

ESEMPIO DI PROGRAMMAZIONE 3D CON PYTHON: LA TERRA INTORNO AL SOLE

Introduzione

Spesso guardando qualche simulazione tridimensionale o studiando la matematica ci si chiede se siamo in grado di riprodurre dei fenomeni al computer usando qualche linguaggio di programmazione apposito. Teoricamente non è una cosa semplice da fare in linguaggi come il C o usando librerie basate su progetti come OpenGL ma, fortunatamente, c'è qualcuno che ci ha semplificato la vita scrivendo delle classi in python che consentono di gestire la grafica tridimensionale: parlo del progetto Visual Python giunto ormai alla versione 5. Proviamo ad implementare un modellino in scala della Terra intorno al Sole.

Download e installazione

Farò riferimento al sistema operativo Gnu/Linux, per la precisione Ubuntu 10.10.

Possiamo installare python-psyco (che velocizza i nostri programmi) e le librerie necessarie per la grafica python-visual digitando da terminale root:

```
apt-get install python-psyco python-visual
```

fatto ciò non vi resterà che avviare il vostro editor preferito (consiglio NetBeans con qualche estensione per interpretare il linguaggio python).

La documentazione dettagliata di tutte le funzioni la troverete in `/usr/share/doc/python-visual/` oppure sul sito delle librerie.

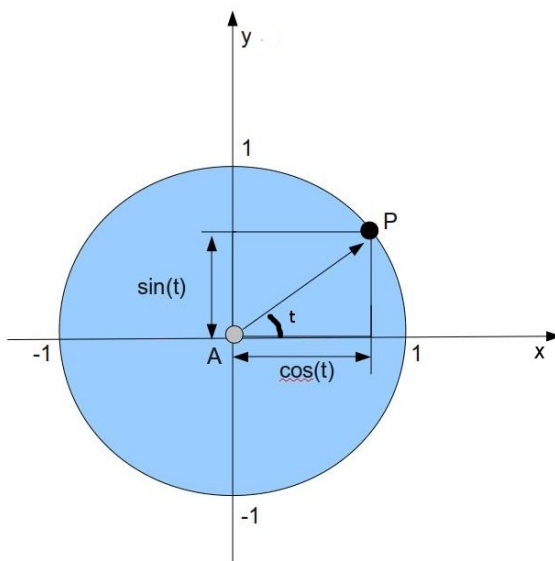
Analisi e Progettazione

Passiamo allo studio del problema e alla soluzione matematica. Proviamo a modellare l'orbita con una circonferenza goniometrica di raggio 1:

Un generico punto P che ruota intorno al centro della circonferenza (ovvero (0,0)) ha coordinate:

$$P=(x, y) \Leftrightarrow x=\cos(t), y=\sin(t), t \in \mathbb{R}$$

Ecco uno schema illustrativo:



Il vettore che congiunge il punto A con il punto B può essere visto come il raggio della circonferenza di lunghezza 1. il punto P ha coordinate $(\cos(t), \sin(t))$. All'aumentare di t , il punto P ruota in senso antiorario al punto A.

In questo modo possiamo vedere il punto P come il pianeta Terra e il punto A come il Sole ed il vettore descritto prima come la distanza del pianeta dal Sole.

Dalla trigonometria sappiamo che

$$\cos^2(t) + \sin^2(t) = 1 \quad (*)$$

il valore 1 rappresenta appunto la lunghezza del vettore che congiunge il punto A al punto P.

E se vogliamo la circonferenza più grande? Per esempio vogliamo che il vettore AP abbia una lunghezza r ? Consideriamo l'uguaglianza:

$$r(\cos^2(t) + \sin^2(t)) = r$$

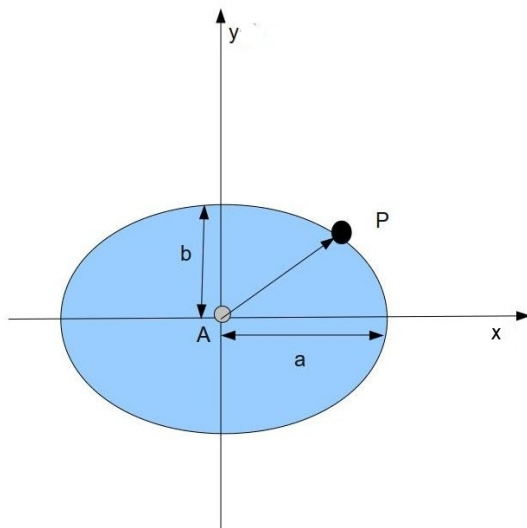
garantita dall'espressione (*). Abbiamo quindi che

$$r(\cos^2(t) + \sin^2(t)) = r(\cos^2(t)) + r(\sin^2(t)) = r$$

Da qui possiamo costruirci le coordinate:

$$(x(t), y(t)) = (r \cos(t), r \sin(t))$$

Abbiamo un piccolo problema: la prima legge di Keplero dice che le orbite dei pianeti non sono circolari ma ellittiche e il sole occupa uno dei fuochi. Abbiamo sbagliato tutto? No!! Possiamo vedere l'ellisse come una "circonferenza avente due raggi diversi": il semiasse maggiore ed il semiasse minore. Ecco un'interpretazione grafica:



Ad esempio possiamo considerare $b = a/2$, ovvero il semiasse minore b è la metà di quello maggiore. Quindi se ad esempio $a = r$ allora b varrà $r/2$. Quindi le coordinate diventano:

$$(x(t), y(t)) = (a \cos(t), b \sin(t))$$

Passiamo all'implementazione in python.

Programmazione

Se siamo abituati a programmare questa è la parte più banale: basta creare la variabile t e farla incrementare dentro un ciclo `while(true)`. Ad ogni iterazione si calcolano le coordinate x e y e poi mediante le istruzioni di disegno rappresentiamo la sfera nella posizione (x,y) .

Aperto l'editor python importiamo le librerie che ci servono:

```
import visual.materials
from visual import *
from visual.materials import *
```

A cosa ci servono? La prima libreria contiene tutti i materiali e le texture da assegnare ai solidi generati con le visual; la seconda è il 'core' delle visual e la terza è come la prima.

Caricato tutto quello che ci serve possiamo passare alla creazione della finestra:

```
scene=display(title="Earth around Sol", width=800, height=600, center=(0,0,0),  
background=(0,0,0))
```

tale finestra ha lo sfondo nero, una risoluzione di 800x600 e l'origine centrata in (0,0).
Di default quando un oggetto che si muove giunge ai bordi della finestra, la risoluzione cambia, ed è una cosa che a noi crea solo fastidio.. Per impedire ciò disabilitiamo tale proprietà:

```
scene.autoscale=false
```

Possiamo ora dichiarare le variabili che ci servono:

```
t=0 ← Variabile indipendente (Angolo compreso tra il raggio e l'asse delle ascisse)
```

```
y=0 ← Variabile dipendente per l'ordinata  $y=\sin(t)$ 
```

```
x=0 ← variabile dipendente per l'ascissa  $x=\cos(t)$ 
```

```
value=0.005 ← incremento della variabile t ad ogni iterazione
```

```
biggest_radius=8 ← semiasse maggiore [nel disegno si chiamava a]
```

```
smallest_radius=5 ← semiasse minore [nel disegno si chiamava b]
```

Ora passiamo alla creazione del pianeta:

```
pianeta=sphere(radius=0.5, material=visual.materials.earth)
```

dove:

radius=raggio del pianeta

material=la texture della Sfera, caricata dalle librerie visual.

Sappiamo la il nostro pianeta è leggermente inclinato. Implementiamo tale caratteristica:

```
pianeta.rotate(angle=-pi/7., axis=(0,0,1), origin=(0,0,0))
```

Passiamo al ciclo while che gestisce l'animazione.

while true:

```
rate(30)
```

```
x=(cos(t))*biggest_radius
```

```
y=(sin(t))*smallest_radius
```

```
t=t+value
```

```
pianeta.pos=(x,y,0)
```

dove:

rate(val): imposta la velocità del ciclo in val, utile nel caso in cui il ciclo gestisce spostamenti di un oggetto: in tal caso val determina la sua velocità.

A questo punto non ci resta che fare solo delle piccole modifiche:

- 1) Disegniamo il sole in uno dei due fuochi, prima del ciclo while scriviamo:
`Sol=sphere(radius=0.5, pos=((biggest_radius)/4,0,0), color=visual.color.yellow, material=visual.materials.emissive)`
- 2) Il movimento deve essere orizzontale (lungo l'asse z) non verticale. Quindi sostituiamo `pianeta.pos=(x,y,0)` con `pianeta.pos=(x,0,y)`
- 3) La terra ruota su stessa:
`pianeta.rotate(angle=pi/200., axis=(0,1,0), origin=(0,0,0))`

Ottimizzazione

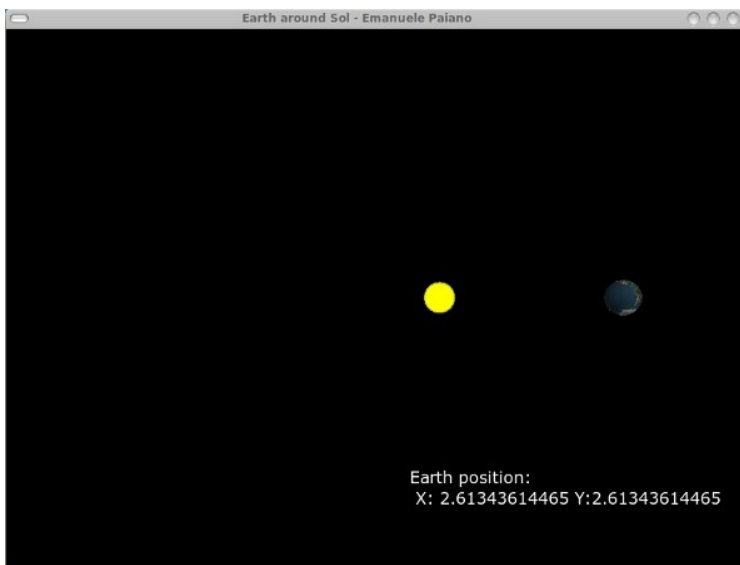
Spesso quando i nostri progetti diventano complessi e richiedono una quantità maggiore di calcolo, un linguaggio interpretato come Python inizia ad essere un vero e proprio macigno per il nostro processore al contrario di quanto non avviene con linguaggi compilati come il C. La soluzione? Importiamo l'acceleratore psyco nei nostri progetti gestito mediante un'eccezione per far sì che anche chi non dispone di tale componente lanci comunque il nostro programmino:

```
try:
    import psyco
    psyco.full()
except ImportError:
    print 'Psyco accelerator not found. This game can be run low.'
```

ed ecco che i nostri problemi sono risolti.

Testing

Possiamo testare il tutto lanciando lo script con il comando `'python nomescrpt.py'` dove `nomescrpt` è l'identificativo del file con cui abbiamo salvato il lavoro, oppure per chi usa NetBeans può premere Shift+F6 per far partire il programma corrente. Se tutto è andato bene dovrebbe apparirvi una finestra simile a questa:



Un segreto nascosto

Facciamo uscire un coniglio dal cappello: ricordate gli occhietti di cartone con una lente rossa e una verde che facevano vedere alcune immagini in un (falso) 3D? Tali librerie supportano la visualizzazione con questi occhiali in modalità stereoscopica. Dopo che avete inizializzato la finestra (variabile `scene`) aggiungete questa riga:

```
scene.stereo='redcyan'
```

indossate gli occhiali e vi ritroverete un 3D (un po' sfasato).. Ci sono altre opzioni per le immagini stereoscopiche ma dovete consultare la documentazione per queste.

Per venirvi incontro ho pubblicato il mio lavoro. Diamo uno sguardo al codice completo:

```
import visual.materials
from visual import *
from visual.text import *
from visual.materials import *

try:
    import psyco
    psyco.full()
except ImportError:
    print 'Psyco accelerator not found. This game can be run low.'

scene=display(title="Earth around Sol - Emanuele Paiano", width=800, height=600,
center=(0,0,0), background=(0,0,0))
#scene.stereo='redcyan'
#scene.fullscreen=true
scene.autoscale=false
t=0
value=0.005

#Sol distance
biggest_radius=8
smallest_radius=5

pianeta=sphere(radius=0.5, material=visual.materials.earth)

Sol=sphere(radius=0.5, pos=((biggest_radius)/4,0,0), color=visual.color.yellow,
material=visual.materials.emissive)

pianeta.rotate(angle=-pi/7., axis=(0,0,1), origin=(0,0,0))

text=label(text="X: Y:", pos=(6,-6,0), border=0, box=0)

while true:
    rate(30)
    x=(cos(t))*biggest_radius
    y=(sin(t))*smallest_radius
    t=t+value
    pianeta.pos=(x,0,y)
```

```
pianeta.rotate(angle=pi/200., axis=(0,1,0), origin=(0,0,0))
text.text=("Earth position: \n X: "+str(y)+" Y: "+str(y))
```

CONCLUSIONI

Non c'è molto da dire se non che le python-visual forniscono delle classi e funzioni molto utili per la creazione semplificata di oggetti 3D. Consultando la documentazione allegata e dando un'occhiata ai vari esempi potete notare la semplicità d'uso.

Con tali librerie è possibile anche scrivere qualche videogioco (infatti un c'è mio progetto non ancora pubblicato) in maniera meno masochista di come si può pensare... Basta rispolverare un pò di Matematica e qualche legge della Fisica: il trucco sta nel considerare lo schermo come un piano cartesiano e vedere le il grafico delle funzioni come traiettorie che l'oggetto in movimento dovrà percorrere. In genere per far muovere un punto basta incrementare la variabile (indipendente) x e porre $y=f(x)$, con f da noi scelta: ad ogni iterazione del ciclo while aggiorniamo la posizione dell'oggetto in $(x, f(x))$. Spero di essere riuscito a stimolare la vostra fantasia.



Emanuele Paiano