Un FPS in Blender

Introduzione

Stavolta non ci sono santi: se va blender allora potete pure fare il gioco.

Requisiti:

Blender 2.61

Fine: c'è già il python preinstallato.

Pronti via.

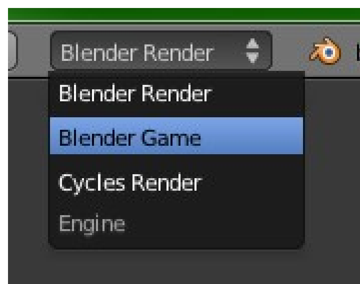
Lanciate blender da linea di comando. La linea di comando vi serve per catturare l'output dell'interprete python. Ricordo che in un tempo che fu all'avvio di blender era eseguita e lasciata aperta anche una finestrella del prompt del dos, ora non più – almeno sul mio PC.

Create una cartella vuota dove volete, da adesso la chiameremo “cartella”, poi salvate il progetto blender che avete aperto in quella cartella, con un nome qualsiasi.

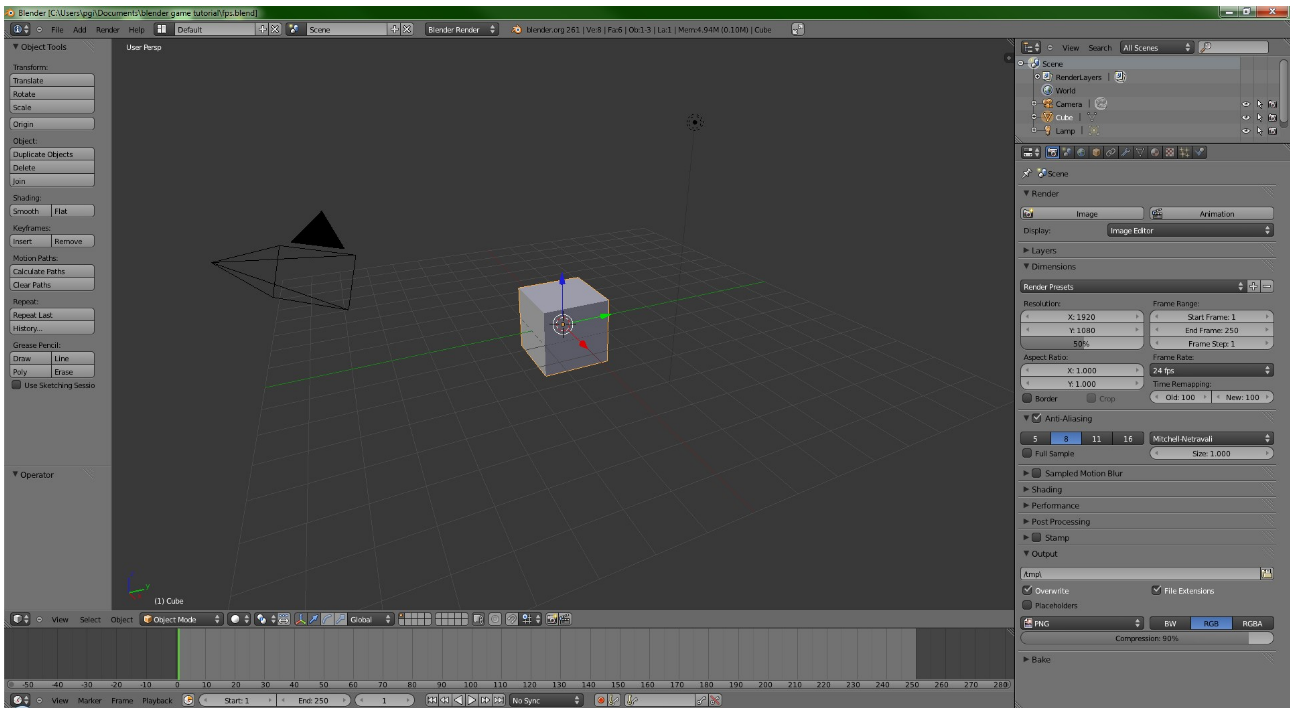
Nella stessa cartella, create un file di testo “gioco.py”: questo conterrà il sorgente python del modulo principale del nostro gioco.

Giochiamo con questa unica cartella poiché, per impostazione predefinita, blender piglia gli script personalizzati che stanno nella stessa cartella del file su cui sta lavorando. Si può fare anche altrimenti ma non è che stiamo qua a scriverci su un romanzo.

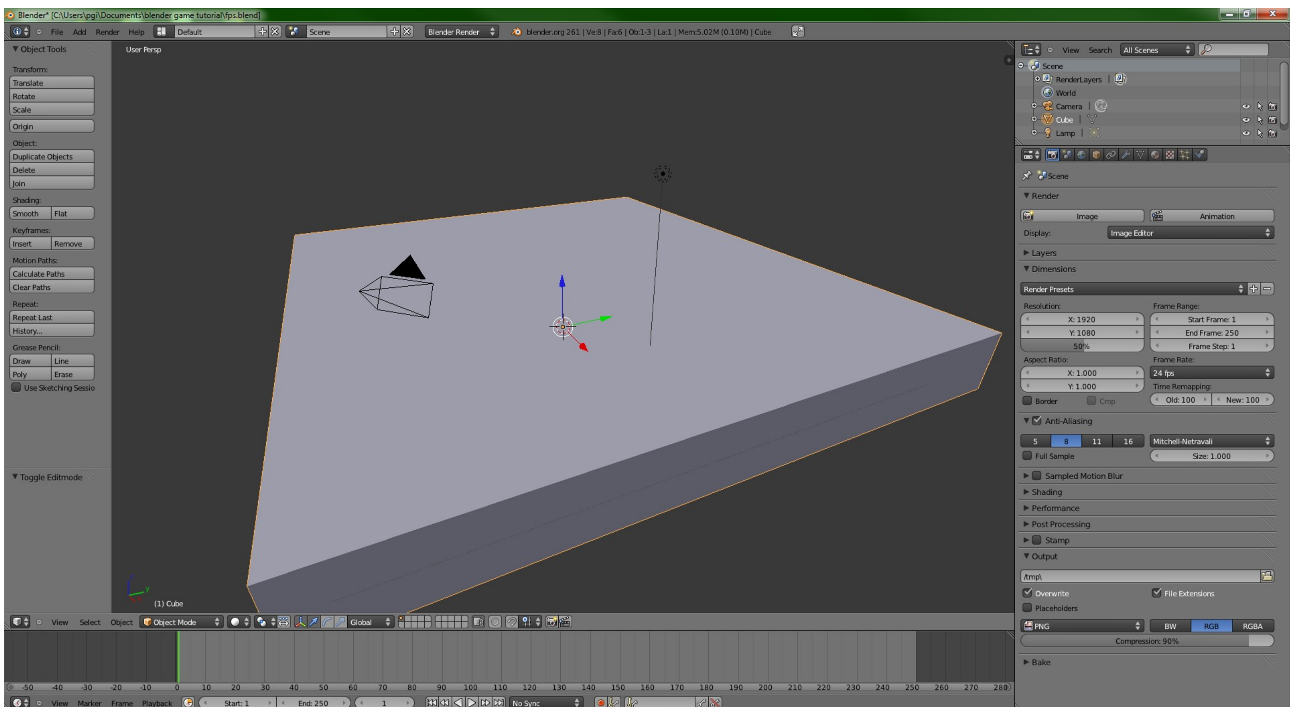
Impostate il renderer da Blender Renderer a Blender Game. Il pulsante per farlo è in alto:



Blender ha una valanga di funzioni e triccheballacche ma non ha un controllo in stile FPS per muoversi in giro. Non c'è problema, sono due righe. La prima cosa che serve è tuttavia una scena in cui muoversi. Ne facciamo una di grande valore artistico: stirate il cubo. Da così:

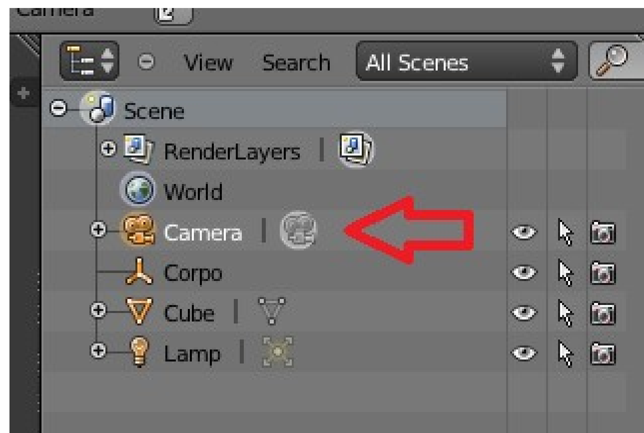


a così:

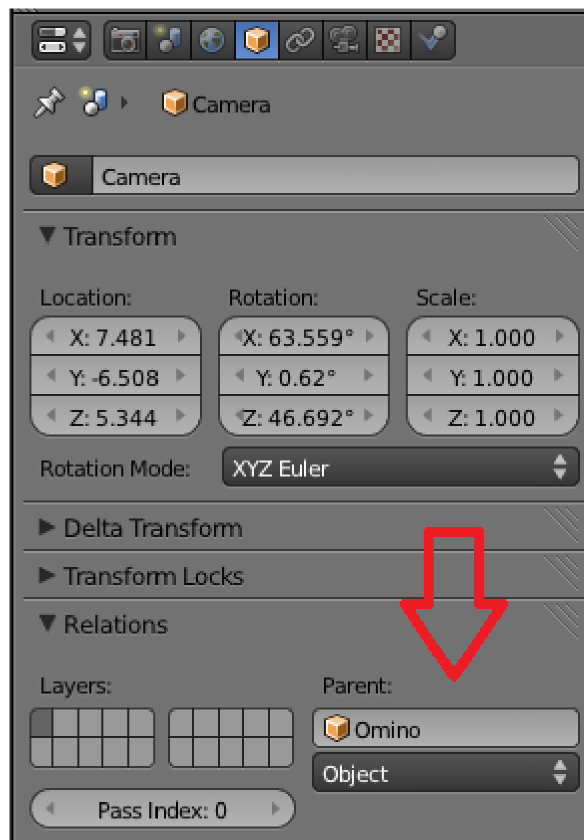


Adesso ci serve l'omino. Aggiungete alla scena un nodo vuoto (Add → Empty) e chiamatelo “Omino”. Potete anche usare un cubo o un cilindro o una palla, non è importante. Posizionate “Omino” un po' sopra il pavimento, ad esempio in (0,0,3).

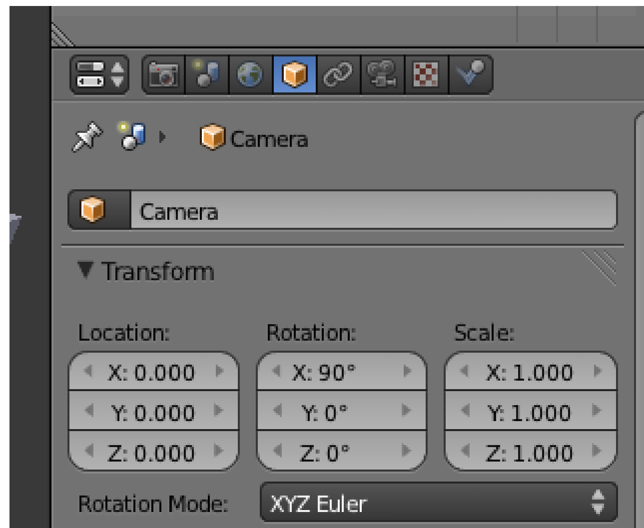
Siccome siamo in un FPS, facciamo in modo che la telecamera stia davanti alla faccia del nostro Omino. Lo facciamo dicento alla telecamera di usare Omino come suo genitore. Selezionate la Camera:



e impostate come Parent di Camera il nostro Omino:

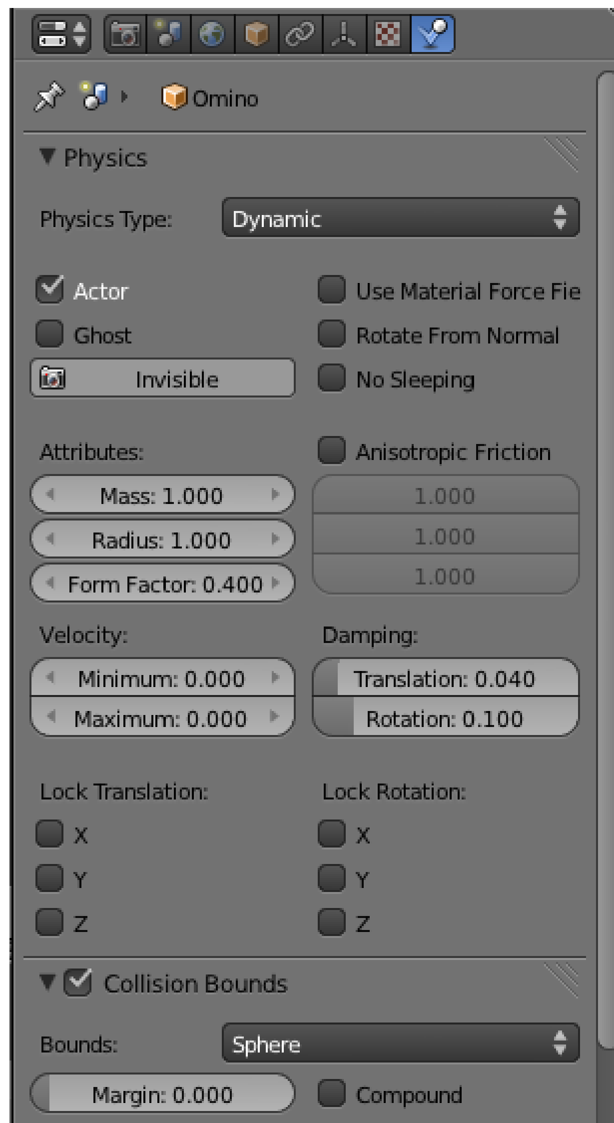


Dopo l'impostazione la Camera piglia e se ne va a Forlimpopoli: lo fa perchè una volta impostata come figlia di, la sua posizione e il suo orientamento sono interpretati relativamente al genitore. Nessun problema: azzeriamo la posizione e impostiamo l'orientamento a (90,0,0):



Bene. Adesso diamo al nostro Omino un volume fisico: ci serve per non affondare nel pavimento o attraversare i muri. Selezionate Omino e nel pannello delle impostazioni fisiche:

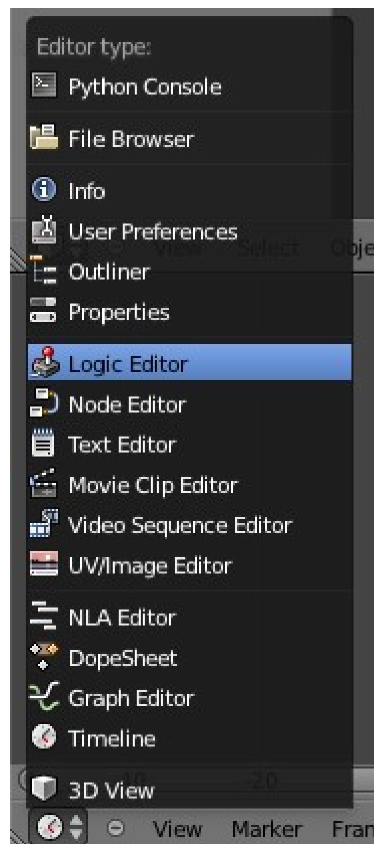
impostate Physics Type su Dynamic, selezionate la casella Collision Bounds e usate come Bounds una sfera:



Ora se spostate il cursore del mouse sul pannello della vista 3D, premete zero (vista camera) e P (avvia il motore di gioco) vi vedrete cadere delicatamente sul pavimento. Adesso ci muoviamo.

Script

Selezionate Omino e visualizzate in qualche parte della finestra di blender il pannello “Logic Editor”:



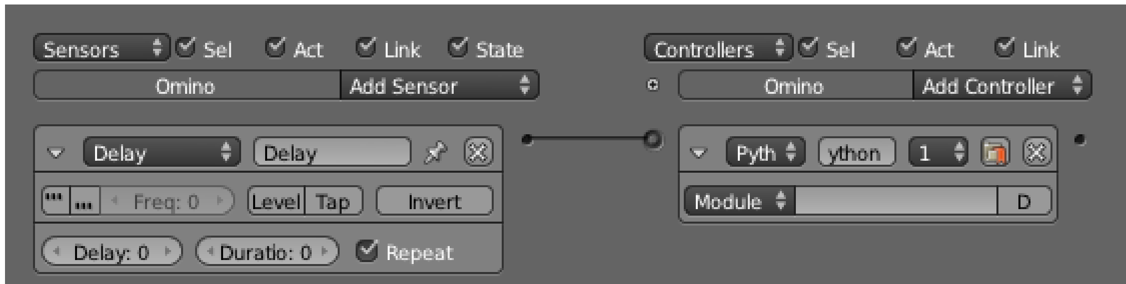
Ad esempio in basso. Vedrete una cosa del genere:

Quello che facciamo con questo pannello è aggiungere un componente logico collegato al nostro Omino. La logica del motore logico di blender è tripartita: i sensori ci dicono quando capita qualcosa, i controllori decidono cosa fare e gli attuatori fanno qualcosa. A noi degli attuatori al momento non ce ne frega una mazza: dobbiamo semplicemente dire al nostro blender di eseguire uno script per ogni ciclo logico del motore di gioco.

Aggiungete un sensore di tipo Delay impostato su Repeat:

Aggiungete un controllore di tipo Python impostato su Module:

Collegate il sensore al controllore:



Scrivete nella casella di testo di fianco a Module “gioco.main”.

Qui “gioco” è il nome del modulo python, “main” è nome della funzione che il motore di gioco invocherà quando il sensore invierà un impulso – nel nostro caso ad ogni ciclo logico.

Il nome del modulo può essere il nome di un modulo che esiste nella memoria di blender (un'unità di testo collegata) o il nome di un file py che esiste nella stessa cartella in cui si trova il file blender in lavorazione. Quindi qui “gioco” fa riferimento al file “gioco.py” che noi abbiamo creato nella nostra cartella.

Usiamo un file e non un'unità di testo nella memoria di blender perchè già mi tocca usare python, se poi devo farlo usando l'editor di testo integrato in blender chiamo la polizia.

Il nostro controllore dice che considera il file gioco.py e in quel modulo python piglia la funzione “main”. Che ancora non c'è e quindi la scriviamo. Nel file gioco.py scriviamo:

```
def main(controller):
      print("hello world!")
```

e salviamo il file. Lanciando il motore di gioco (mouse sul pannello 3d, premere P) vedremo comparire sulla console una fila di “hello world!”:

Perfetto: adesso non ci resta che spulciare un po' le api del blender game engine. La cui documentazione è a spanne ma non è che si possa desiderare tutto. Per muoverci in stile FPS ci serve sapere tre cose:

1. cosa c'è nella scena, così da andare a pescare l'oggetto che vogliamo muovere
2. quando l'utente preme un tasto e quando no
3. quando l'utente muove il mouse e da che parte lo fa

Tutto è già bell'e preconfezionato quindi faremo in fretta.

Cosa c'è nella scena.

Questo ci interessa sia per muovere il nostro omino sia per sparare alle cose, raccattare i power-up e cose così. La scena attiva si ottiene invocando:

```
bge.logic.getCurrentScene()
```

Il valore restituito ha una lista di elementi contenuti nella scena:

```
scena = bge.logic.getCurrentScene()
listaOggettiNellaScena = scena.objects
```

La lista supporta una ricerca per nome:

```
scena = bge.logic.getCurrentScene()
listaOggettiNellaScena = scena.objects
omino = listaOggettiNellaScena["Omino"]
```

Dunque se scriviamo nel nostro gioco.py:

```
import bge
```

```
scena = bge.logic.getCurrentScene()
listaOggettiNellaScena = scena.objects
omino = listaOggettiNellaScena["Omino"]
print("Ho trovato: ", omino)

def main(controller):
      pass;
```

e lanciamo il gioco, sulla console apparirà una cosa del genere:

```
Blender Game Engine Started
Ho trovato:  Omino
Blender Game Engine Finished
```

E qui siamo a posto.

Tasto premuto.

Il bge (blender game engine) ha della api ad hoc per l'input da tastiera. Si può fare così:

```
import bge

tastoDaControllare = bge.events.WKEY
statoPremuto = bge.logic.KX_INPUT_ACTIVE

def main(controller):
      statoTastiera = bge.logic.keyboard.events
      tastoPremuto = statoTastiera[tastoDaControllare] == statoPremuto
      if(tastoPremuto):
            print("Stai premendo la W")
```

Avviate il gioco, premete W e vedrete una serie di "Stai premendo la W" apparire sulla console. In bge.events c'è un KEY per ogni tasto.

Movimento del mouse.

Non c'è un precotto che ci dica "hey, il mouse si è spostato a destra di un punto e in giù di tre" quindi bisogna adottare la strategia classica:

1. quando il gioco parte ficca il mouse al centro dello schermo
2. ad ogni ciclo
   1. piglia la posizione corrente del mouse
   2. calcola lo spostamento D rispetto al centro dello schermo
   3. rificca il mouse al centro dello schermo

Sarà poi lo spostamento D a dirci da che parte il mouse si sia mosso (segno di d.x e d.y) e, eventualmente, di quanto si sia mosso (valore di d.x e d.y).

Qui le API devono dirci e farci fare un sacco di cose. Vediamole una per una.

Quanto è grande la finestra di gioco e dove sta il suo centro?

```
larghezzaFinestra = bge.render.getWindowWidth()
```

```
altezzaFinestra = bge.render.getWindowHeight()
centroX = larghezzaFinestra / 2
centroY = altezzaFinestra / 2
```

Come faccio a spostare il mouse al centro della finestra di gioco?

```
bge.render.setMousePosition(int(centroX), int(centroY))
```

Dove sta il mouse adesso?

```
mouseX = larghezzaFinestra * bge.logic.mouse.position[0]
mouseY = altezzaFinestra * bge.logic.mouse.position[1]
```

Come calcolo lo spostamento del mouse rispetto al centro della finestra?

```
mouseDX = mouseX – centroX
mouseDY = mouseY – centroY
```

Insomma, tutto fatto anche se abbiamo dovuto scriverlo.

Per testare il tutto scriviamo nel nostro gioco.py un programma che stampi sullo schermo una stringa con la direzione (su, giù, sinistra, destra) dello spostamento del mouse.

```
import bge

sogliaRilevamento = 2
larghezzaFinestra = bge.render.getWindowWidth()
altezzaFinestra = bge.render.getWindowHeight()
centroX = int(larghezzaFinestra / 2)
centroY = int(altezzaFinestra / 2)
bge.render.setMousePosition(centroX, centroY)

def main(controller):
	mouseX = larghezzaFinestra * bge.logic.mouse.position[0]
	mouseY = altezzaFinestra * bge.logic.mouse.position[1]
	mouseDirX = mouseX - centroX
	mouseDirY = mouseY - centroY
	mouseDX = abs(mouseDirX)
	mouseDY = abs(mouseDirY)
	bge.render.setMousePosition(centroX, centroY)

	if(mouseDX > sogliaRilevamento):
		if(mouseDirX > 0): print("DESTRA")
		elif(mouseDirX < 0): print("SINISTRA")

	if(mouseDY > sogliaRilevamento):
		if(mouseDirY < 0): print("ALTO")
		elif(mouseDirY > 0): print("BASSO")
```

Qui di diverso c'è solo la “sogliaRilevamento”. Che si può tradurre così: se stiamo lì a guardare la differenza di un pixel finisce che tremoliamo come uno col delirium tremens. Controlliamo se il mouse si è mosso di almeno uno zic confrontando lo spostamento assoluto del mouse lungo un asse (cioè senza il segno, il destra o sinistra) con la soglia: se passa, allora si può usare, altrimenti ignoriamo.

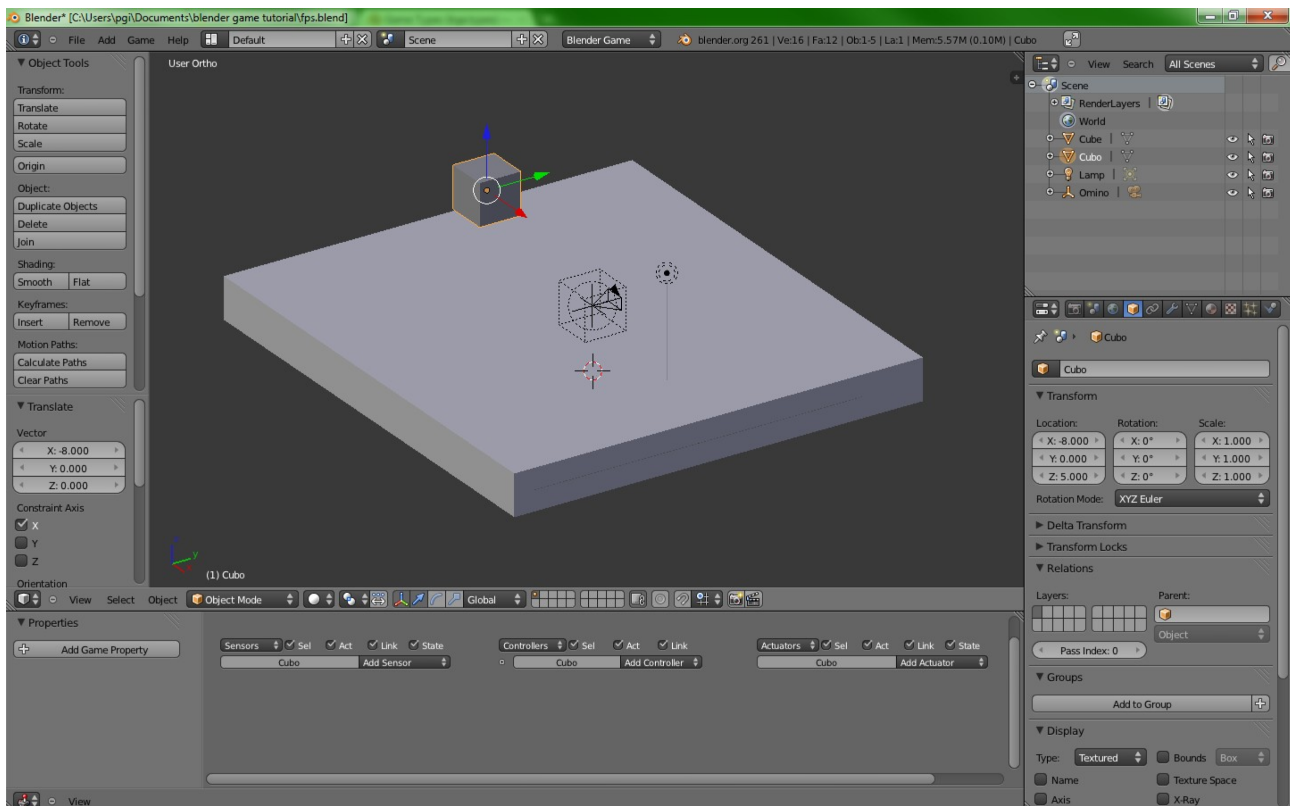
Combiniamo questo codice con quello che sappiamo della pressione dei tasti, stampando sempre stringhe sullo schermo. Otterremo lo scheletro al quale collegare l'applicazione di rotazioni e spostamenti – che vedremo tra poco. Salta fuori, ad esempio:

```
import bge

sogliaRilevamento = 2
larghezzaFinestra = bge.render.getWindowWidth()
altezzaFinestra = bge.render.getWindowHeight()
centroX = int(larghezzaFinestra / 2)
centroY = int(altezzaFinestra / 2)
bge.render.setMousePosition(centroX, centroY)

def main(controller):
	#mouse
	mouseX = larghezzaFinestra * bge.logic.mouse.position[0]
	mouseY = altezzaFinestra * bge.logic.mouse.position[1]
	mouseDirX = mouseX - centroX
	mouseDirY = mouseY - centroY
	mouseDX = abs(mouseDirX)
	mouseDY = abs(mouseDirY)
	bge.render.setMousePosition(centroX, centroY)

	if(mouseDX > sogliaRilevamento):
		if(mouseDirX > 0): print("DESTRA")
		elif(mouseDirX < 0): print("SINISTRA")

	if(mouseDY > sogliaRilevamento):
		if(mouseDirY < 0): print("ALTO")
		elif(mouseDirY > 0): print("BASSO")

	#tastiera
	tasti = bge.logic.keyboard.events
	wPremuto = tasti[bge.events.WKEY] == bge.logic.KX_INPUT_ACTIVE
	sPremuto = tasti[bge.events.SKEY] == bge.logic.KX_INPUT_ACTIVE
	aPremuto = tasti[bge.events.AKEY] == bge.logic.KX_INPUT_ACTIVE
	dPremuto = tasti[bge.events.DKEY] == bge.logic.KX_INPUT_ACTIVE

	if(wPremuto): print("VAI AVANTI")
	if(sPremuto): print("VAI INDIETRO")
	if(aPremuto): print("VAI A SINISTRA")
	if(dPremuto): print("VAI A DESTRA")
```

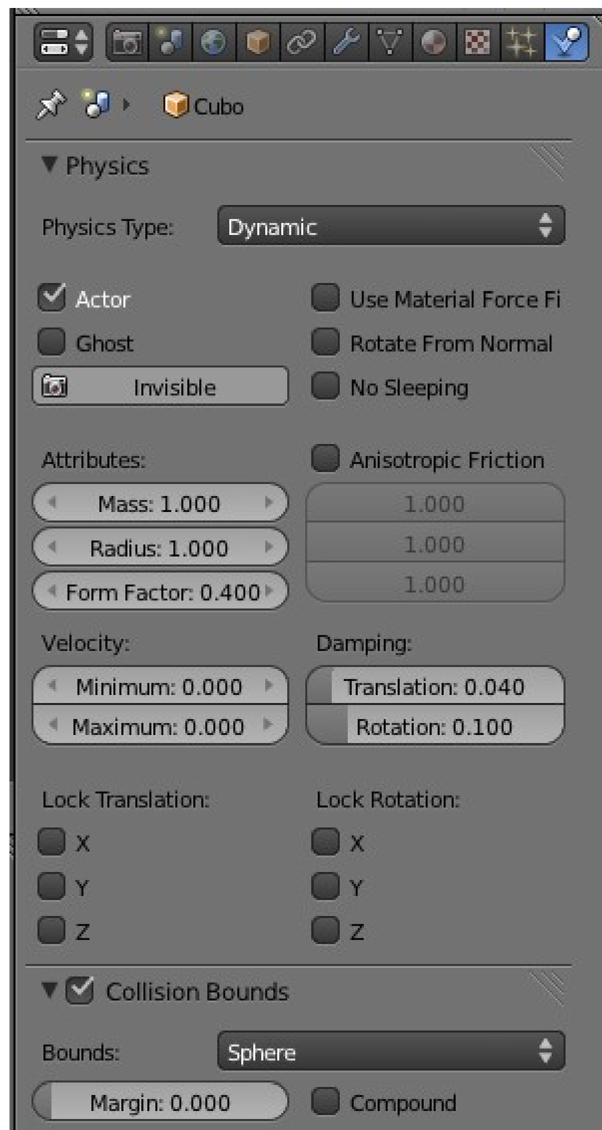
Ora se anziché scrivere stringhe spostiamo l'omino siamo a posto.

Spostamento e rotazione di un oggetto.

Qui c'è il truccone. Se un oggetto ha una controparte fisica allora possiamo spostarlo impostando la sua velocità lineare. Aggiungete un cubo alla scena, da qualche parte sopra al piano:



Chiamatelo “Cubo” e, nella scheda Physics, impostatelo come dinamico con un volume di collisione di tipo sfera (usiamo la sfera se no dopo non gira):



Adesso nel nostro script lo spostiamo in avanti pasticciando con la sua velocità lineare:

```
import bge

scene = bge.logic.getCurrentScene()
cubo = scene.objects["Cubo"]

def main(controller):
      cubo.applyMovement((0, 0.01, 0), True)
```

L'asse y muove avanti e indietro, l'asse x a destra e sinistra. A dirla tutta dipende da com'è orientata la scena ma nel pannello 3D è pieno di freccine che vi indicano la via.

Lanciando il gioco vedrete che il cubo si sposta in avanti. La rotazione funziona allo stesso modo, cambia il parametro:

```
import bge

scene = bge.logic.getCurrentScene()
cubo = scene.objects["Cubo"]

def main(controller):
      cubo.applyRotation((0, 0, 0.01), True)
```

Questo è uno “ruota intorno all'asse Z”, quello che punta in alto.

Le velocità permangono. Be', a meno della frizione ma non stiamo qui a fare gli Hawking de' noantri: azzeriamola quando secondo l'input non dovremmo muoverci.

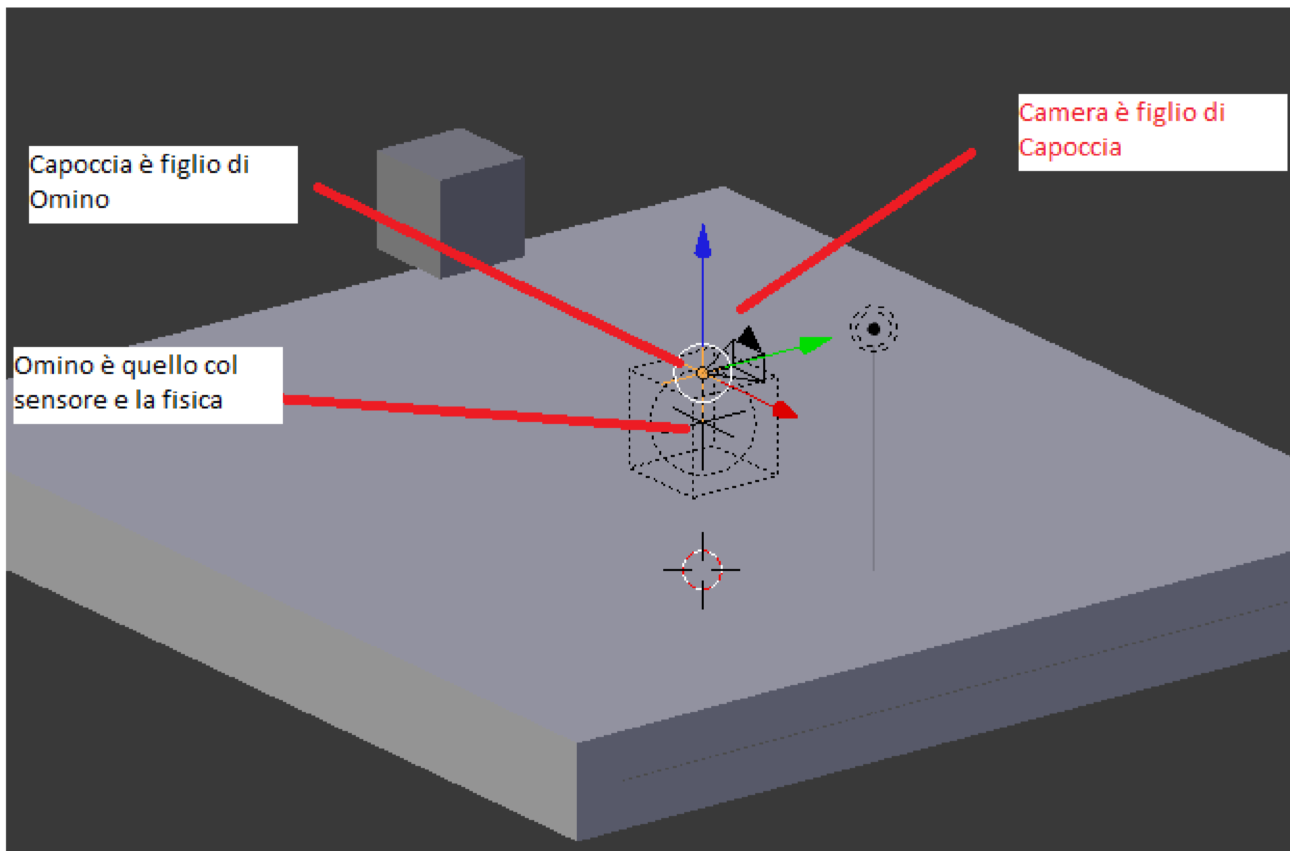
Il lavoraccio finito.

Per quanto detto, potremmo scrivere (ma non scriveremmo!):

```
import bge

sogliaRilevamento = 2
larghezzaFinestra = bge.render.getWindowWidth()
altezzaFinestra = bge.render.getWindowHeight()
centroX = int(larghezzaFinestra / 2)
centroY = int(altezzaFinestra / 2)
scene = bge.logic.getCurrentScene()
omino = scene.objects["Omino"]
bge.render.setMousePosition(centroX, centroY)
MSPEED = 0.01
RSPEED = 0.01

def main(controller):
	#mouse
	mouseX = larghezzaFinestra * bge.logic.mouse.position[0]
	mouseY = altezzaFinestra * bge.logic.mouse.position[1]
	mouseDirX = mouseX - centroX
	mouseDirY = mouseY - centroY
	mouseDX = abs(mouseDirX)
	mouseDY = abs(mouseDirY)
	bge.render.setMousePosition(centroX, centroY)

	vx = 0
	vy = 0
	rz = 0
	rx = 0

	if(mouseDX > sogliaRilevamento):
		if(mouseDirX > 0): rz = -RSPEED
		elif(mouseDirX < 0): rz = RSPEED

	if(mouseDY > sogliaRilevamento):
		if(mouseDirY < 0): rx = RSPEED
		elif(mouseDirY > 0): rx = -RSPEED

	#tastiera
	tasti = bge.logic.keyboard.events
	wPremuto = tasti[bge.events.WKEY] == bge.logic.KX_INPUT_ACTIVE
	sPremuto = tasti[bge.events.SKEY] == bge.logic.KX_INPUT_ACTIVE
	aPremuto = tasti[bge.events.AKEY] == bge.logic.KX_INPUT_ACTIVE
	dPremuto = tasti[bge.events.DKEY] == bge.logic.KX_INPUT_ACTIVE

	if(wPremuto): vy = MSPEED
	if(sPremuto): vy = -MSPEED
	if(aPremuto): vx = -MSPEED
	if(dPremuto): vx = MSPEED

	omino.applyMovement((vx, vy, 0), True)
	omino.applyRotation((rx, 0, rz), True)
```

Infatti se lo scrivessimo (e non l'avessimo fatto!) scopriremmo che l'omino si imbarca quando gira. Adesso non sto qui a fare il matematico che non è proprio il caso ma dipende dall'ordine delle rotazioni: basta dare una testa all'omino.

Aggiungiamo un empty "Capoccia" alla scena, impostiamo come Parent di Capoccia Omino e come Parent di Camera Capoccia. Capoccia, la ruotiamo in su e in giù. Omino a destra e sinistra ed è bell'e che finita. Per esser chiari:



Quindi nel codice prendiamo capoccia nell'intestazione:

```
capoccia = scene.objects["Capoccia"]
```

e nel main, in fondo, applichiamo a Omino la rotazione destra-sinistra e a Capoccia quella in alto in basso:

```
      omino.applyRotation((0, 0, rz), True)
      capoccia.applyRotation((rx,0,0), True)
```

Questo risolve anche un problema con l'arrampicamento sui muri (se il corpo ruota verso l'alto per via della frizione "arrampica" o una cosa così).

A questo punto si può anche fare un po' di chirurgia estetica sul codice. Poichè il main è idealmente destinato a contenere un sacco di cose, è meglio se scriviamo una cosa di questo genere:

```
import fps

fpsController = fpsutils.FPSNavigator().setBody("Omino").setHead("Capoccia")

def main(controller):
```

```
	fpsController.update()
```

O come preferite, purchè non sia un volume della treccani dove a malapena metteremmo uno spillo. Basta un modulo a parte, che noi salviamo sempre nella stessa cartella del file blender.

```
#fpsutils.py
import bge

WIN_HEIGHT = bge.render.getWindowHeight()
WIN_WIDTH = bge.render.getWindowWidth()
WIN_CX = int(WIN_HEIGHT / 2)
WIN_CY = int(WIN_WIDTH  / 2)

def restoreMouseLocation():
	bge.render.setMousePosition(WIN_CX, WIN_CY)

def findGameObjectByName(gameObjectName):
	return bge.logic.getCurrentScene().objects[gameObjectName]

def signOf(number):
	return (number < 0) - (number > 0)

def isKeyDown(bgeEventsKey):
	return bge.logic.keyboard.events[bgeEventsKey]==bge.logic.KX_INPUT_ACTIVE

class FPSNavigator:

	def __init__(self):
		self.body = None
		self.head = None
		self.mouseThreshold = 4
		self.rspeed = 0.01
		self.mspeed = 0.01
		self.forth = bge.events.WKEY
		self.back = bge.events.SKEY
		self.left = bge.events.AKEY
		self.right = bge.events.DKEY

	def setMotionSpeed(self, floatValue):
		self.mspeed = floatValue
		return self

	def setRotationSpeed(self, floatValue):
		self.rspeed = floatValue
		return self

	def setHead(self, gameObjectName):
		self.head = findGameObjectByName(gameObjectName)
		return self

	def setBody(self, gameObjectName):
		self.body = findGameObjectByName(gameObjectName)
		return self

	def setForthKey(self, bgeEventsKey):
		self.forth = bgeEventsKey
		return self

	def setBackKey(self, bgeEventsKey):
		self.back = bgeEventsKey
		return self

	def setLeftkey(self, bgeEventsKey):
```

```
        self.left = bgeEventsKey
        return self

    def setRightKey(self, bgeEventsKey):
        self.right = bgeEventsKey
        return self

    def update(self):
        mp = bge.logic.mouse.position
        mx = mp[0] * WIN_WIDTH
        my = mp[1] * WIN_HEIGHT
        dx = (mx - WIN_CX)
        dy = (my - WIN_CY)
        dx = signOf(dx) * (abs(dx) > self.mouseThreshold)
        dy = signOf(dy) * (abs(dy) > self.mouseThreshold)
        self.body.applyRotation((0,0,dx * self.rspeed), True)
        self.head.applyRotation((dy * self.rspeed,0,0), True)
        dx = isKeyDown(self.right) - isKeyDown(self.left)
        dy = isKeyDown(self.forth) - isKeyDown(self.back)
        self.body.applyMovement((dx*self.mspeed, dy*self.mspeed, 0), True)
        restoreMouseLocation()
```

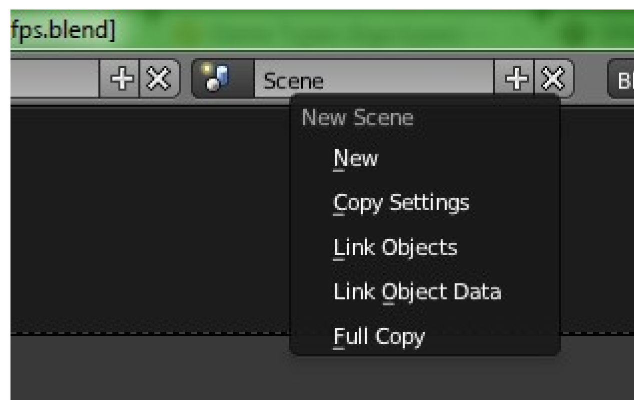
E il main diventa, come detto:

```
import fpsutils

fps = fpsutils.FPSNavigator().setHead("Capoccia").setBody("Omino")
fps.setMotionSpeed(0.025)

def main(controller):
    fps.update()
```

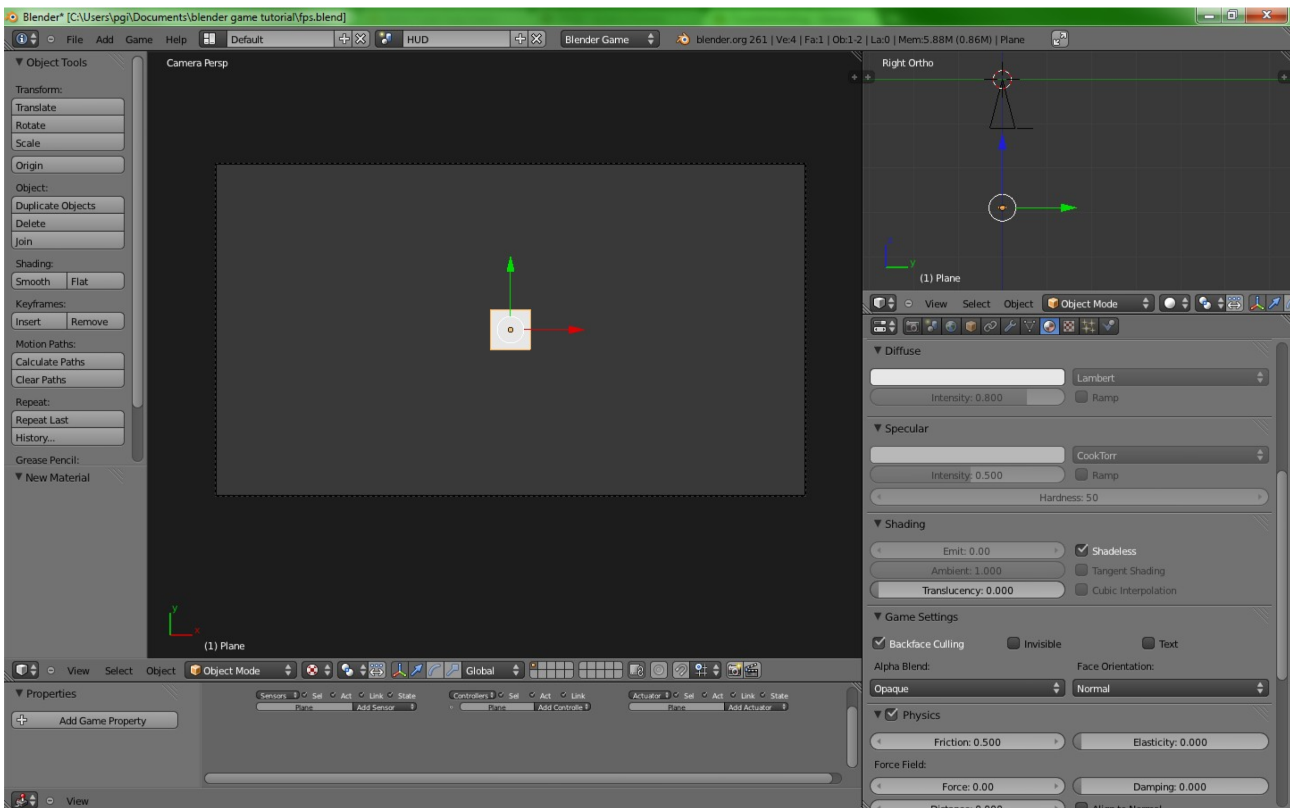
Che è già meno rivoltante. Per provarlo premete zero sul tastierino numerico per impostare la vista sulla camera corrente, poi premete P per avviare il gioco.

Sparare.

Per sparare ci vuole almeno un mirino se no è veramente triste. Per il mirino ci vuole l'HUD. L'HUD in blender è più semplice da fare che da spiegare, io poi non sono un grafico quindi è tutta da ridere. Si tratta di creare una seconda scena (c'è il pulsante in alto per aggiungere un'altra), aggiungere una telecamera se non c'è, e poi piazzare di fronte alla telecamera tutto quello che ci pare, nel nostro caso un quadrato con una texture a mirino. Poi la proiezione della scena come HUD si fa un attuatore. Iniziamo creando la nuova scena:

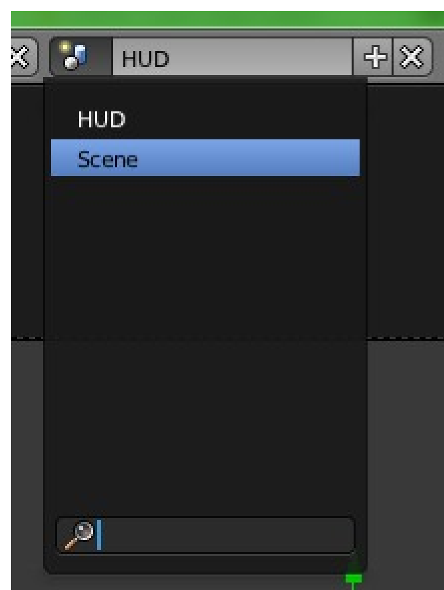


Chiamiamola HUD. Poi Add → Camera. Non importa com'è orientata la camera, quella appena aggiunta guarda in basso: va bene così. Add → Mesh → Plane e lo spostate di fronte alla telecamera. Meglio se lavorate con due pannelli 3D, uno che proietta la scena dal punto di vista della camera, uno con cui spostare le cose.



Quello che vedete attraverso la camera in questa scena sarà l'hud del gioco (quando avremo impostato l'attuatore). Io tengo il quadrato bianco perchè per metterci un'immagine trasparente che non faccia totalmente schifo mi servono almeno un paio di mesi voi divertitevi.

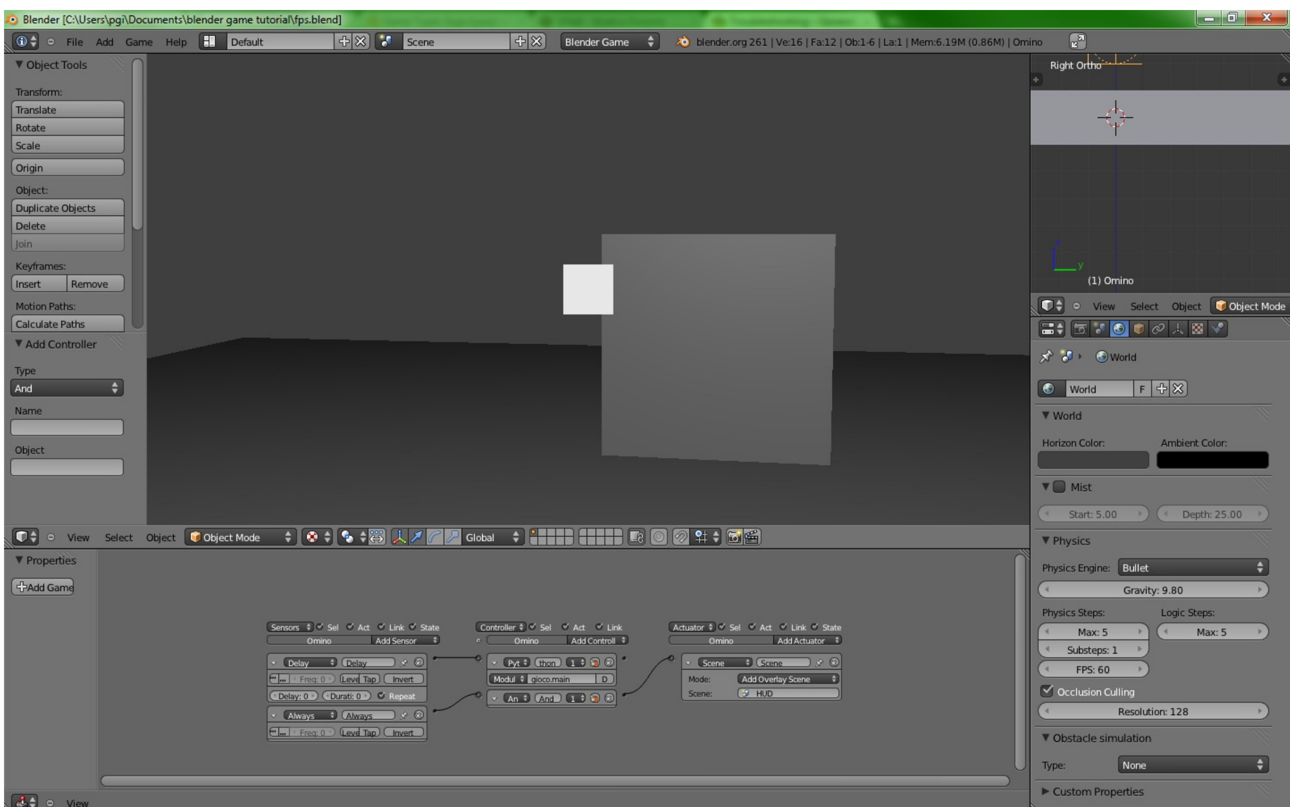
Torniamo alla scena precedente, quella del gioco:



e nel Logic Editor (il pannello in basso con il sensore e il controllo) aggiungiamo un sensore Always e un attuatore Scene. Colleghiamo il sensore always direttamente all'attuatore scene:

blender ci aggiunge un controller and:

Impostiamo l'attuatore Scene to Add Overlay Scene e nel campo Scene selezioniamo la scena HUD. E questo è quando. Un po' macchinoso ma ci sono dei vantaggi nell'avere una scena come hud, ad esempio la possibilità di manipolarne gli oggetti con le stesse api con cui maneggiamo il gioco 3d. Questo è quello che salta fuori a gioco avviato:



Come si spara. Il procedimento è, al solito, quasi banale:

1. quando premo il pulsante sinistro del mouse
2. se ho qualcosa nel mirino ed è un bersaglio valido
3. allora eliminalo dalla scena

Poi si può ricamare: i se potrebbero essere più d'uno (se ho un'arma, se sono entro una certa distanza, se ho munizioni eccetera) e gli allora variegati (danneggia il bersaglio, riproduci un suono, elimina il bersaglio se la sua vita scende sotto un certo valore). Se ne possono pensare di tutti i colore: ad esempio se sparassi un proiettile incediario il bersaglio prenderebbe fuoco quindi anziché colpirlo e via dovrei far partire una procedura che scarica la vita della vittima un po' per volta eccetera eccetera. Tutto è possibile e di solito nemmeno troppo complicato. Vediamo i passaggi.

Quando premo il pulsante sinistro del mouse.

Funziona un po' come per i pulsanti della tastiera.

```
import bge

def main(controller):
      mouse = bge.logic.mouse
      leftDown = mouse.events[bge.events.LEFTMOUSE] == bge.logic.KX_INPUT_ACTIVE
      if(leftDown):
            print("left down!")
```

Questo stampa “left down!” quando e finchè il pulsante sinistro è premuto.

Se ho qualcosa nel mirino.

Qui si piglia la telecamera (è un FPS) e si dice semplicemente:

```
target = camera.getScreenRay(0.5, 0.5, 100)
```

La funzione getScreenRay con i parametri 0.5,0.5,100 spara un raggio dal centro dello schermo (0.5, 0.5) in direzione della telecamera e restituisce il primo oggetto nella scena che interseca quel raggio entro 100 unità di distanza. Otteniamo la telecamera attiva con:

```
camera = bge.logic.getCurrentScene().active_camera
```

Se riprendiamo il codice che usava fpsutils e aggiungiamo il pezzo del mouse combinato con il raggio, possiamo già vedere il risultato:

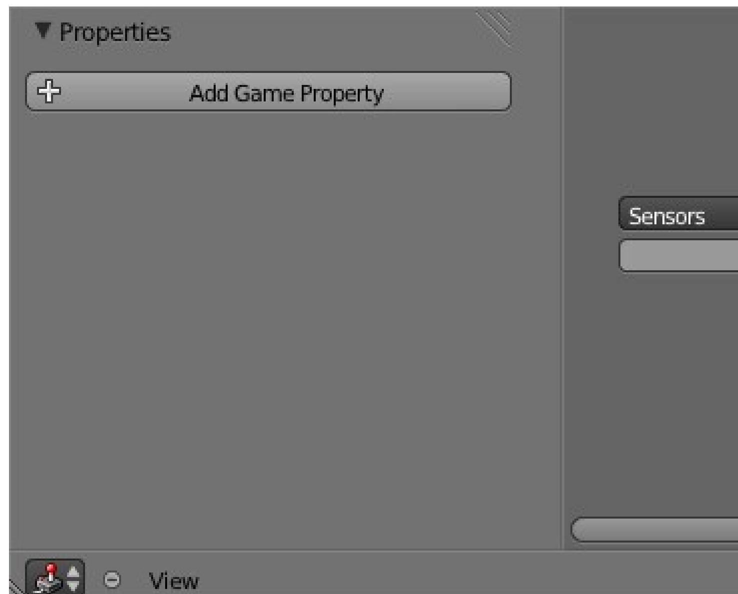
```
import fpsutils, bge

fps = fpsutils.FPSNavigator().setHead("Capoccia").setBody("Omino")
fps.setMotionSpeed(0.025)

def main(controller):
      fps.update()

      mouse = bge.logic.mouse
      leftDown = mouse.events[bge.events.LEFTMOUSE] == bge.logic.KX_INPUT_ACTIVE
      if(leftDown):
            camera = bge.logic.getCurrentScene().active_camera
            target = camera.getScreenRay(0.5, 0.5, 100)
            print("Colpito: ", target)
```

Se è un bersaglio valido.

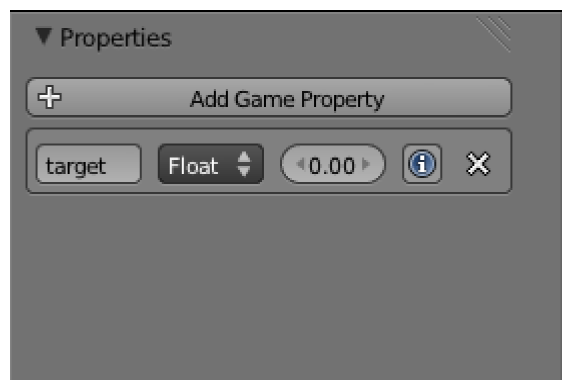
Cosa sia un bersaglio valido e cosa no dipende dal gioco. Una volta stabilito che certe cose sono bersagli ad altre no, possiamo ficcarlo nel gioco in tanti modi. Ad esempio potremmo dire che tutte le cose che si chiamano “target” qualcosa siano bersagli:

```
            if((target != None) and (target.name.startswith("target"))):
                  print("colpito")
```

Oppure possiamo usare le proprietà degli oggetti di gioco. Nel pannello dei sensori, a sinistra, c'è una sezione Properties con un pulsante “Add Game Property”:



Cosa ci si possa fare è il segreto di Pulcinella: si impostano proprietà per gli oggetti (coppie chiave-valore). Basta selezionare l'oggetto che vogliamo che sia un bersaglio – io uso il cubo volante, premere add game property e dare un nome convenzionale alla proprietà (che può essere numerica o di testo), ad esempio chiamiamola... target =D.



A queste proprietà si accede con il metodo “get(nome, valore)” dell'oggetto. Il nome è il nome della proprietà, il valore è il valore restituito nel caso in cui quella particolare proprietà non esista. Per intenderci, l'oggetto a cui ho applicato la proprietà si chiama Cubo, io posso dire:

```
cubo = bge.logic.getCurrentScene().objects[“Cubo”]
x = cubo.get(“target”, “Bingo”)
y = cubo.get(“pippo”, 120)
```

x vale 0.0, perchè la proprietà c'è e vale quello, y vale 120 perchè non c'è una proprietà “pippo” e io ho passato 120 come valore predefinito.

Quindi se il getScreenRay mi restituisce un valore diverso da None e quel KX_GameObject (questo è il tipo python degli oggetti presenti nella scena di gioco) ha una proprietà che si chiama “target” ho il mio bersaglio:

```
import fpsutils, bge
```

```
fps = fpsutils.FPSNavigator().setHead("Capoccia").setBody("Omino")
fps.setMotionSpeed(0.025)

def main(controller):
	fps.update()

	mouse = bge.logic.mouse
	leftDown = mouse.events[bge.events.LEFTMOUSE] == bge.logic.KX_INPUT_ACTIVE
	if(leftDown):
		camera = bge.logic.getCurrentScene().active_camera
		target = camera.getScreenRay(0.5, 0.5, 100)
		checkValidTarget(target)

def checkValidTarget(gameObject):
	if(gameObject != None):
		valid = gameObject.get("target", None) != None
		if(valid):
			print("Ho colpito un bersaglio")
```

L'opzione B passa direttamente da getScreenRay che ha un quarto parametro opzionale pari al nome della proprietà che un oggetto deve possedere per poter essere considerato intersecato dal raggio. Cioè quello che ho scritto qui sopra equivale a dire:

```
import fpsutils, bge

fps = fpsutils.FPSNavigator().setHead("Capoccia").setBody("Omino")
fps.setMotionSpeed(0.025)

def main(controller):
	fps.update()

	mouse = bge.logic.mouse
	leftDown = mouse.events[bge.events.LEFTMOUSE] == bge.logic.KX_INPUT_ACTIVE
	if(leftDown):
		camera = bge.logic.getCurrentScene().active_camera
		target = camera.getScreenRay(0.5, 0.5, 100, "target")
		if(target != None):
			print("Ho colpito un bersaglio")
```

Che è più comodo.

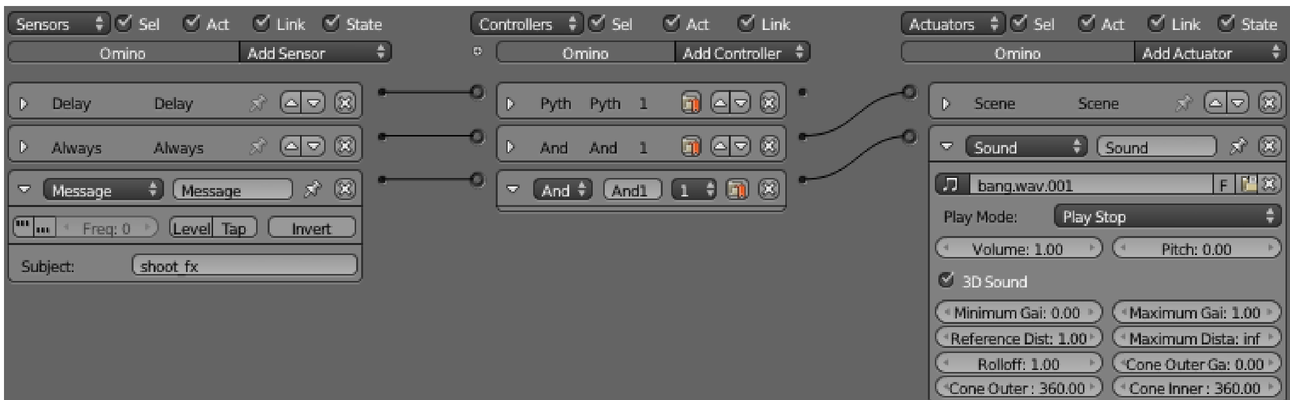
Allora eliminalo dalla scena.

Per rimuovere un oggetto dalla scena basta invocare il suo metodo endObject():

```
		if(target != None):
			target.endObject()
```

Suono.

Qui ci potremmo scrivere un'enciclica. Forse il modo più semplice è creare un sensore di tipo Message collegato ad un attuatore di tipo Sound:



Un sensore Message è una sorta di interruttore pronto a "scattare" quando sparate nel sistema una certa stringa. La stringa si imposta con il campo Subject del sensore. Il mio Message ha come stringa "shoot_fx" così quando nel codice scrivo:

```
bge.logic.sendMessage("shoot_fx")
```

il sensore si attiva ed esegue l'attuatore sound. L'attuatore Sound riproduce un suono. Per uno sparo di pistola il suo "Play Mode" dovrebbe essere "Play End", che significa "una volta che hai iniziato a suonare, vai fino alla fine della traccia anche se il sensore non è più attivo": il bang fa bang anche se ormai il grilletto l'ho mollato. Se fosse un, chessò, laser, allora il Play Mode dovrebbe essere "Play Stop", vale a dire "suona finchè il sensore è attivo" e noi dovremmo continuare a mandare il messaggio "shoot_fx" finchè intendiamo suonare.

Bando alle ciance, passiamo al codice. Aggiungiamo una classe di utilità a fpsutils, per la gestione dell'input del mouse:

```
#in fpsutils.py
#Utilità per la gestione dell'input del mouse
class MouseInput:

      def __init__(self):
            self.leftMouseClickCallbacks = []
            self.leftMouseDownCallbacks = []
            self.leftMouseUpCallbacks = []
            self.leftWasDown = False

      def update(self):
            me = bge.logic.mouse.events
            leftDown = me[bge.events.LEFTMOUSE] == bge.logic.KX_INPUT_ACTIVE
            if(leftDown and (not self.leftWasDown)):
                  self.leftWasDown = True
                  for c in self.leftMouseClickCallbacks: c()
            elif(self.leftWasDown and (not leftDown)):
                  self.leftWasDown = False
                  for c in self.leftMouseUpCallbacks: c()
            elif(leftDown):
                  for c in self.leftMouseDownCallbacks: c()

      def addLeftClickCallback(self, fun):
            self.leftMouseClickCallbacks.append(fun)

      def addLeftDownCallback(self, fun):
            self.leftMouseDownCallbacks.append(fun)

      def addLeftUpCallback(self, fun):
            self.leftMouseUpCallbacks.append(fun)
```

Nel metodo update() MouseInput si arrabatta per capire se il pulsante sinistro del mouse è premuto, rilasciato o cliccato. Qui per semplicità prendo il click come il primo di una serie di eventi mouse premuto, a rigore bisogna prenderlo al rilascio se il tempo intercorso rispetto alla prima pressione sia minore di una soglia... non stiamo qui a scrivere l'almanacco del giorno prima, dopo e durante.

Per farlo funzionare, si invoca ciclicamente il suo metodo update(), come per FPSNavigator, e si aggiungono ai metodi add che interessano le funzioni che vogliamo siano richiamate.

Il codice del gioco che riproduce lo sparo e abbatte il bersaglio è questo:

```
import fpsutils, bge

fps = fpsutils.FPSNavigator()
fps.setMotionSpeed(0.025).setHead("Capoccia").setBody("Omino")
mouseInput = fpsutils.MouseInput()
mouseInput.addLeftClickCallback(lambda:shoot())

def main(controller):
	fps.update()
	mouseInput.update()

def shoot():
	bge.logic.sendMessage("shoot_fx") #suono dello sparo
	camera = bge.logic.getCurrentScene().active_camera
	target = camera.getScreenRay(0.5, 0.5, 100, "target")
	if(target != None): target.endObject()
```

Cambiare il suono di un SoundActuator.

Cambia l'arma, cambia il suono. Farlo è roba da nulla: basta cambiare il valore della proprietà sound del SoundActuator. Il valore deve essere un aud.Factory:

```
import aud  #se non "import aud" almeno una volta
            #prima di usare Factory, blender va in crash

filename = qualche nome di file audio
sa = un qualche SoundActuator
fac = aud.Factory(filename)
sa.sound = fac
```

Tutto qua. Come si arriva in codice al SoundActuator? Si passa per l'oggetto a cui è collegato, cioè l'oggetto nell'editor di blender nella cui scheda dei mattoncini logici compare quel SoundActuator. Nel nostro caso è Omino. Se il SoundActuator si chiama "Weapon" (il nome è stabilito nel pannello della logica) allora dirò:

```
gameObject = bge.logic.getcurrentScene().objects["Omino"]
soundActuator = gameObject.actuators["Weapon"]
```

Supponendo di voler cambiare arma quando l'utente preme i tasti 1 o 2, possiamo quindi dire una cosa di questo tipo:

```
import fpsutils, bge, aud

weaponSounds = ["bang.wav", "gunshot.wav"]
```

```
fps = fpsutils.FPSNavigator()
mouseInput = fpsutils.MouseInput()
keyInput = fpsutils.KeyInput()
soundFun = fpsutils.SoundFun("Omino", "Weapon")

fps.setMotionSpeed(0.025).setHead("Capoccia").setBody("Omino")
mouseInput.addLeftClickCallback(lambda:shoot())
keyInput.addKeyPressedCallback(bge.events.TWOKEY, lambda:changeWeapon(1))
keyInput.addKeyPressedCallback(bge.events.ONEKEY, lambda:changeWeapon(0))

def main(controller):
	fps.update()
	mouseInput.update()
	keyInput.update()

def shoot():
	bge.logic.sendMessage("shoot_fx") #suono dello sparo
	camera = bge.logic.getCurrentScene().active_camera
	target = camera.getScreenRay(0.5, 0.5, 100, "target")
	if(target != None): target.endObject()

def changeWeapon(index):
	soundFun.replaceSound(weaponSounds[index])
```

Le novità sono evidenziate. Tutta roba della nonna, sono semplici funzioni personalizzate definite in un altro modulo (sempre fpsutils) giusto per concentrarci sul succo. KeyInput (aggiunto a fpsutils.py) è fatto così:

```
#Utilità per la gestione dell'input da tastiera
class KeyInput:

	def __init__(self):
		self.triggers = []

	def update(self):
		keys = bge.logic.keyboard.events
		for c in self.triggers: c.check(keys)

	def addKeyDownCallback(self, bgeEventsKey, callback):
		self.triggers.append(KeyDownTrigger(bgeEventsKey, callback))

	def addKeyPressedCallback(self, bgeEventsKey, callback):
		self.triggers.append(KeyPressedTrigger(bgeEventsKey, callback))

#Usato da KeyInput per gestire l'abbassamento di un tasto
class KeyDownTrigger:
	def __init__(self, key, callback):
		self.key = key
		self.callback = callback

	def check(self, keys):
		if(keys[self.key] == bge.logic.KX_INPUT_ACTIVE):
			self.callback()

#Usato da KeyInput per gestire la pressione di un tasto
class KeyPressedTrigger:
	def __init__(self, key, callback):
		self.key = key
		self.callback = callback
		self.wasDown = False

	def check(self, keys):
		down = keys[self.key] == bge.logic.KX_INPUT_ACTIVE
```

```
        if(down and (not self.wasDown)):
            self.wasDown = True
            self.callback()
        elif(self.wasDown and (not down)):
            self.wasDown = False
```

E SoundFun è fatto così:

```
#Sound utilities
class SoundFun:

    def __init__(self, gameObjectName, soundActuatorName):
        self.soundActuator = findGameObjectByName(gameObjectName
            ).actuators[soundActuatorName]

    def reinit(self, gameObjectName, soundActuatorName):
        self.soundActuator = findGameObjectByName(gameObjectName
            ).actuators[soundActuatorName]

    def replaceSound(self, relativeFileName):
        fac = aud.Factory(relativeFileName)
        self.soundActuator.sound = fac
```

Azione.

Le azioni sono le animazioni, via. La creazione di animazioni in blender è quasi esoterica, almeno per me. E' molto più facile farle partire una volta create. Le azioni sono associate agli oggetti e basta dire:

```
gameObject.playAction(nome, frameIniziale, frameFinale)
```

Immaginando (così, a caso) che il nostro bersaglio abbia un'animazione di nome “Drop”, che lo fa cadere a terra, per eseguirla quando è colpito aggiungeremmo al nostro codice di gioco:

```
import fpsutils, bge, aud

weaponSounds = ["bang.wav", "gunshot.wav"]
fps = fpsutils.FPSNavigator()
mouseInput = fpsutils.MouseInput()
keyInput = fpsutils.KeyInput()
soundFun = fpsutils.SoundFun("Omino", "Weapon")

fps.setMotionSpeed(0.025).setHead("Capoccia").setBody("Omino")
mouseInput.addLeftClickCallback(lambda:shoot())
keyInput.addKeyPressedCallback(bge.events.TWOKEY, lambda:changeWeapon(1))
keyInput.addKeyPressedCallback(bge.events.ONEKEY, lambda:changeWeapon(0))

def main(controller):
    fps.update()
    mouseInput.update()
    keyInput.update()

def shoot():
    bge.logic.sendMessage("shoot_fx") #suono dello sparo
    camera = bge.logic.getCurrentScene().active_camera
    target = camera.getScreenRay(0.5, 0.5, 100, "target")
    if(target != None):
        target.playAction("Drop", 0, 20)
```

```
def changeWeapon(index):
	soundFun.replaceSound(weaponSounds[index])
```

E quando l'animazione è finita? Forse c'è un evento precotto (be', dovrebbe esserci almeno), io non l'ho trovato. Ma che sarà mai! Basta un “poller”.

```
#actionpoller.py
class ActionPoller:

	def __init__(self):
		self.actionCallbackPairs = []

	#verifica se ci sono oggetti registrati che non eseguono animazioni
	#e in caso invoca le rispettive funzioni associate
	def update(self):
		removables = []
		for p in self.actionCallbackPairs:
			gameObject = p[0]
			callback = p[1]
			if(not gameObject.isPlayingAction()):
				callback(gameObject)
				self.actionCallbackPairs.remove(p)

	#aggiunge una funzione f(gameObject) invocata quando l'oggetto non
	#esegue animazioni
	def addActionEndCallback(self, gameObject, callback):
		self.actionCallbackPairs.append((gameObject, callback))
```

La logica è questa: quando colpisco un bersaglio, faccio partire l'animazione “schiatta” e aggiungo una funzione all'ActionPoller. L'ActionPoller invocherà quella funzione quando l'oggetto avrà terminato la sua animazione. Il tutto funziona grazie alla funzione predefinta evidenziata.

In gioco.py basta quindi aggiungere, in intestazione:

```
actionPoller = actionpoller.ActionPoller()
```

Più che altro perchè io ho la fissa delle classi ma si potrebbe fare anche con un modulo nudo e crudo, per poi dire in shoot():

```
def shoot():
	bge.logic.sendMessage("shoot_fx") #suono dello sparo
	camera = bge.logic.getCurrentScene().active_camera
	target = camera.getScreenRay(0.5, 0.5, 100, "target")
	if(target != None):
		target.playAction("Drop", 0, 20)
		actionPoller.addActionEndCallback(target, lambda t:t.endObject())
```

Il bersaglio cade e viene rimosso (cade sarebbe “Fall” ma m'è venuto “Drop”). C'è una minuzia relativa al fatto che se si spara al bersaglio che cade è aggiunto un secondo “callback” che cercherà di rimuovere l'oggetto quando il sistema l'avrà già eliminato. L'intoppo riguarda il fatto che l'eliminazione dell'oggetto con endObject sembra proprio gettare nel fosso il puntatore C o una cosa così: tentare di farlo due volte fa sputare a blender un errore.

Proprietà degli oggetti.

Ultima cosa: “giocare” con le proprietà degli oggetti. Abbattiamo i bersagli solo se colpiti due volte.

Gli oggetti hanno un metodo:

```
get(nome, valore predefinito)
```

che ottiene il valore di una proprietà impostata tramite il pannello della logica di gioco O restituisce il valore predefinito. Per impostare il valore di una proprietà di gioco o crearne una nuova si usa la notazione:

```
oggetto[nome] = valore
```

La stessa può essere usata per accedere al valore:

```
valore = oggetto[nome]
```

ma qui occorre che nome sia anche il nome di una proprietà esistente.

Stabilendo che tutti i notri bersagli abbiano una vita pari a 2 e che i colpi sottraggano 1, senza andare a creare un valore per ogni oggetto target, possiamo dire:

```
life = target.get(“life”, 2)
life -= 1
target[“life”] = life
```

Cioè se life non esiste allora vale due e che esista o non esista essa è comunque ridotta di uno e rificcata in target. Il metodo “shoot()” cambia così:

```
def shoot():
      bge.logic.sendMessage("shoot_fx") #suono dello sparo
      camera = bge.logic.getCurrentScene().active_camera
      target = camera.getScreenRay(0.5, 0.5, 100, "target")
      if(target != None):
            life = target.get("life", 2)
            life -= 1
            target["life"] = life
            if(life == 0):
                  target.playAction("Drop", 0, 100)
                  actionPoller.addActionEndCallback(target,
                        lambda t:t.endObject())
```

Questo risolve anche il problema dell'eliminazione di un oggetto già animato (più che altro perchè la condizione che causa l'animazione impedisce il ripetersi dell'accodamento del callback).

Bene.

Uno protrebbe andare avanti per anni, ci sono un sacco di cose che si possono-devono fare. Ma sono tutte “robine” come queste. La prossima volta vediamo come farci correre dietro da un po' di rettangoloni. Perchè il problema è sempre quello: fare un ambiente che sia decente alla vista e lì ci vuole il grafico. Animale rarissimo e schivo.