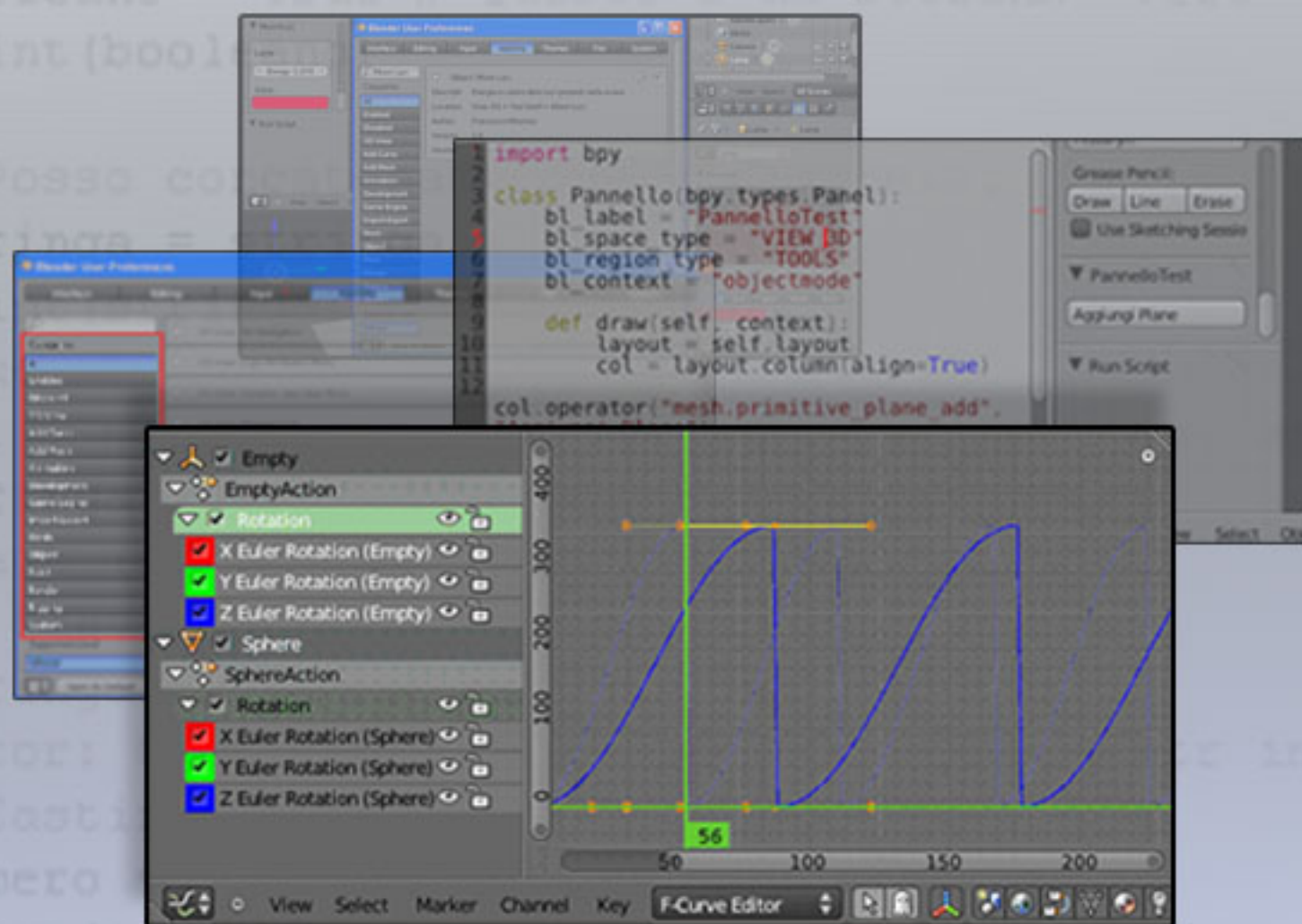


Programmare Script e Add-Ons per Blender 2.5

Volume I



Francesco "RedBaron85" Milanese
Blender Foundation Certified Trainer

Programmare script e Add-Ons per Blender 3D 2.5

Volume 1

SOMMARIO

Premessa	5
Le basi del linguaggio Python	6
Il linguaggio Python	6
Cosa serve per programmare in Python	7
Il prompt dell'IDLE	8
Importazione di moduli	9
Commenti	9
L'indentazione	10
Tipi di dati primitivi e definizione delle variabili	10
Lo slicing	12
Le liste	13
Gli insiemi	14
Le tuple	15
I dizionari	16
I comandi dir e type	17
Il blocco IF	18
Il ciclo FOR	19
Il ciclo WHILE e l'istruzione BREAK	20
Le funzioni	21
Classi ed ereditarietà	23
Gestione delle eccezioni	24
Altri argomenti	26
Informazioni e operazioni preliminari	27
La documentazione delle API Blender Python	27
Creazione degli script e degli Add-Ons: strumenti disponibili	29
Descrizione dei principali sottomoduli di bpy	34
Primi passi con le Blender Python API	36
Trasformazioni in Object Mode	44

Animazioni Object Mode: un case study	53
Creazione di un Add-On per Blender, informazioni preliminari	61
Operatori e modificatori in Object ed Edit Mode	66
ESEMPIO 1	70
ESEMPIO 2	71
Moduli di Input / Output su file: un Case Study	73
Case Study: il plugin “Add Point”	81
Case Study: il plugin “Mixer Luci”	90

Premessa

La possibilità di estendere le funzionalità di base di Blender 3D mediante la definizione di script e Add-Ons in linguaggio Python è uno dei punti di forza di questo meraviglioso programma.

Le API di programmazione e l'intero sistema di gestione dei plugin hanno subito notevoli cambiamenti rispetto alla versione 2.4 del programma, rendendo inutilizzabili gli script sviluppati per tali release ma più facile la scrittura dei plugin destinati alla versione 2.5.

In questo libro, realizzato facendo riferimento alla versione 2.59 stabile delle API Blender Python, dopo un capitolo introduttivo riguardante il linguaggio Python puro si procederà alla definizione di script sempre più articolati fino alla creazione di veri e propri plugin, detti Add-Ons, integrabili in maniera stabile nell'applicazione.

Il libro è rivolto a chi ha una buona conoscenza degli elementi principali di Blender 3D 2.5 e possibilmente un po' di familiarità, se non con il linguaggio Python puro, con i concetti fondamentali della programmazione orientata agli oggetti.

Francesco *RedBaron85* Milanese è un Blender Foundation Certified Trainer.

Ha realizzato e gestisce il sito web **RedBaron85.com** (www.redbaron85.com), dove pubblica periodicamente videotutorials, ebooks e guide sull'utilizzo di Blender 3D e altri programmi di CG.

Tiene inoltre corsi, seminari e consulenze, sia per enti pubblici che privati, su vari software e linguaggi di programmazione riguardanti la Computer Grafica 3D.

Per deselezionare tutti gli elementi presenti nella scena è sufficiente scrivere:

```
>>> for obj in bpy.data.objects:  
...     obj.select = False  
...  
>>> |
```

mentre per selezionarli tutti dovremo scrivere:

```
>>> for obj in bpy.data.objects:  
...     obj.select = True  
...  
>>> |
```

Con tutti gli elementi della scena selezionati, dopo l'ultima operazione, approfittiamone per mostrare **una brutta sorpresa**: digitando

```
>>> oggettiSelezionati  
[bpy.data.objects["Pavimento"], bpy.data.objects["Camera"]]  
>>> |
```

non otterremo, in output, la lista *aggiornata* degli elementi attualmente selezionati!

Questa considerazione vale per tutte le variabili create *a partire da elementi context*, in quanto il contesto cambia continuamente, e nelle variabili create *a partire da altre variabili definite in "stati precedenti"*, perché in questi casi non c'è un aggiornamento a catena; ad esempio, aggiungendo un cubo nella scena e digitando:

```
>>> for obj in elementiScena:  
...     print(obj.name)  
...  
Camera  
Cube  
Lamp  
Pavimento  
>>> |
```

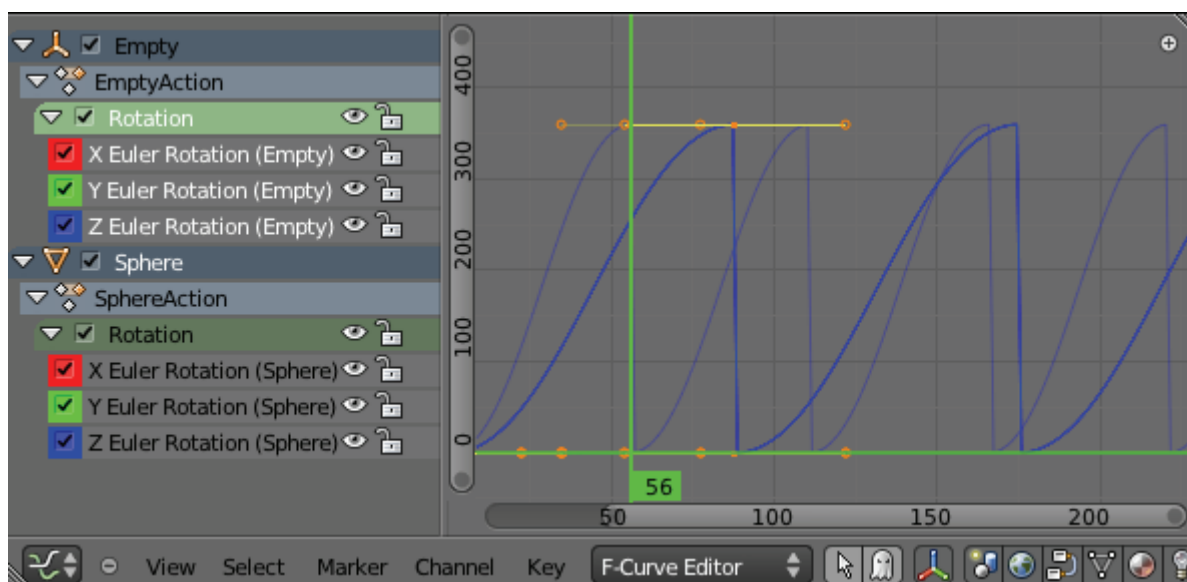
avremo tale oggetto nell'elenco restituito, ma non lo troveremo nella lista *"listaMesh"*, definita da un altro oggetto prima dell'inserimento del cubo:

```
>>> for elemento in listaMesh:  
...     print(elemento.name)  
...  
Pavimento  
>>> |
```

A questo punto si prelevano, dalle liste create, solo i primi elementi (in realtà, ciascuna lista contiene solo un elemento, ma per evitare di doverlo richiamare con `[0]` in seguito lo isoliamo ponendolo in una variabile temporanea) e si creano due oggetti *FModifier* di tipo “CYCLES”, ossia appunto due modificatori *CYCLES* per *F-Curves*, associandoli al volo alle due *curve IPO*.

```
>>>
>>> fcM = fcurvesMercurio[0]
>>> fcE = fcurvesEmpty[0]
>>>
>>> modM = fcM.modifiers.new(type="CYCLES")
>>> modE = fcE.modifiers.new(type="CYCLES")
>>>
```

Il risultato, oltre che nella **3D View** (avviando l’animazione) può essere verificato anche aprendo una finestra **Graph Editor** ed osservando le *curve IPO dei canali Z* della *Empty* e della sfera *Mercurio*: sono cicliche!



Osservando il **Graph Editor** ci rendiamo conto anche di un’altra cosa: **l’interpolazione** ai due keyframes è quella implementata di default da Blender, con le maniglie *Bezier*, ossia un’interpolazione “dolce”, mentre noi preferiremmo un’interpolazione lineare...

Dalle **API Blender Python** leggiamo che una *FCurve* mette a disposizione il campo *keyframes_points*, una collection di *FCurveKeyframePoints* ciascuno dei quali rappresenta uno dei *keyframes* della curva (i “punti” sulla curva nel **Graph Editor**); poniamo tali liste in due variabili d’appoggio, per comodità:

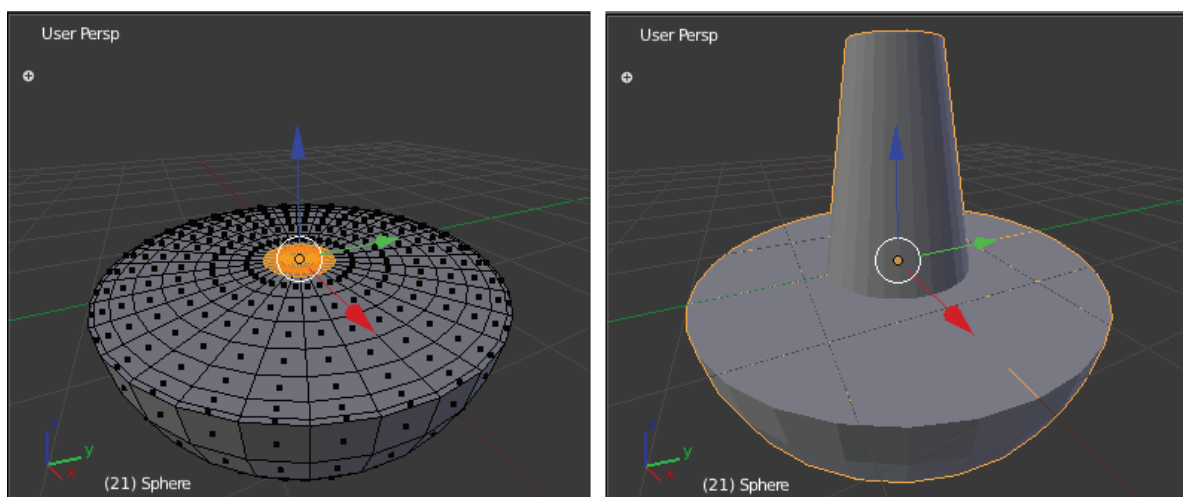
Quanto detto non implica l'impossibilità di applicare delle operazioni a vertici, spigoli o facce selezionate interattivamente in **Edit Mode** nella **3D View**; quello che bisogna fare, in questo caso, è:

- effettuare la selezione degli elementi da trasformare nella **3D View** in **Edit Mode**;
- passare in **Object Mode** (Blender memorizzerà comunque la selezione effettuata in Edit Mode);
- applicare lo script agli elementi (vertici, spigoli, facce, ...) con campo "*select*" a **True**.

Riprendendo la sfera editata con lo script visto precedentemente, selezioniamo qualche vertice in **Edit Mode** in una **3D View**, poi passiamo in **Object Mode** ed eseguiamo lo script definito qui di seguito nella finestra **Python Console**, osservando poi il risultato a video nella **3D View**.

```
>>>
>>> for v in verticiSfera:
...     if v.select == True:
...         v.co[2] = 2.0
...
>>> |
```

L'immagine seguente mostra la selezione effettuata in **Edit Mode** e il risultato dell'esecuzione dello script (eseguito *dopo* il passaggio in **Object Mode**).



Diverso è il caso delle **trasformazioni** da applicare ad una sottoparte di un oggetto; con **.co**, infatti, abbiamo modificato dei *valori* propri dell'elemento, ma a volte è necessario effettuare *operazioni* quali traslazioni, suddivisioni, beveling, ecc...

Queste *operazioni* sono definite in **bpy.ops**, ossia appunto le **operations** messe a disposizione da Blender (e che possiamo estendere, definendo *ops* personalizzate, come vedremo nel Case Study in **Appendice A**).

Quello che faremo sarà quindi:

1. creare una **lista di stringhe**, ciascuna delle quali conterrà i **dati** di un **frame**;
2. in ogni **stringa**, inserire i valori recuperati dalle **particelle** ai vari **frame**, formattandoli secondo le modalità espresse precedentemente;
3. scrivere questa **lista** su **file**, un elemento per riga.

Nel sistema particellare, **una singola particella** è identificata mediante **“particle”**; intuitivamente, l’oggetto **“particles”** di un **“particle_system”** è una **collection**, una collezione ordinata di oggetti **“particle”**.

Per recuperare le **informazioni di posizione di una particella** ad un dato frame, quindi, è sufficiente scrivere:

```
>>> SistemaParticellare.particles[0].location
Vector((-0.3589247763156891, 0.9041329026222229, 0.21718668937683105))
>>> |
```

e, per estensione, il **ciclo** di recupero delle informazioni su tutte le particelle per 100 frames e di scrittura di tali dati in una stringa (da inserire nella list “dati”) può essere definito come segue:

```
>>> import math # Necessaria per round(n,2), che arrotonda n a due cifre decimali
>>> dati = []
>>> for i in range(100):
...     frameCorrente = i+1;
...     riga = "(";
...     bpy.ops.anim.change_frame(frame=frameCorrente)
...     for p in range(10):
...         x = round(float(SistemaParticellare.particles[p].location.x), 2)
...         y = round(float(SistemaParticellare.particles[p].location.y), 2)
...         z = round(float(SistemaParticellare.particles[p].location.z), 2)
...         riga = riga + str(x) + "," + str(y) + "," + str(z) + ")#("
...     riga = riga[:-2] # Serve a togliere "#(" a fine riga
...     riga = riga + "\n"
...     dati.append(riga)
...
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}
```

Segue la fase di **scrittura dei dati su file**: è sufficiente aprire un **descrittore di file** (un oggetto, associato ad un **file**) e scrivere, mediante ciclo **for**, le stringhe contenute in **dati** (stringhe che terminano in **\n** per implementare il ritorno a capo automatico), per poi **chiudere il descrittore** del file con **“close()”**.

box (disposti, di default, in colonna, un elemento per riga) appena creato; per stampare il nome della **Lamp** in questione all’inizio di ogni **box**, utilizziamo la funzione “*label(text, icon)*”.

Attenzione: all’interno del blocco data vengono memorizzati anche gli elementi cancellati o non più utilizzati; per mostrare i comandi delle **Lamp** effettivamente presenti nella scena, all’interno del blocco *for* verrà posto un blocco *if* che controllerà che il numero di utilizzatori (*users*) del blocco dati in esame sia maggiore di 0; in caso contrario, non prenderà in considerazione quel blocco dati e non ne riporterà i parametri nel pannello.

Ecco quindi il corpo centrale dello script:

```
class MixerLuci(bpy.types.Panel):
    bl_label = "MixerLuci"
    bl_space_type = "VIEW_3D"
    bl_region_type = "TOOLS"
    bl_context = "objectmode"

    def draw(self, context):
        layout = self.layout
        col = layout.column(align=True)
        for l in bpy.data.lamps:
            if l.users > 0:
                box = layout.box()
                box.label(l.name)
                box.prop(l, "energy")
                box.prop(l, "color")
```

Completano la definizione dell’*Add-On* il dizionario *bl_info* e le funzioni per **registrare** e togliere classi e moduli dall’ambiente di **Blender**. Attivando lo script come **Add-On**, il suo pannello sarà sempre visibile nella **Tool Shelf** di ogni **3D View**.

