clip dal **Project** allo slot **Audio Source** nell'Inspector.

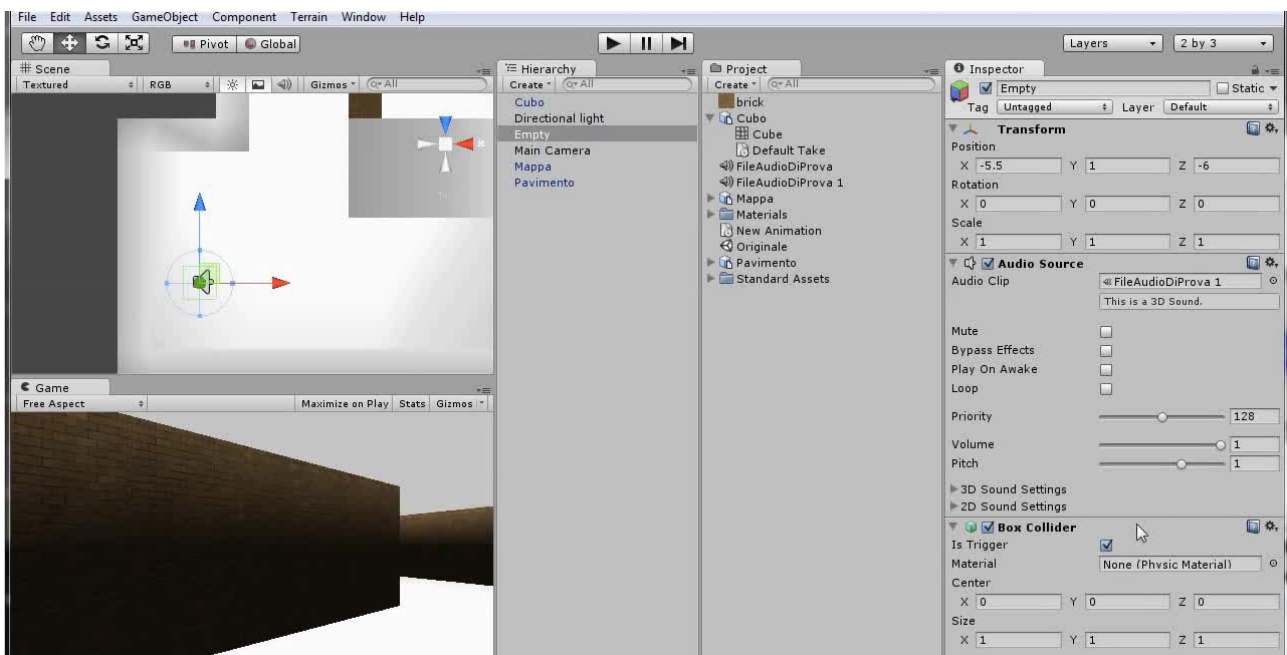
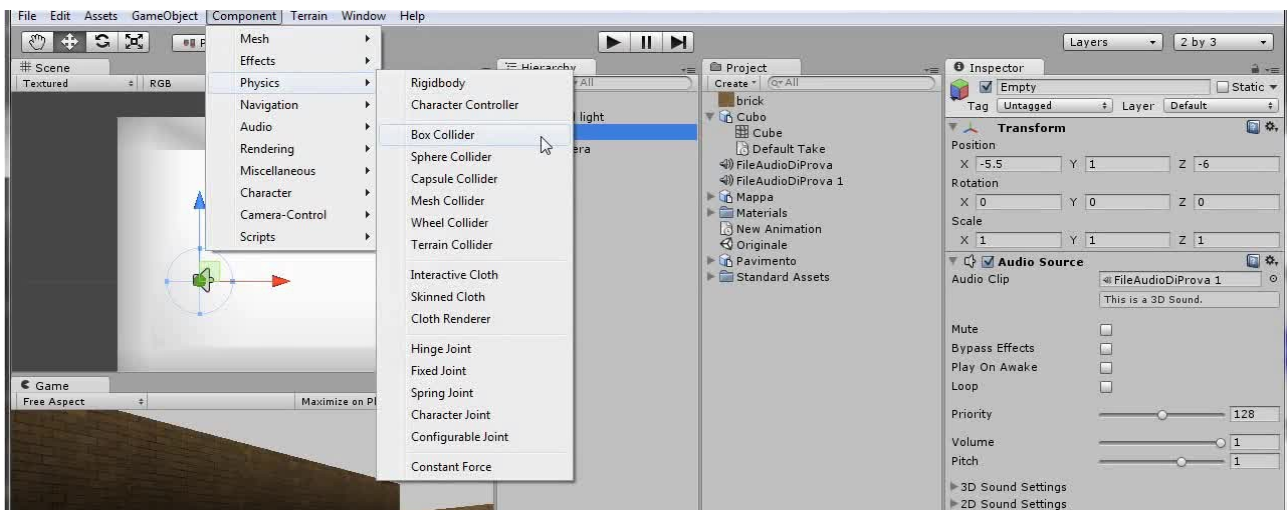


Come prima cosa, **deselezioniamo** la casella **“Play on Awake”** per il **Component Audio Clip**, che altrimenti farebbe partire la riproduzione del file audio all'avvio del gioco; noi, invece, vogliamo **farla partire a comando**, in base a un evento, via codice.



L'evento, in particolare, è l'ingresso della **Main Camera** (che, come si può verificare in alto nell'Inspector, ha di default il tag “*MainCamera*”) in un'area di attivazione, che dobbiamo ancora creare.

Con la **Empty** selezionata e già provvista dell'**Audio Source**, aggiungiamo un componente **Box Collider**, con l'opzione **isTrigger** selezionata: questa opzione esclude il **Box Collider** dalla simulazione fisica e ci consente di utilizzare la funzione nativa **OnTriggerEntry**, che si attiva se un oggetto **Collider** (come l'**FPS Controller** della **Main Camera**, che di default ha un **Capsule Collider**) entra all'interno del **Box Collider**.



Creiamo quindi uno script **JavaScript**, con nome a piacere, nel **Project**, quindi associamolo alla **Empty** (che, ricordiamo, possiede già **Audio Source** e **Box Collider**, quest'ultimo scalato a piacere per definire lo spazio di attivazione), quindi apriamo tale script in **MonoDevelop** per digitare le istruzioni necessarie.

Tutto si svolge nella funzione **OnTriggerEnter**, nativa di **Unity** e che viene attivata automaticamente quando un oggetto entra nel **Collider** in modo **isTrigger**, per cui creiamo tale blocco:

```
function OnTriggerEnter(col:Collider){  
}
```

facendo attenzione a O, T, E e C maiuscole (l'oggetto col sarà l'oggetto che entrerà in collisione con la **Empty**); quindi, all'interno di tale blocco, apriamo un blocco **if** che verifica se la **collisione** avviene con un oggetto con **tag “MainCamera”**, come la nostra telecamera, con:

```
if(col.gameObject.tag=="MainCamera")
```

quindi, all'interno di tale **if** (anche senza parentesi graffe, visto che si tratta di una sola istruzione), scriviamo l'unica istruzione che serve per eseguire la clip audio associata:

```
audio.Play();
```

fatto, abbiamo finito! **“audio”** è quindi un oggetto nativo di **Unity** che si riferisce alla clip **Audio Source** associata all'oggetto che possiede lo script ed è sufficiente chiamare la **funzione Play** di tale oggetto **per eseguire la clip audio!**

Potete quindi **salvare lo script** (per rendere effettive le modifiche, altrimenti non succederà nulla) e **far partire il progetto**: quando entrerete nell'area di **collisione**, verrà **riprodotto** il file audio, che tra l'altro verrà riprodotto fino alla fine anche uscendo da quell'area e, nel caso di suono stereo o “3D”, lo sentirete **con diverse intensità ai lati** a seconda del vostro **orientamento** rispetto alla fonte sonora (provate a ruotare il personaggio sul posto durante l'esecuzione del file audio).

Le basi della riproduzione di un file audio si riducono quindi a **audio.Play()**; è vero che esistono altri campi per il componente **Audio**, come controlli su **Volume**, **Pitch**, eccetera, ma “scoprirli” è facile grazie alla documentazione online di **Unity**, quindi vi invito a esplorare le varie opzioni, se

avete bisogno di un maggiore controllo; qui, comunque, voglio darvi una dritta sul **cambio di file audio da riprodurre** con la stessa fonte.

Può capitare, infatti, di dover **associare file audio diversi ad un'unica sorgente**: nel museo virtuale, dopo aver eseguito il primo file audio, alla pressione di un oggetto si potrebbe passare a un secondo file audio, e così via, mentre in un videogioco potremmo creare un solo **Audio Source**, da legare al personaggio, per il rumore dello sparo, però potremmo dover riprodurre file audio diversi a seconda dell'arma usata (sconsiglio di creare un **Audio Source** per arma: meglio un **Audio Source per giocatore**, cambiando però la **clip** audio da riprodurre).

Quello che dobbiamo fare, in questi casi, è definire una **variabile globale** (all'inizio dello script, fuori da qualsiasi funzione) di tipo **AudioClip** (con A e C maiuscole), o magari **un intero array** di variabili **AudioClip** nel caso dovessero essere tante (soluzione presente nei capitoli relativi al *Car Tutorial*), ad esempio:

```
var clip1 : AudioClip;  
var clip2 : AudioClip;
```

e così via, quindi **associare vari file audio dal Project agli slot** di queste variabili nell'Inspector, mediante drag-and-drop, come abbiamo fatto all'inizio di questo tutorial; a questo punto, **per cambiare il clip audio da riprodurre** scegliendolo tra queste variabili, è sufficiente scrivere:

```
audio.clip = [nome variabile AudioClip da riprodurre]
```

cosa che va fatta con degli **if** o altri strumenti; ad esempio, nella funzione di riproduzione dell'audio, potremmo scrivere:

```
se [if] l'arma selezionata è una pistola allora usa l'audio clip clip1 con  
audio.clip = clip1;  
se [if] l'arma selezionata è un BFG 9000 allora usa l'audio clip clip2 con  
audio.clip = clip2;
```

e, dopo aver impostato il clip, scrivere semplicemente:

```
audio.Play();
```

che riprodurrà il clip impostato.

Ricapitolando: per riprodurre un file audio è necessario aggiungere un componente **Audio Source** specificando il clip audio (da caricare come asset di gioco nel **Project**) e, nello script dato all'oggetto che deve riprodurlo, digitare **audio.Play()**; nel nostro particolare caso, questa istruzione si trova in **OnTriggerEnter** se l'oggetto che entra in collisione con il **Box Collider** in modo **isTrigger** è una telecamera, ossia ha il tag “**MainCamera**”.

Da notare che potete comandare la riproduzione dell'audio **anche per un oggetto diverso**, in quanto è sempre possibile, per uno script, accedere ad un altro **Game Object** (ad esempio con **Find**), quindi accedere a un componente di quel **Game Object** (nel nostro caso, la componente audio) e alle sue funzioni (in questo caso, **Play()**); nei capitoli precedenti abbiamo visto più volte come accedere, via script, ad altri oggetti di gioco e alle loro proprietà o componenti.

È possibile **scegliere l'audio clip** tra variabili di script con **audio.clip** e in generale sono disponibili altri campi e metodi per operazioni più avanzate, che però non vedremo in questa sede (ma potete esplorare voi stessi la documentazione online).

* * *

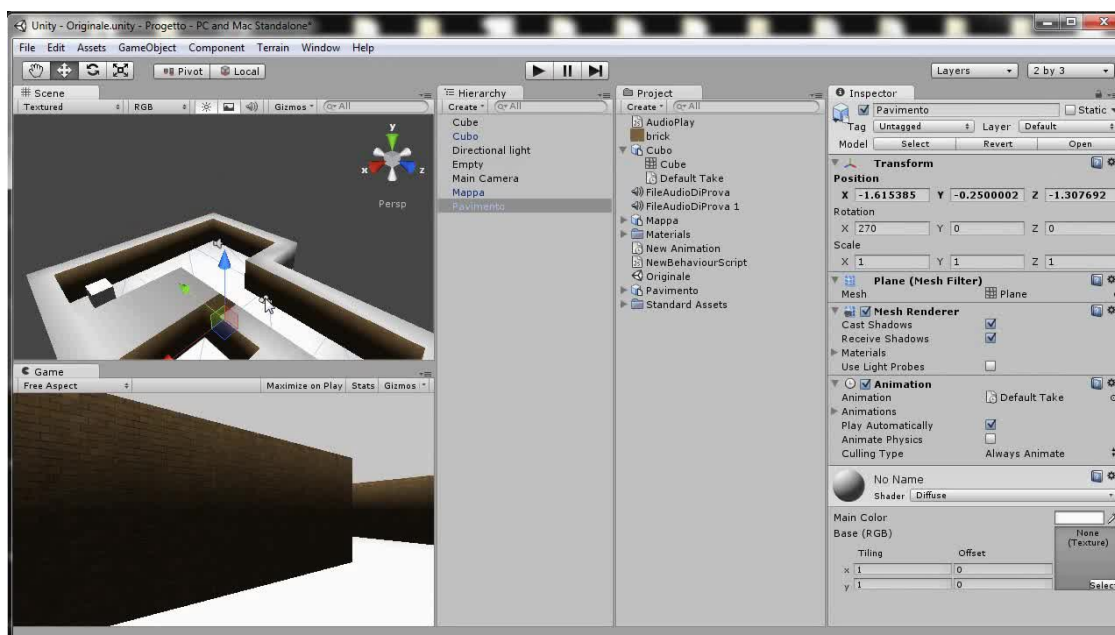
Un piccolo esercizio : ricordate il tutorial del corso base nel quale venivano sparati dei proiettili da un “cannoncino” e tali proiettili colpivano dei Box soggetti a collisioni? Riprendetela e provate ad aggiungere degli effetti sonori in base ai vari eventi, ad esempio lo sparo alla pressione della barra spazio oppure la collisione in caso di contatto tra proiettili e bersaglio.

* * *

Come capire se un oggetto è inquadrato dalla telecamera

In questo brevissimo tutorial vedremo un modo **per capire se un oggetto** (ed eventualmente quale) **è nel campo visivo della telecamera** o di un altro oggetto di gioco in una scena **Unity 3D** mediante **Raycast** di **Physics**.

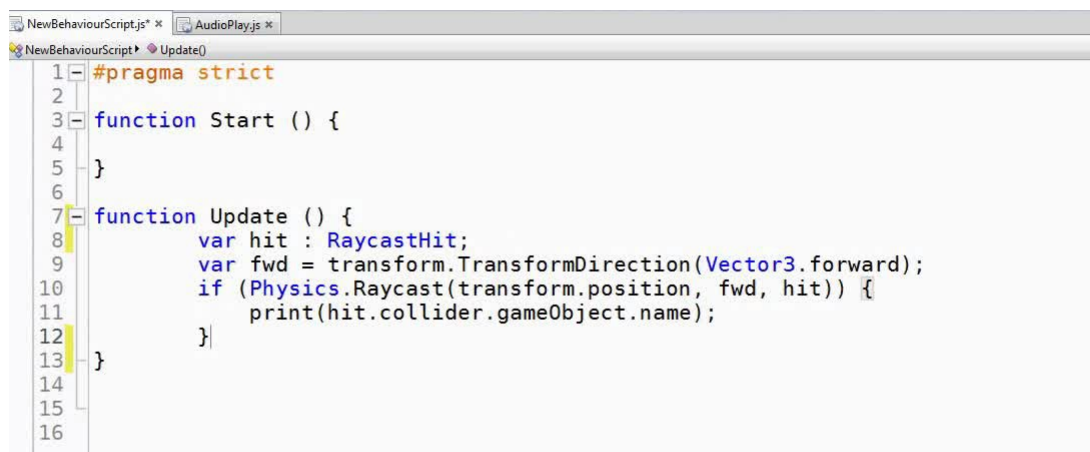
Può capitare di dover determinare se un oggetto è attualmente inquadrato dalla telecamera oppure se si trova di fronte a un altro oggetto (ad esempio, se si progetta un nemico, questi può “accorgersi” della nostra presenza se ci vede).



In entrambi i casi, si ricorre a **Physics.Raycast**, che proietta un raggio in grado di **intercettare i Colliders** che ha davanti e di determinare il nome e altre caratteristiche (come i **tag**) dei **Game Objects** che li possiedono, per cui, tornando all'esempio del nemico, è possibile distinguere tra alleati o avversari, giusto per fare un esempio, e in generale è possibile capire cosa si sta guardando.

Abbiamo parlato di **Physics.Raycast** in uno dei dieci capitoli sulle basi di **Unity 3D**, in particolare per rilevare l'oggetto cliccato mediante tocco su dispositivi **touch**: in quel caso, si convertivano le **coordinate del dispositivo** in coordinate di scena, si proiettava un raggio e si vedeva qual era l'oggetto colpito; quella che vedremo qui è una **versione semplificata**, ma in quel caso non si parlava del **TransformDirection**, che stiamo per trattare.

Una delle “firme” del metodo nativo **Physics.Raycast** è: **vettore 3D di origine**, **vettore 3D di destinazione**, **oggetto RaycastHit**; grazie a quest'ultimo possiamo capire qual è l'oggetto intercettato, ma prima dobbiamo parlare dei due vettori 3D.



```
1 #pragma strict
2
3 function Start () {
4
5 }
6
7 function Update () {
8     var hit : RaycastHit;
9     var fwd = transform.TransformDirection(Vector3.forward);
10    if (Physics.Raycast(transform.position, fwd, hit)) {
11        print(hit.collider.gameObject.name);
12    }
13 }
14
15
16
```

Il primo **vettore 3D** è l'**origine del raggio**: intuitivamente, corrisponde alla telecamera o comunque all'oggetto che lo possiede, per cui è sufficiente usare **transform.position**.

Per il **vettore 3D di destinazione**, invece, **Unity 3D** ci viene in aiuto con:

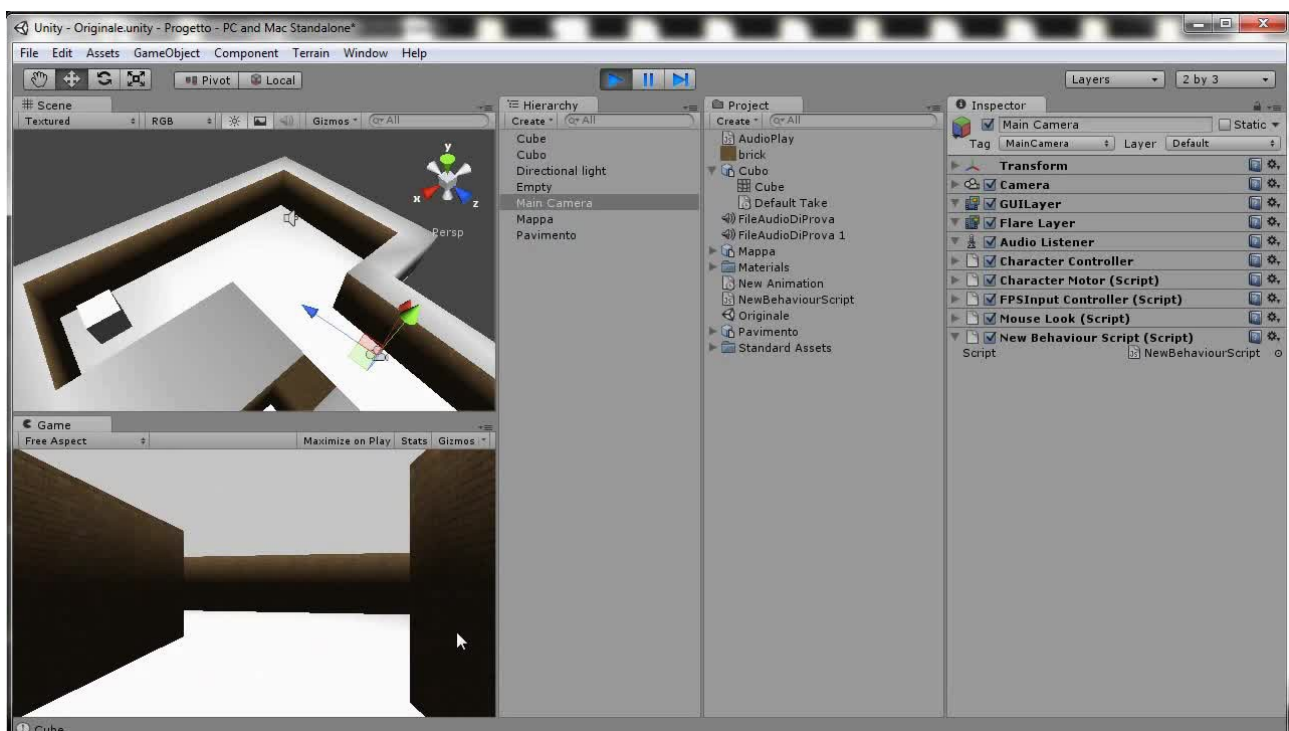
```
transform.TransformDirection(Vector3.forward)
```

ossia la direzione “**in avanti**” rispetto all'oggetto che lo possiede.

Tutte le istruzioni che ci servono vanno messe, tipicamente, in **Update**, in quanto tale funzione viene richiamata automaticamente ad ogni frame di gioco da **Unity**; scriviamo quindi in tale blocco:

```
var hit : RaycastHit;
var fwd = transform.TransformDirection(Vector3.forward);
if (Physics.Raycast(transform.position, fwd, hit)) {
    print(hit.collider.gameObject.name);
}
```

In questo modo vedremo – nella sezione **Debug** della finestra di gioco – il nome dell'oggetto inquadrato, SE questo è provvisto di un **Collider**; avviando il gioco in questo momento, ad esempio, io vedo inizialmente “**Empty**” perché nell'angolo a destra c'è una Empty (con questo nome) provvista di **Box Collider**; se guardo le pareti vedo ancora la scritta Empty, rimasta da prima, in quanto le pareti NON hanno un componente **Collider**, quindi non vengono “viste” dal **Raycast**; se, infine, inquadro il **Cube** sospeso a mezz'aria, a sinistra, vedo la scritta “**Cube**”, perché questo è il nome di tale *Game Object* – questo sì provvisto di **Collider**, come visibile nell'Inspector – così come lo troviamo nel **Project**.



Dal momento che possiamo capire qual è l'oggetto inquadrato, **possiamo verificare**, con degli **if**, **delle condizioni**... e qui potremmo davvero parlare in eterno dei possibili esempi ma mi limito a dirvi come fare per vedere se gli oggetti (telecamera e oggetto intercettato) si trovano ad una certa **distanza**, cosa che ci permette di verificarre una combinazione utile in vari casi; ad esempio, in un museo virtuale, possiamo far partire un commento audio, come visto nel capitolo su Audio Source, se stiamo guardando un oggetto e siamo “relativamente vicini”, oppure potremmo creare un radar o un sistema di puntamento missili in un gioco di guerra (“perdendo il contatto” col bersaglio se questo esce da un determinato raggio d'azione), e così via.

La **posizione dell'oggetto** che possiede lo script e **quella dell'oggetto inquadrato** sono, in effetti, dei **vettori 3D**, conservati in *transform.position* e Unity mette a disposizione una funzione nativa di Vector3, **Distance**, che ha la seguente firma:

```
Vector3.Distance(Vector3, Vector3) : float
```

per cui la funzione prende in input due parametri di tipo **Vector3** (nel nostro caso, i due *transform.position*, della telecamera e del *gameObject* colpito) e **restituisce un numero** con decimali (*float*) che indica la distanza tra i due punti.

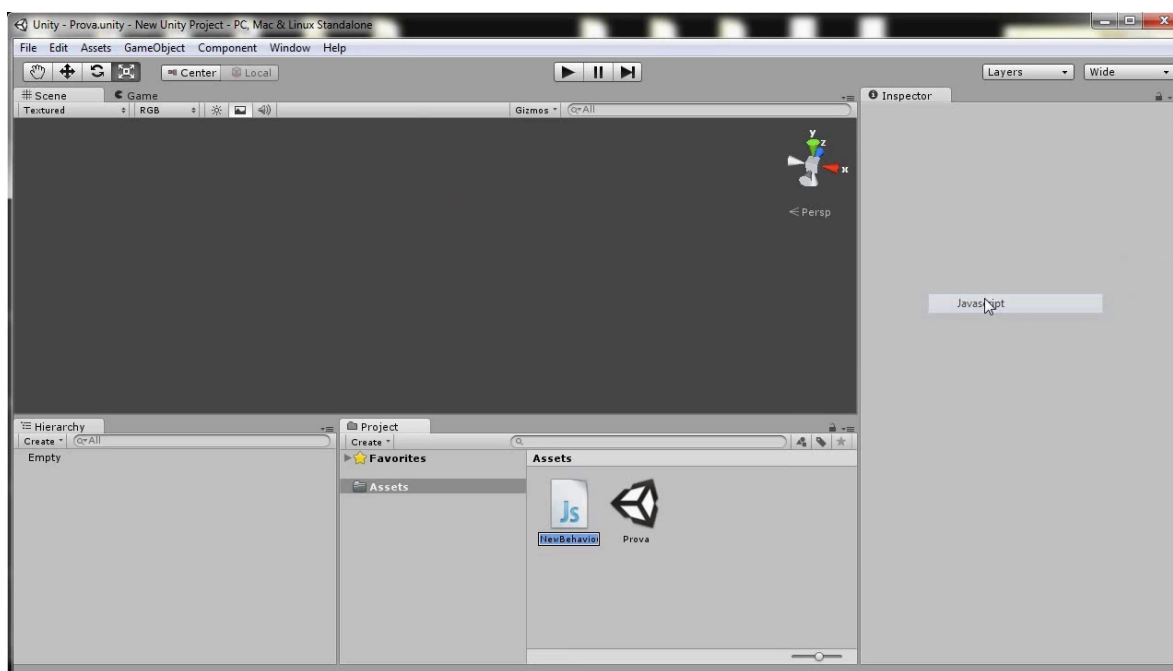
Questo è solo uno dei possibili esempi di ulteriori controlli (oltre al **name** o al **tag** del *Game Object* del *Collider*) che possono dar luogo a combinazioni interessanti, per applicazioni che spaziano dal tour virtuale a un gioco di guerra; ne abbiamo approfittato, più che altro, per parlare anche di **Vector3.Distance**.

Riprendendo il codice scritto precedentemente, abbiamo visto che l'elemento intercettato è depositato nella variabile **hit**, quindi dovremo fare *hit.collider.gameObject.transform.position* per ricavare il **Vector3** che indica la **posizione dell'oggetto colpito** e utilizzare questo valore come parametro di **Vector3.Distance** (per l'altro parametro sarà sufficiente scrivere *transform.position*, che si riferirà all'oggetto che possiede lo script, nel nostro caso la telecamera); questo valore andrà quindi usato in un **if** con un minore o un maggiore per determinare se l'oggetto si trova entro o oltre una certa distanza dalla telecamera.

Script JS: accedere ad un Array definito in un altro script

Questo breve **tutorial** sulla **programmazione JavaScript in Unity** nasce dalla richiesta di un utente del Forum *effe12.com* che mi ha chiesto **come poter accedere, da uno script, ad un array di interi** definito in un altro script. Il **tutorial** è stato realizzato con la versione **4.2 di Unity**.

Utilizziamo una scena completamente vuota ad eccezione di un oggetto – una **Empty** – che ci servirà per **inserire le istanze degli Script** nella scena ed eseguirli.



Creiamo quindi il **primo Script JavaScript**, chiamandolo ad esempio **“ScriptConArray”**, in modo da identificarlo chiaramente, quindi apriamo **MonoDevelop** per editarlo.

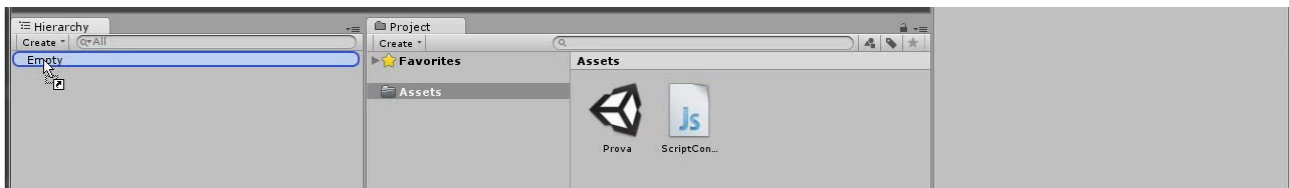
Nell'editor, scriviamo – in spazio globale, fuori da qualsiasi funzione – la seguente istruzione di **dichiarazione dell'array di interi**:

```
var ciao : int[];
```

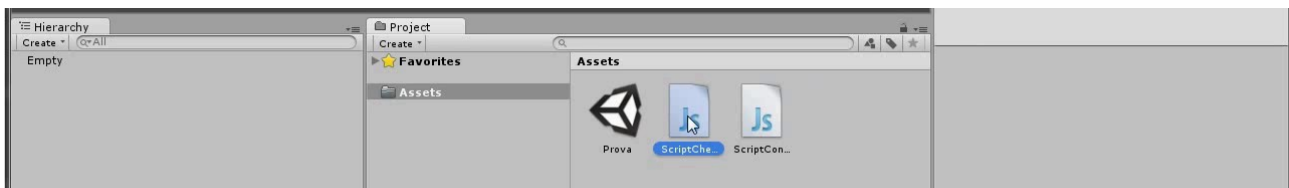

All'interno della funzione **Start**, che viene invocata all'avvio dello script (e, quindi, verrà invocata all'avvio di **Unity**, perché inseriremo questo script nella **Empty**, presente sin dall'avvio dell'applicazione), scriviamo l'inizializzazione dell'array, con:

```
ciao = [0,1,2,3,4,5];
```

Salviamo questo script, per rendere effettive le modifiche, e, se non l'abbiamo già fatto, **associamolo all'oggetto Empty** mediante click e trascinamento **da Assets a Hierarchy**.



A questo punto creiamo **un secondo script JavaScript**, chiamandolo ad esempio **“ScriptCheLegge”**, quindi apriamolo in **MonoDevelop** per editarlo.



In spazio globale, fuori da qualsiasi funzione, dichiariamo – senza inizializzarla – una variabile di tipo **ScriptConArray**, chiamandola ad esempio **“scriptDaLeggere”**, con:

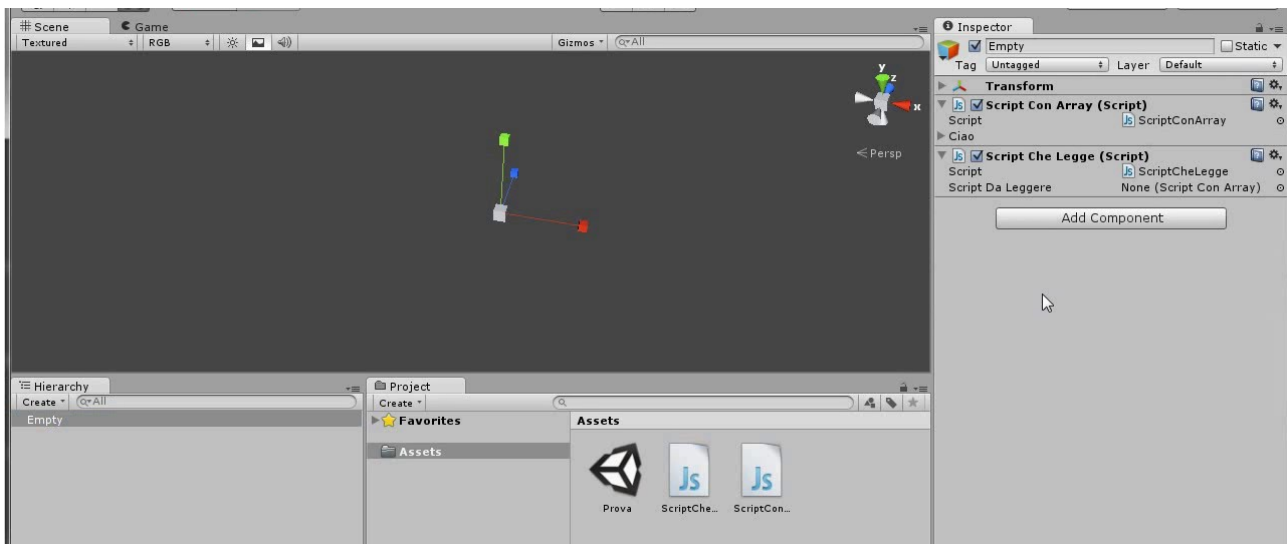
```
var scriptDaLeggere : ScriptConArray;
```

Attenzione, quindi: **“ScriptConArray”**, che è un Component, viene utilizzato qui come un tipo di dato a tutti gli effetti, al pari di un intero, di una stringa o di un qualsiasi oggetto. I suoi campi e metodi saranno quindi **campi e metodi accessibili come se si trattasse di una funzione di String** o della classe *Float*.

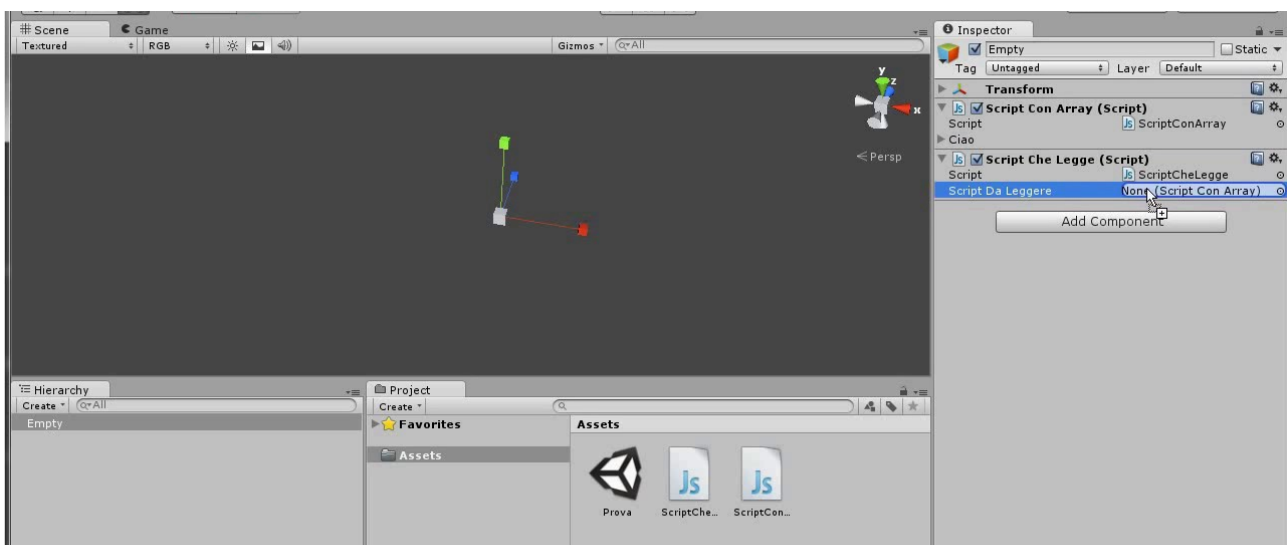
Salviamo quanto fatto finora, per applicare questa modifica, e torniamo un attimo all'editor di gioco di **Unity**.

Associamo anche questo script, mediante **drag and drop**, all'oggetto **Empty**, che adesso avrà sia *“ScriptConArray”* che *“ScriptCheLegge”*.

Selezioniamo l'oggetto **Empty**: nell'Inspector, notiamo che per *ScriptCheLegge* è presente un campo, chiamato *“scriptDaLeggere”*, di tipo *“ScriptConArray”*.



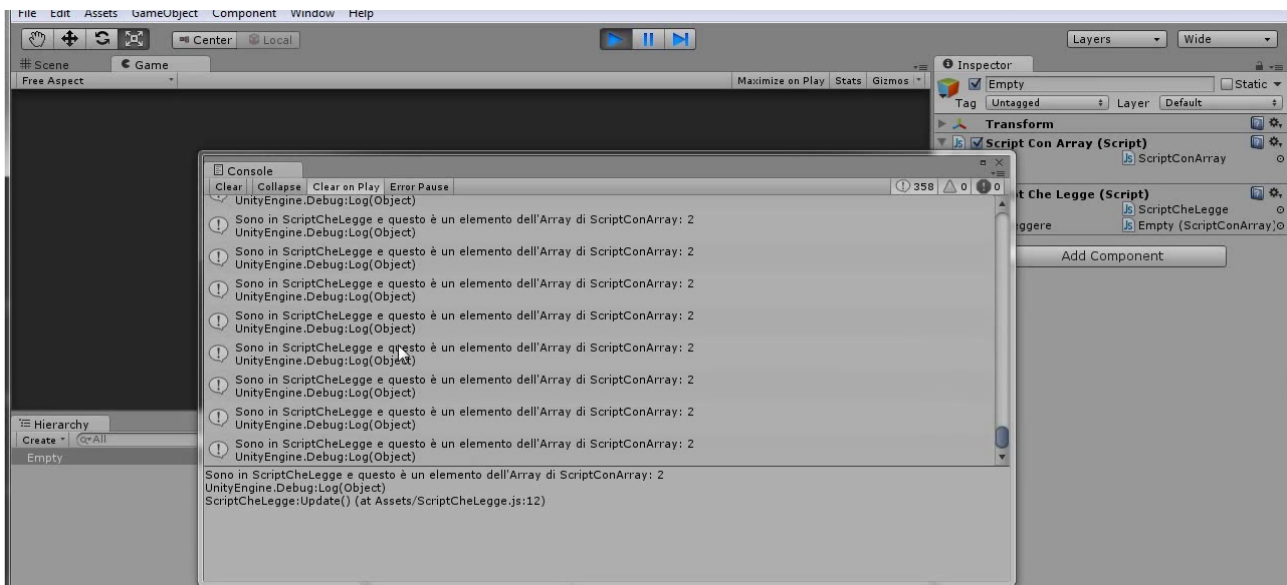
Facendo **click TRASCINANDO SENZA RILASCIARE** sul componente *“ScriptConArray”*, nell'Inspector della **Empty**, trasciniamo tale elemento nello slot *“scriptDaLeggere”* del componente *“ScriptCheLegge”*: abbiamo appena associato questo componente script alla variabile definita nell'altro script.



Torniamo quindi in **MonoDevelop** per continuare a modificare il secondo script, “**ScriptCheLegge**”, inserendo in particolare la seguente **istruzione nella funzione Update**:

```
Debug.Log("Sono in ScriptCheLegge e questo è un elemento dell'array di  
ScriptConArray: " + (scriptDaLeggere.ciao)[2]);
```

Salviamo, come sempre, per rendere effettive le modifiche, quindi **torniamo all'Editor principale di Unity** e premiamo su **Play**, osservando poi la scritta che verrà stampata **nella barra di stato**.



L'esempio mostrato è molto semplice ma ho preferito renderlo tale per mostrare “**in astratto**” come sia possibile utilizzare, come variabile di uno script, un altro componente (e, quindi, anche un altro script) definito per lo stesso oggetto, impostandolo tra l'altro via trascinamento nell'Inspector, per poi accedere ai suoi campi.

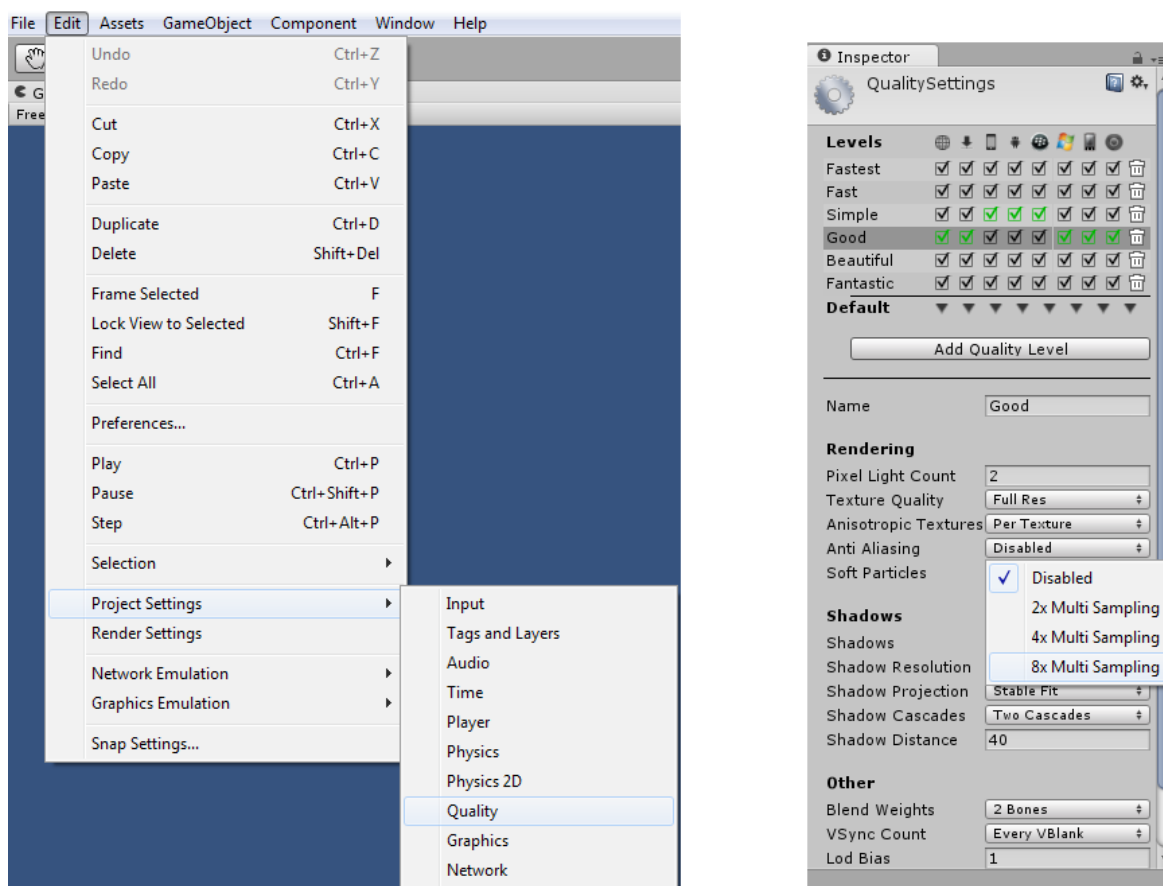
Faccio notare che **lo script da leggere può far parte anche di un altro oggetto di gioco**: in quel caso, lo script che legge dovrà avere **una variabile di tipo GameObject**, anziché **ScriptConArray**, e bisognerà **associare a tale variabile il GameObject mediante trascinamento nell'Inspector**; la lettura dell'array avverrà quindi con il nome del **Game Object**, “**punto**”, il **nome dello script**, “**punto**”, il **nome della variabile** (ossia accedendo nell'ordine a *Oggetto, Componente, Campo*).

* * *

Tips & Tricks: dove si trovano le impostazioni dell'Anti-Aliasing (AA)

Le **impostazioni di AntiAliasing (AA)**, ossia il filtro "antiscialettatura" del rendering della scena in **Unity 3D** si trovano, insieme a molti altri parametri di qualità della resa del progetto, nella scheda **"Quality"**, richiamabile dal menù **Edit - Project Settings - Quality**.

La scheda verrà visualizzata nell'**Inspector**, come visibile nell'immagine seguente.



Come si può vedere, **Unity** prevede delle preimpostazioni per la qualità della resa in fase di gioco: da **Fastest** (il più veloce, ma meno accurato, con effetti e funzionalità ridotte per ottenere una buona

velocità di esecuzione a scapito della qualità) a **Fantastic** (ottima qualità, che però può introdurre notevoli rallentamenti, soprattutto su dispositivi mobile).

La preimpostazione selezionata di default è **"Good"**, dove l'**AntiAliasing** è **disattivato**; per attivarlo, cliccare sul menù a scomparsa associato e **scegliere un'opzione di qualità**; intuitivamente, **8x darà risultati migliori**, ma può introdurre alcuni rallentamenti nell'esecuzione del progetto, soprattutto nelle scene con molti oggetti.

L'immagine seguente mostra **lo stesso oggetto** in modalità Play senza (a sinistra) e con (a destra) **AA con Sampling** (campionamento, proporzionale alla qualità di resa) a **8x** (esaminare, in particolare, i vari bordi, ad esempio quelli delle gambe del tavolo)



* * *

Unity + GIMP: creare delle Height Maps / Parallax Maps

In questo **tutorial** vedremo come preparare correttamente una **texture immagine di displacement** da utilizzare per i **materiali** con effetto **Parallax Bump in Unity 3D**. Si tratta di un tutorial di livello medio, che dà per scontata la conoscenza delle **Normal Maps** e del concetto di **Displacement Maps**; qui vedremo in particolare **come applicare una mappa Displacement ad uno Shader Unity**.

Questo effetto consente di dare **un'illusione di profondità** su una superficie composta da pochi poligoni (o comunque non molto dettagliata, a livello di geometria) mediante l'utilizzo di una **texture**, tipicamente in **bianco e nero**, che indica quali zone lasciare al livello base (parti in nero) e quali portare **in rilievo** (parti in bianco; ai grigi corrisponderanno quindi effetti intermedi).

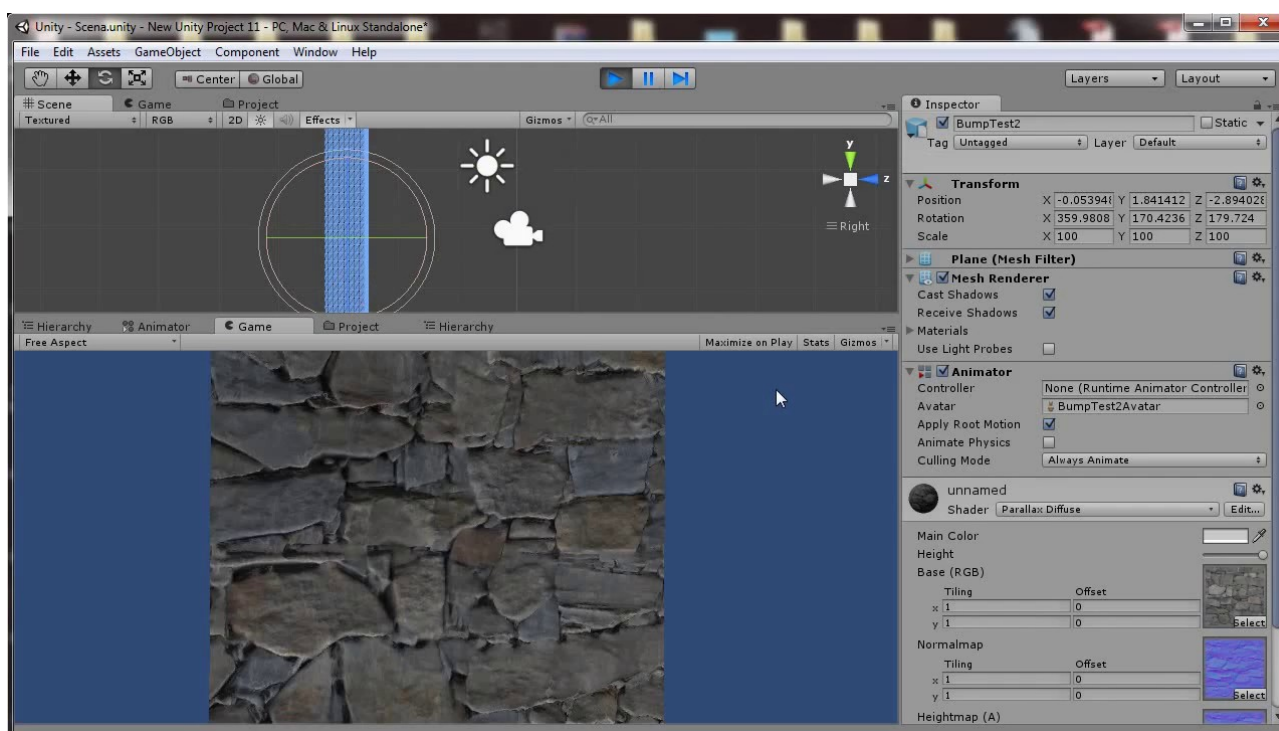
Si tratta di un effetto molto **simile** a quello ottenuto con le **Normal Maps**, ma leggermente migliore, che restituisce un **maggior senso della profondità**. La combinazione ideale consiste nell'utilizzare le mappe **Parallax per le deformazioni più grosse** (tenendo conto che si tratta comunque di un effetto “simulato”, non di una deformazione geometrica vera e propria!) e **le mappe Normal per i dettagli più minuti**.



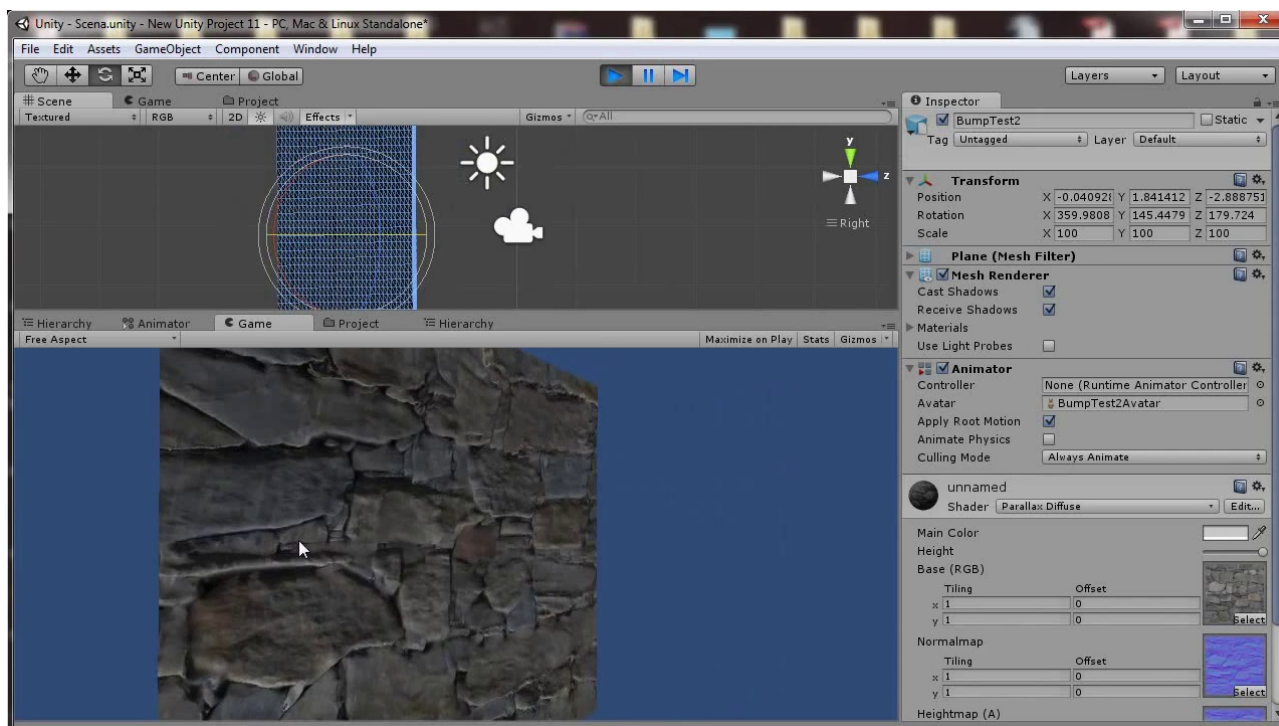
Attenzione, una precisazione importante: come detto all'inizio, in questo tutorial non vedremo come realizzare da zero una mappa di *displacement* per un oggetto, ma solo **come “impacchettarla”** correttamente in un file immagine, mediante **GIMP**, per utilizzarla come **Parallax Bump texture** in Unity.

Unity mette a disposizione diversi materiali (*shaders*) di base, oltre a consentire la creazione di **shaders personalizzati**; tra quelli di base, abbiamo ad esempio **Parallax Diffuse**, che consente di realizzare oggetti dotati di una texture per la colorazione diffusa (ovvero la colorazione di base vera e propria dell'oggetto), una **Normal Map** e una **Height Map**.

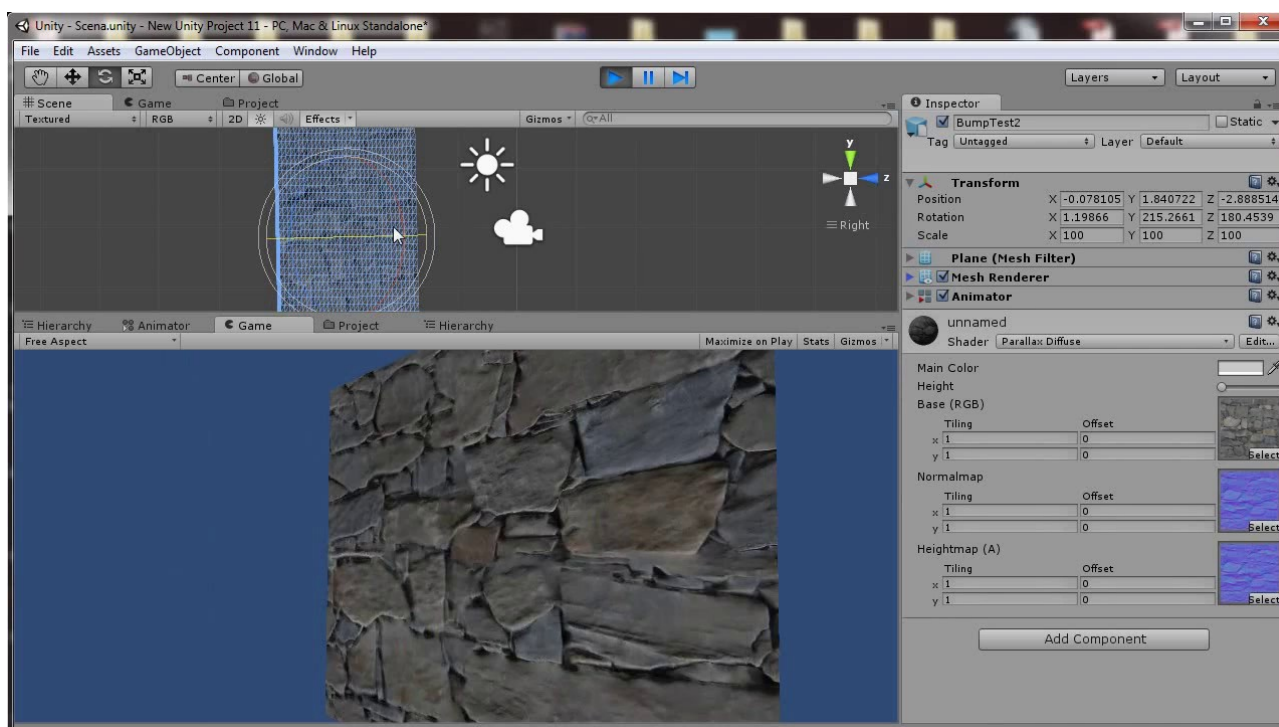
Osserviamo **la scena visibile** nell'immagine qui sotto: in questo caso abbiamo una telecamera, una fonte di luce direzionale (per illuminare la scena) e un semplice oggetto che, sostanzialmente, è un parallelepipedo molto schiacciato nel verso della profondità.



Come potete notare, **la mesh non è modellata**: è stata sì schiacciata in un verso, ottenendo una forma simile a un plane verticale, ma **le rocce che la compongono non sono state modellate** sulla geometria; eppure, guardando l'oggetto da varie angolazioni si ha l'impressione che qualche geometria ci sia.



Questa impressione è data dalle **mappe Normal e Displacement**; quest'ultima, in particolare, in **Unity** è detta **Heightmap** ed è implementata tramite gli **Shaders** del gruppo **Parallax**.



Lo shader che ho scelto è **Parallax Diffuse** e, come suggerisce il nome, consente di specificare il **colore di base (Diffuse Base RGB)**, l'effetto **Normal** e la **Heightmap**, che è la mappa vera e propria dell'effetto **Parallax**, o **Displacement** o **Bump** che dir si voglia.

Accanto alla scritta “**Heightmap**”, nella scheda del materiale dell'oggetto, troviamo **la lettera A** tra parentesi tonde. Questa **A** sta per **Alpha**, il canale di opacità e trasparenza presente in alcuni file immagine (ad esempio, è **presente nei PNG**, ma non nei file JPG o BMP) e indica che la texture da utilizzare per tale canale deve essere **memorizzata sotto forma di canale Alpha** di un'immagine.

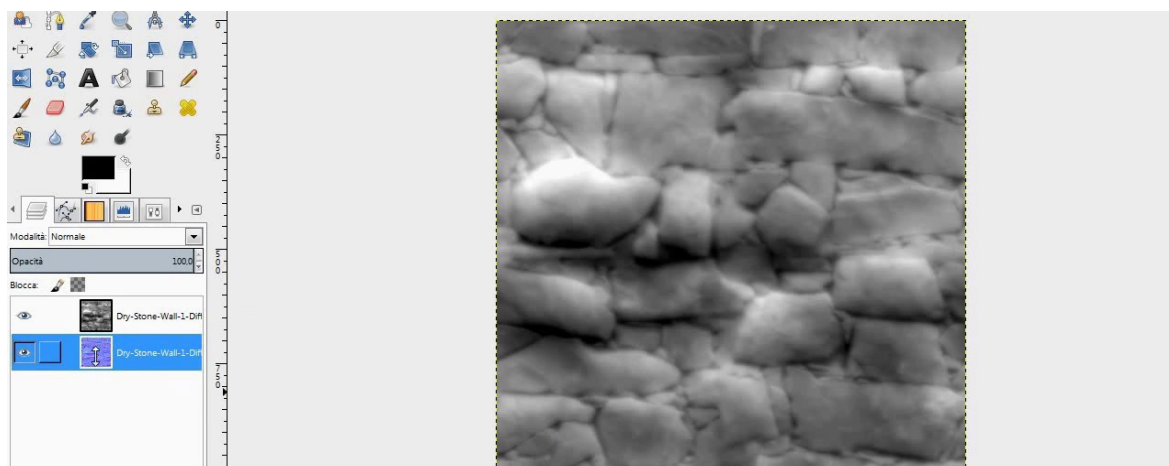
Generalmente, **si utilizza il canale Alpha della Normal Map**.

Riassumendo: gli effetti *Normal* e *Height*, negli *Shader Parallax*, vanno a braccetto; si utilizza una sola immagine per entrambi, ma per la **Normalmap** si utilizza il canale dei colori RGB, mentre per la **Heightmap** si utilizza il canale dell'opacità **Alpha**, per cui ci serve un file immagine con canale di trasparenza, ad esempio un file PNG.

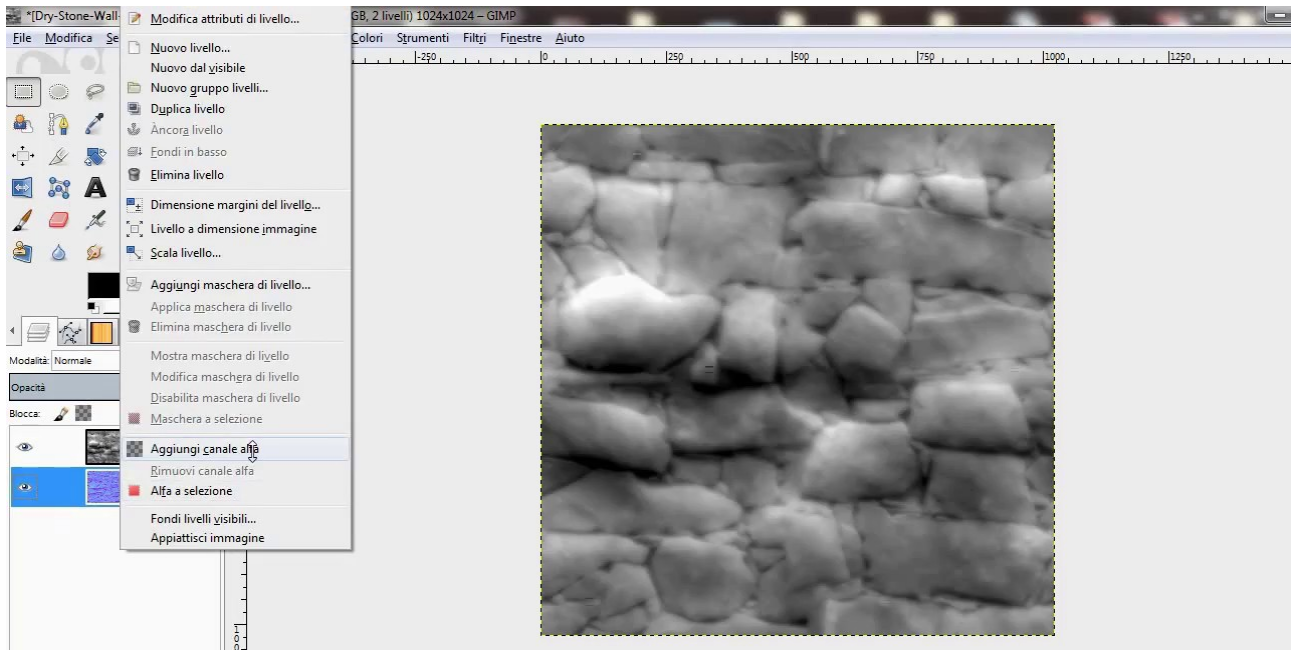
Il problema – e il vero argomento di questo tutorial – sta nel fatto che **spesso le Heightmap vengono generate o realizzate come immagini vere e proprie in scala di grigi**, non come canali **Alpha** di altre immagini.

Vediamo quindi come impacchettare una Normal e una Height in un unico file immagine, in particolare **un png con Alpha**.

Apriamo **GIMP** e carichiamo, come livelli separati, le due immagini di partenza: la **Normal Map** e la **Heightmap**; per far ciò, prima apriamo (con File – Apri) il file immagine della **Normal Map**, quindi scegliamo **File – Apri come livelli** e carichiamo, nello stesso progetto, la **Heightmap**, che verrà inserita come livello superiore nella finestra dei livelli.

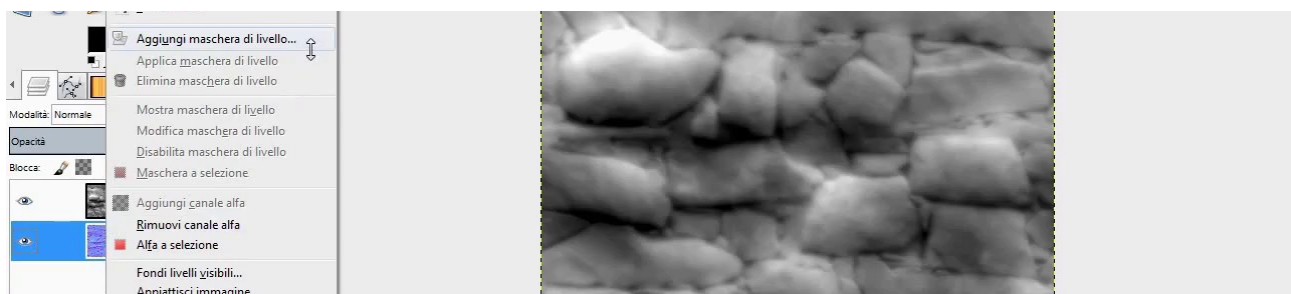


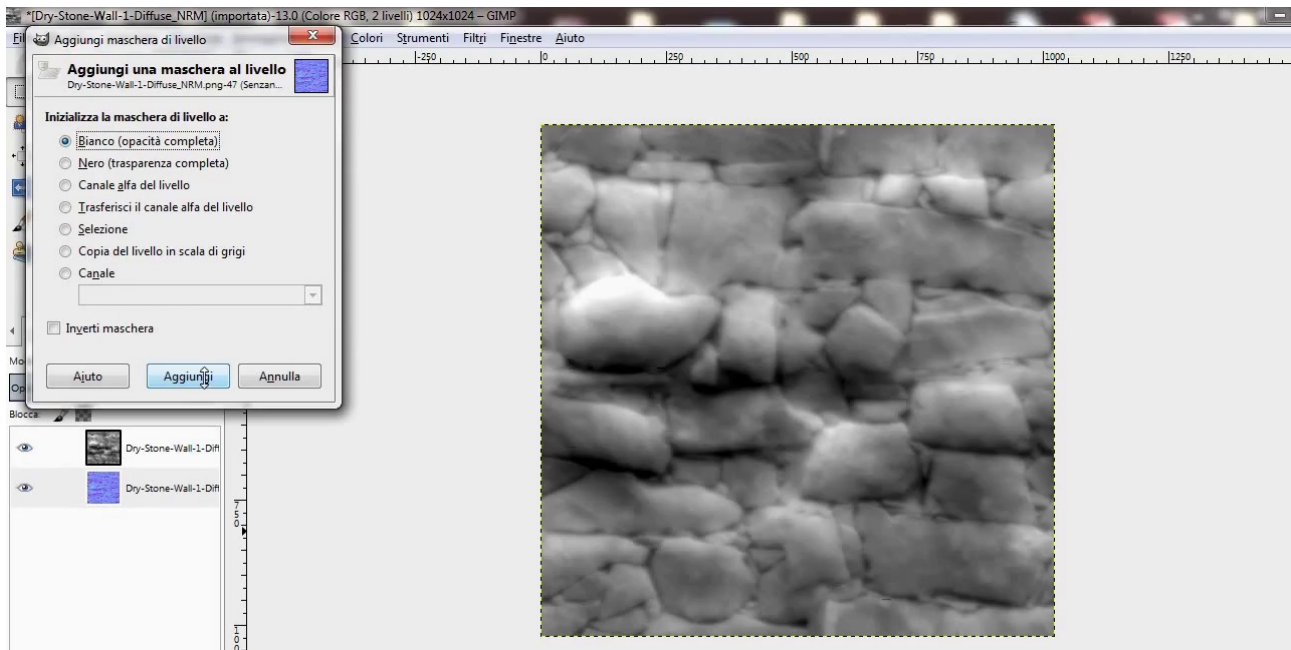
Vogliamo inserire tale immagine come **maschera Alpha del livello Normal**, quindi per prima cosa dobbiamo appunto dotare il livello **Normal** di un canale **Alpha**; per far ciò, selezioniamo tale livello (l'immagine, nella finestra dei livelli, dovrà avere un bordino bianco), quindi facciamo click destro e scegliamo **“Aggiungi canale Alfa”**.



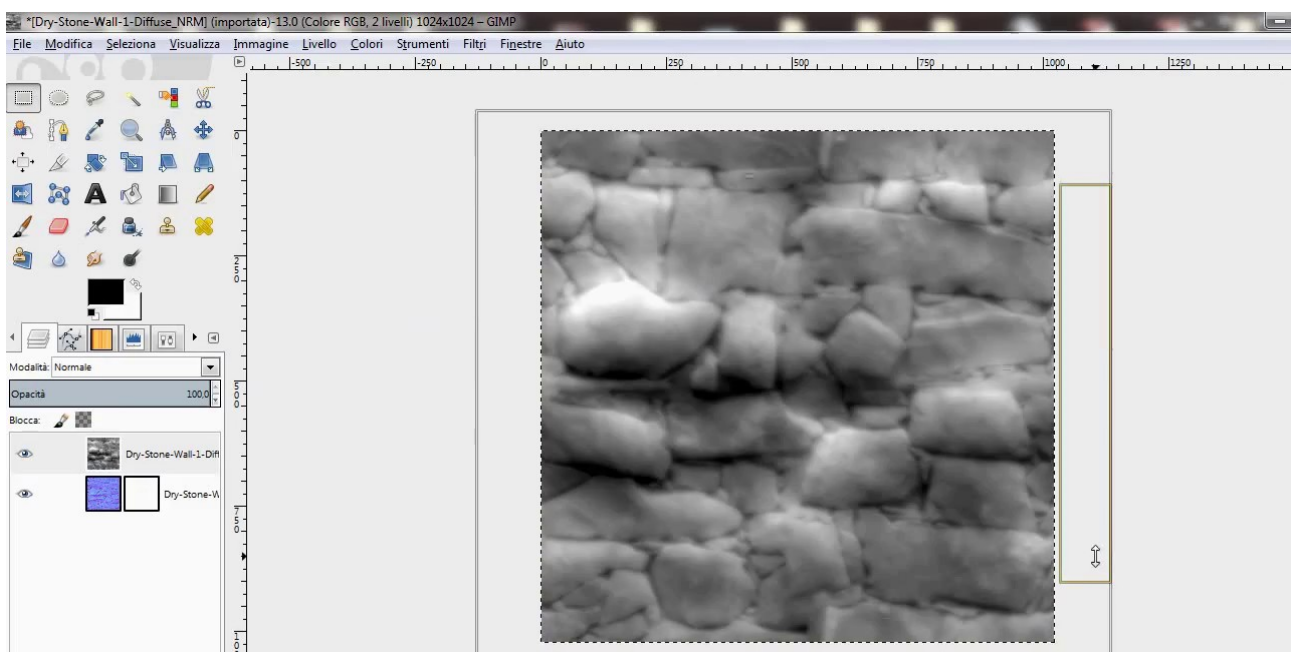
Il canale Alfa c'è, ma non la maschera di livello, la quale generalmente può essere utilizzata per realizzare vari effetti di miscelazione tra i livelli, ma che è, a tutti gli effetti, una maschera di trasparenza.

Facciamo quindi nuovamente click destro sul livello della **mappa Normal** e scegliamo, questa volta, **“Aggiungi maschera di livello”**. Nella finestra di dialogo che appare, scegliete una qualsiasi modalità, ad esempio la prima voce (**“Bianco – Opacità completa”**).



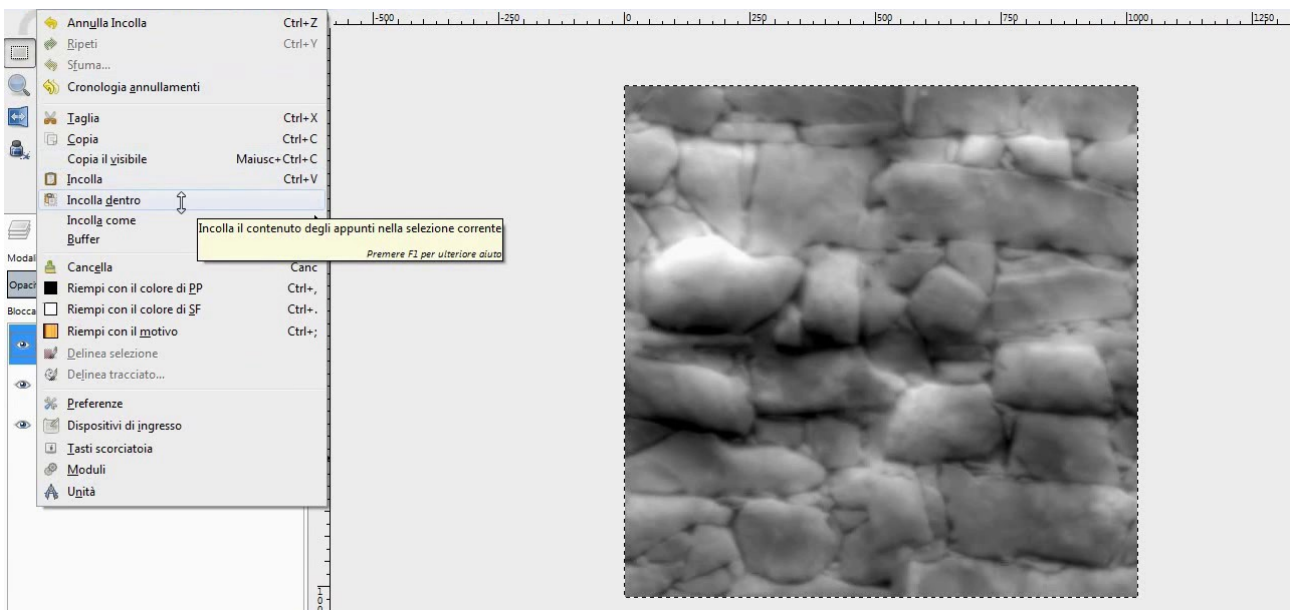
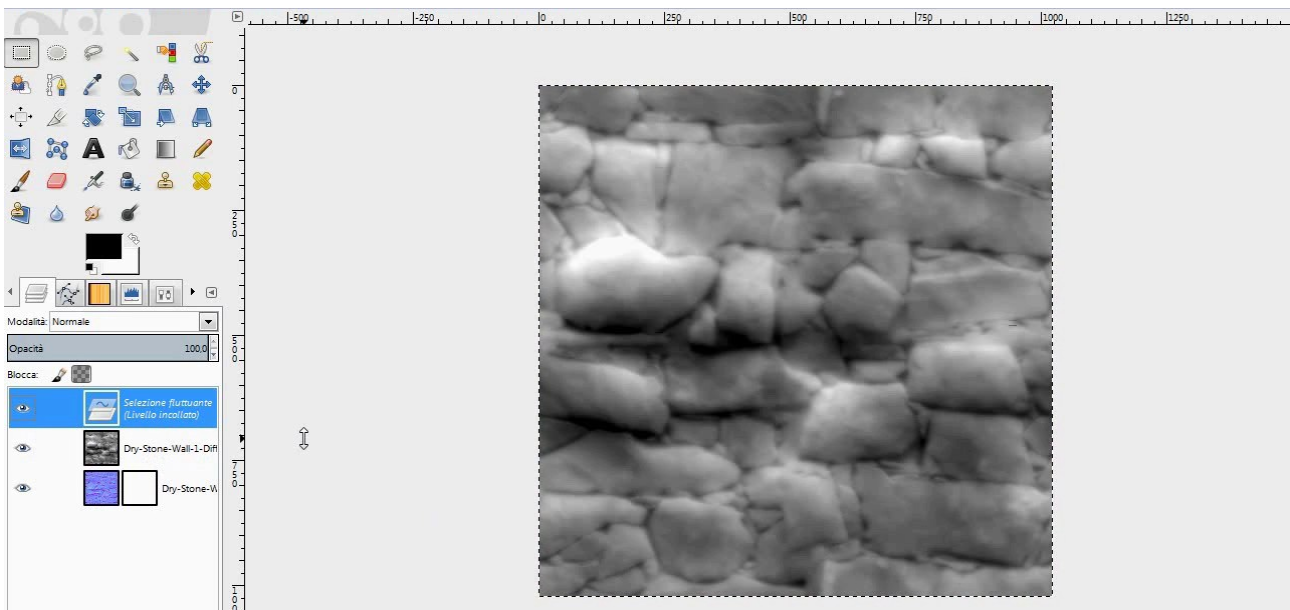


A questo punto, **selezioniamo il livello della mappa Heightmap** (anche qui, dovrà apparire un bordino bianco intorno all'anteprima, ad indicare che il livello è stato selezionato correttamente); selezioniamo lo strumento di selezione rettangolare e, con click sinistro e trascinamento, **definiamo un'area di selezione intorno all'immagine**. La selezione non deve essere accurata, potete andare “larghi”, oltre i bordi dell'immagine, come sto facendo a video.

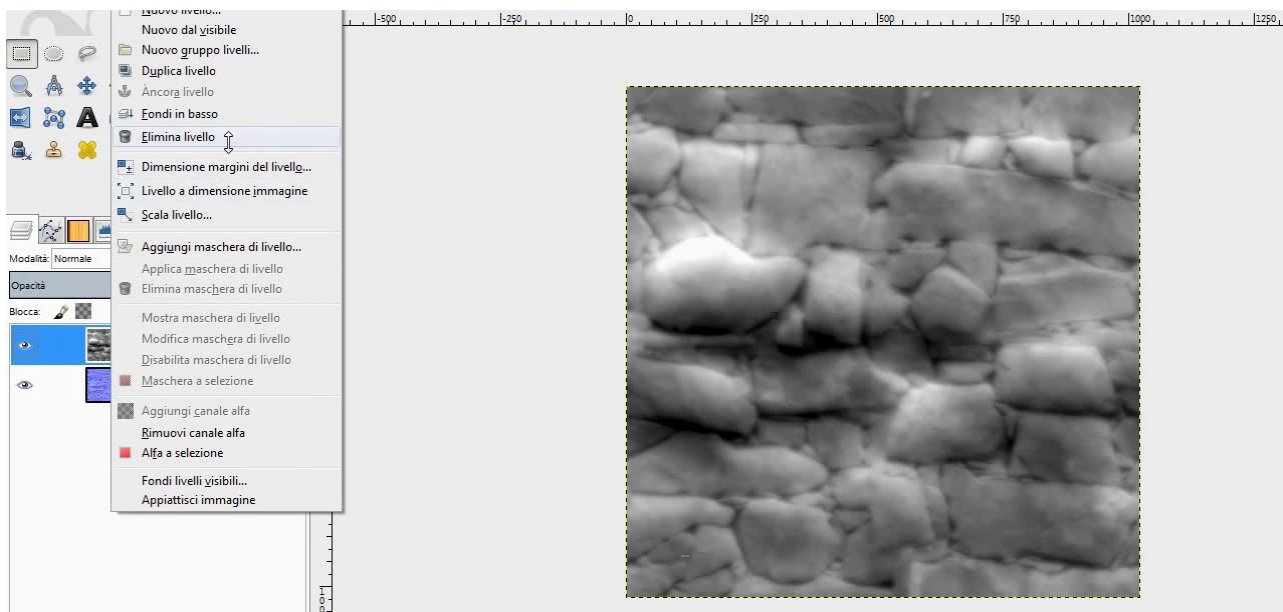


Premiamo **CTRL C** per mettere in memoria tale selezione (ovvero, l'immagine *Heightmap*) e facciamo nuovamente click sulla maschera di livello del livello **Normal**, selezionandola.

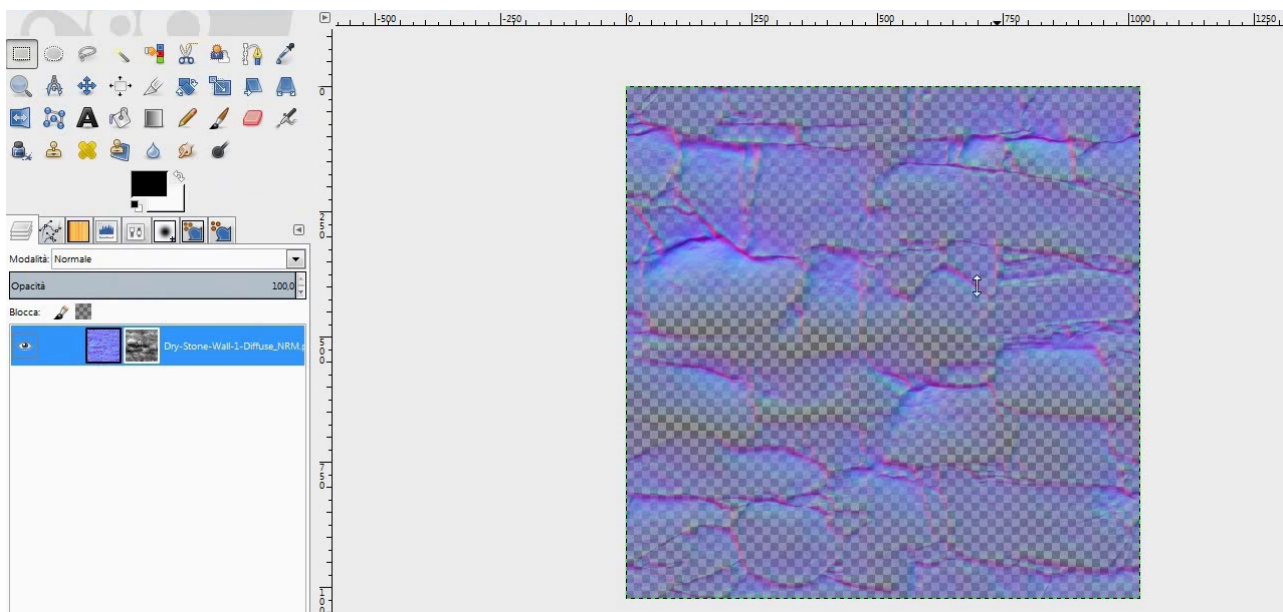
Premiamo **CTRL V** per incollare la selezione: **GIMP** la metterà in un “livello fluttuante”, come indicato nella finestra dei livelli; per applicare tale “livello fluttuante (incollato)” come maschera di livello, scegliamo **Modifica – Incolla dentro** (che, come dice il tooltip, “incolla il contenuto degli appunti nella selezione corrente”): fatto!

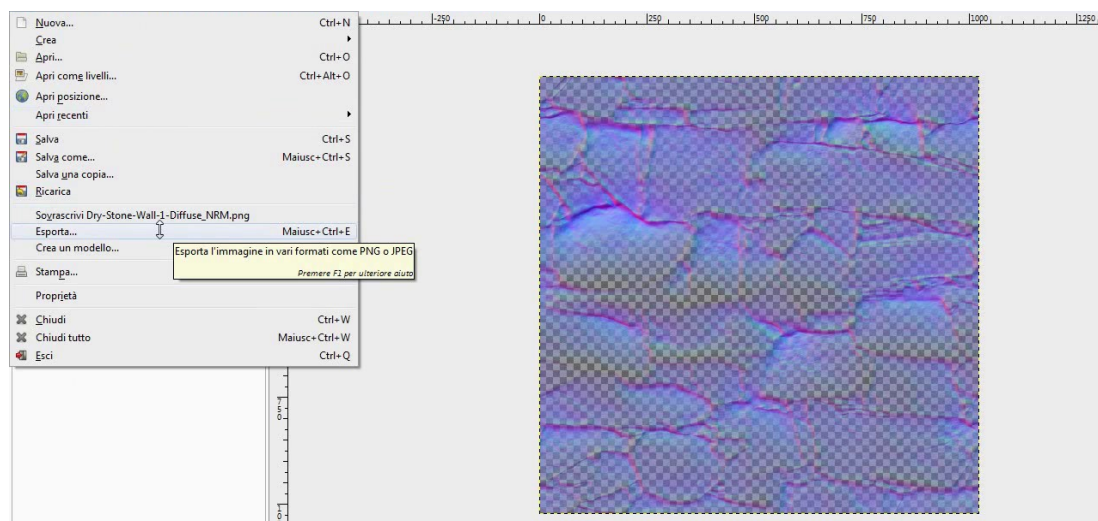


Dalla finestra dei livelli **eliminiamo** sia il livello fluttuante che il livello con l'immagine **Heightmap**, che ormai non ci serve più (in entrambi i casi, click destro sul livello e **“Elimina livello”**).

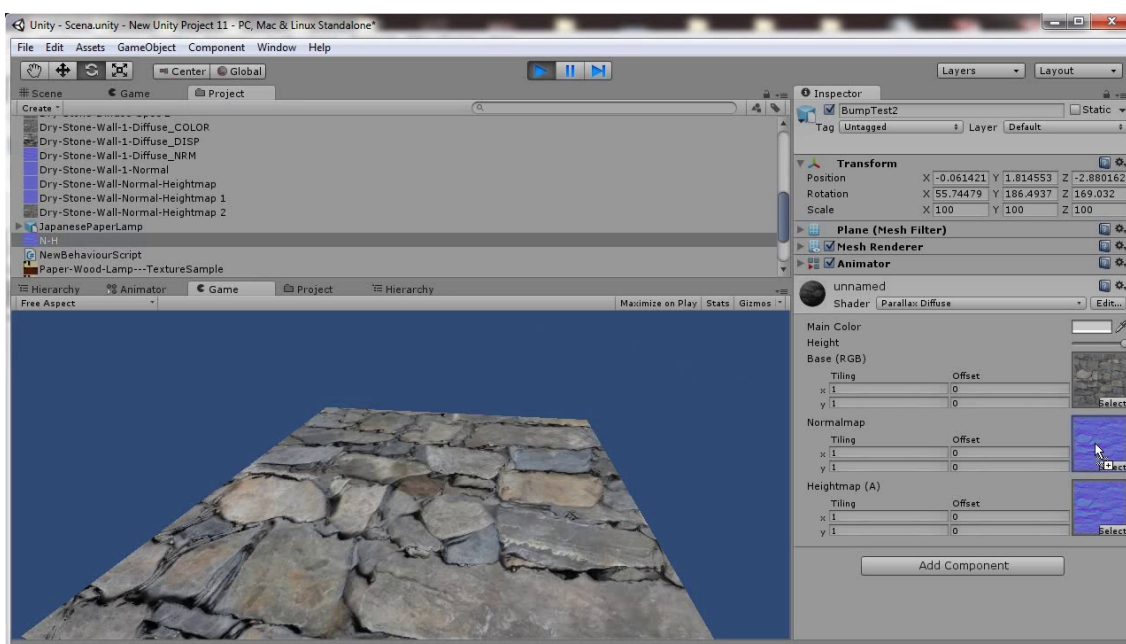


Salviamo quindi l'immagine, provvista di canale **Alpha**, come file **PNG**, mediante **File – Esporta**.





A questo punto, **una volta importata la nuova immagine** (una PNG che nel canale colori **RGB** è una **Normal Map**, mentre nel canale **Alpha** è una **Heightmap**) come **Asset in Unity**, applichiamo tale immagine **sia come texture Normalmap che come Heightmap** (una sola immagine, quindi, contenente entrambe le informazioni, su canali separati). Attenzione: cliccate su **“Fix Now”** sotto la thumbnail della **Normal Map**, nell'**Inspector**, per rendere l'immagine una **Normal Map corretta**.



Provate ora, in modalità **Play**, a variare sia l'**angolo di visualizzazione** che il valore dello **slider Height**, nell'Inspector; tale valore regola l'**ammontare dell'effetto Displacement** e vi consiglio di **non esagerare**, perché valori elevati potrebbero introdurre brutte distorsioni nel risultato finale.

* * *

* * *