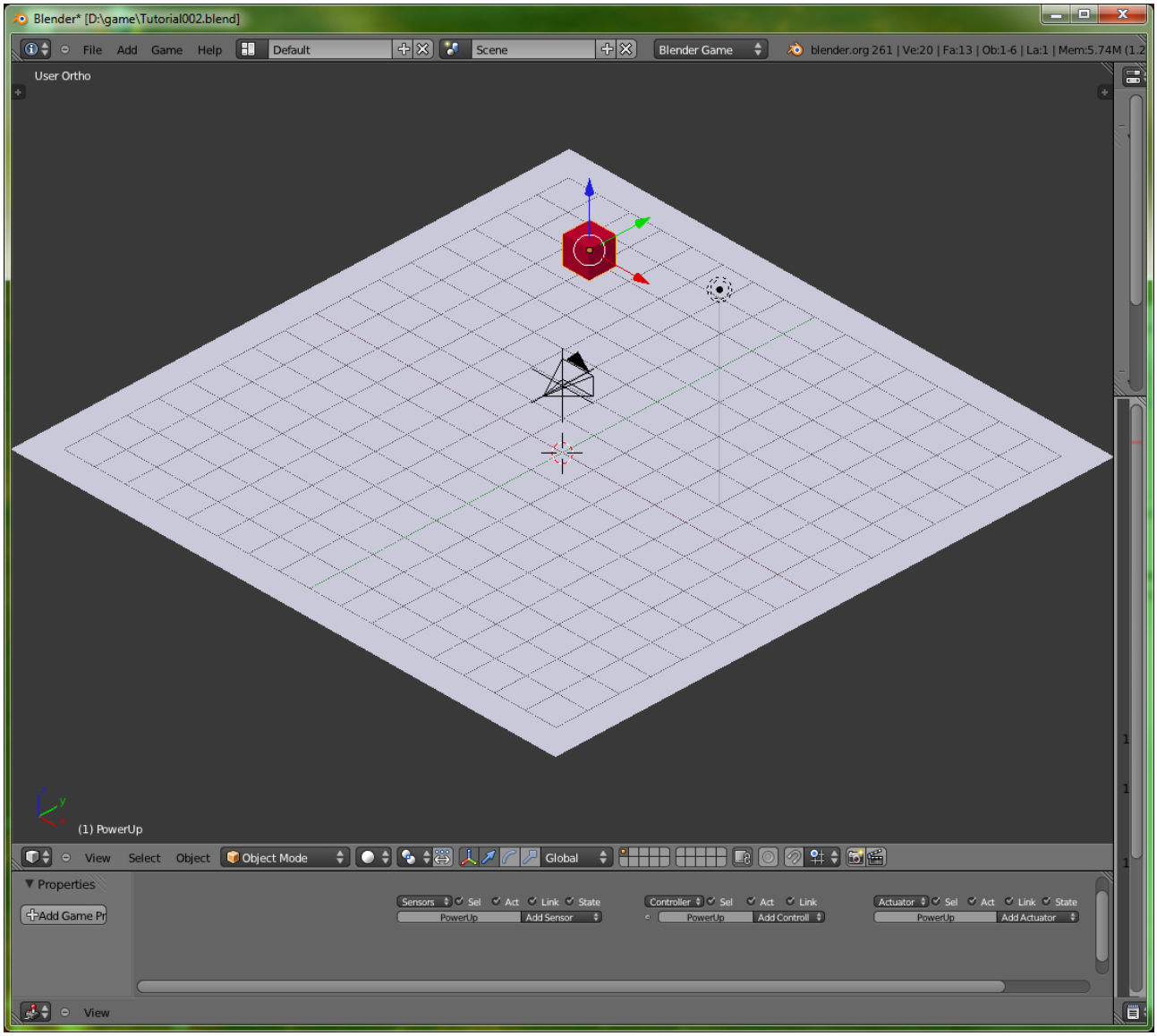
Blender FPS – Parte II

E via.

Abbiamo visto come muoverci all'interno di un livello col classico WASD + Mouse. Adesso parliamo un po' di interazione.

Raccogliere un power-up

I power-up sono quegli oggetti che l'utente può raccogliere passandogli sopra: vita, munizioni, cose così. Prendiamo una scena di base con un giocatore controllato dal navigatore fps visto l'altra volta. Aggiungiamo un cubetto da qualche parte:

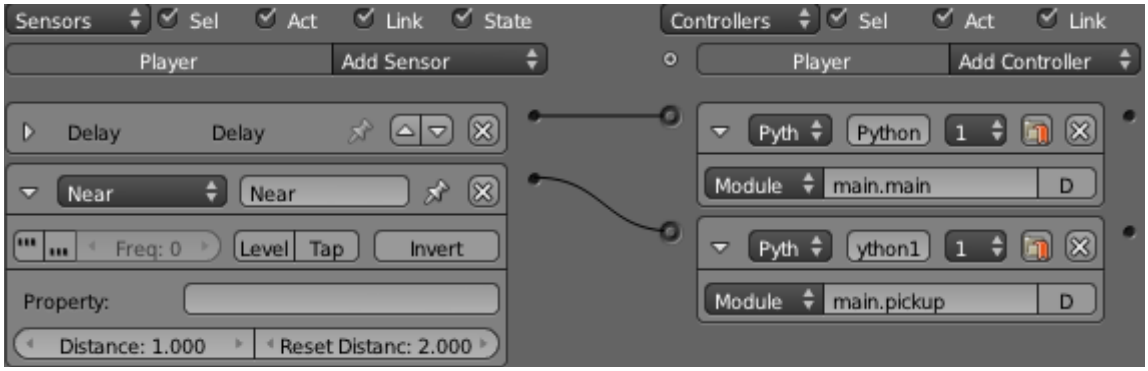


Nelle proprietà fisiche del cubo selezioniamo la casella “Actor”. Un “Actor” è qualcosa che riporta le collisioni ai sensori di prossimità.

Prendiamo il modulo python che applica la navigazione fps e aggiungiamo una funzione “pickup”:

```
import bge, fpsutils


navigator = fpsutils.FPSNavigator()
navigator.setHead("Player_head")
navigator.setBody("Player")
updatelist = [navigator]

def main(controller):
    for e in updatelist: e.update()
    pass

def pickup(controller):
    print("pick up")
    pass
```

Adesso aggiungiamo un sensore che ad ogni collisione invocherà il metodo “pickup”. Selezioniamo il corpo del giocatore, aggiungiamo un sensore di tipo “Near” e colleghiamolo ad un secondo controller di tipo Python, impostato su Module, che invoca pickup:



Da notare che a differenza dello script il modulo è inizializzato una sola volta: quando si verifica una collisione è chiamato il metodo pickup ma non è ri-eseguito il blocco di inizializzazione del modulo (cioè quello che non sta dentro alle funzioni).
Il sensore “Near” rileva sia l'avvicinamento che l'allontamento di un oggetto da un altro. A noi interessa solo quando il giocatore è abbastanza vicino, quindi scriviamo:

```
def pickup(controller):
    sensor = controller.sensors["Near"]
    if(sensor.positive):
        print("Pick up!")
        pass
    pass
```

Cosa facciamo quando il giocatore si avvicina abbastanza? Tutto dipende da cosa stiamo per raccogliere. Sappiamo cosa abbiamo avvicinato tramite la proprietà “hitObject” del sensore Near:

```
def pickup(controller):
    sensor = controller.sensors["Near"]
    if(sensor.positive):
        pickable = sensor.hitObject
        pass
    pass
```

E possiamo applicare una sorta di inversione di controllo per generalizzare il concetto di “fai qualcosa quando stai per raccattarmi”. Con questa convenzione: ogni oggetto che può essere raccolto definisce come sua proprietà una funzione del giocatore da richiamare al momento della collisione.

```
#pickup logic
def doNothing(target):
    print("Doing nothing on ", target)
    pass

#registro delle funzioni applicabili ad un oggetto "pickup"
pickup_callbacks = {"doNothing":doNothing}

def pickup(controller):
    sensor = controller.sensors["Near"]
    if(sensor.positive):
        pickable = sensor.hitObject
        functionKey = pickable.get("pickup_callback", "doNothing")
        executePickableCallback(pickable, functionKey)
        pass
    pass

def executePickableCallback(pickable, functionKey):
    pickup_callbacks.get(functionKey, doNothing)(pickable)
    pass
```

Quando il giocatore si avvicina ad un oggetto che può essere raccolto, l'oggetto determina quale funzione del giocatore deva essere eseguita. Qui

```
function = pickable.get("pickup_callback", "doNothing")
```

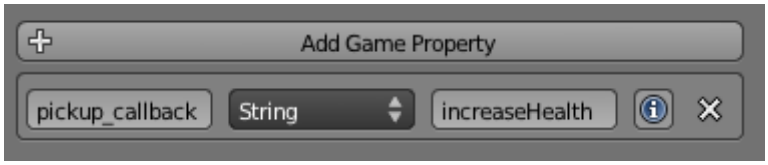
significa “dammi il valore della proprietà pickup_callback dell'oggetto rilevato dal sensore near o doNothing se quella proprietà manchi”. Quel valore è poi usato come chiave nel dizionario (di funzioni) pickup_callbacks: la funzione individuato dalla chiave è invocata, se presente, tramite la funzione executePickableCallback.

Ora dotiamo il nostro giocatore della capacità di accrescere la propria vita. Se la vita del giocatore è inferiore ad un certo massimo la aumentiamo di un certo altro valore. Il certo altro valore è il minimo tra il valore del power-up e la differenza tra la massima vita possibile e quella attuale. Se il power-up (che è di tipo vita perchè invocherà la nostra funzione di accrescimento vita) non ha un valore proprio, ne usiamo uno predefinito minimo. Insomma:

```
import bge, fpsutils

print("initializing module")

PLAYER_MAX_HEALTH = 100
DEFAULT_HEALTH_UP_VALUE = 10
KEY_HEALTH = "health"
KEY_VALUE = "value"

player = bge.logic.getCurrentScene().objects["Player"]
navigator = fpsutils.FPSNavigator()
navigator.setHead("Player_head")
navigator.setBody("Player")
updatelist = [navigator]

def main(controller):
```

```
    for e in updatelist: e.update()
    pass

#Incrementa la vita del giocatore
def increaseHealth(pickup):
    value = pickup.get(KEY_VALUE, DEFAULT_HEALTH_UP_VALUE)
    currentHealth = player.get(KEY_HEALTH, PLAYER_MAX_HEALTH)
    if currentHealth < PLAYER_MAX_HEALTH:
        currentHealth = min(PLAYER_MAX_HEALTH, currentHealth + value)
        player[KEY_HEALTH] = currentHealth
        pickup.endObject()
        pass
    print("HEALTH: ", player.get(KEY_HEALTH, PLAYER_MAX_HEALTH))
    pass

#pickup logic
def doNothing(target):
    print("Doing nothing on ", target)
    pass

#registro delle funzioni applicabili ad un oggetto "pickup"
pickup_callbacks = {"doNothing":doNothing, "increaseHealth":increaseHealth}

def pickup(controller):
    sensor = controller.sensors["Near"]
    if(sensor.positive):
        pickable = sensor.hitObject
        functionKey = pickable.get("pickup_callback", "doNothing")
        executePickableCallback(pickable, functionKey)
        pass
    pass

def executePickableCallback(pickable, functionKey):
    pickup_callbacks.get(functionKey, doNothing)(pickable)
    pass
```

Quindi selezioniamo il power-up e, tramite il pannello della logica, aggiungiamo una proprietà di nome “pickup_callbacks”, tipo String e valore “increaseHealth”:



Fine: quando saremo abbastanza vicini all'oggetto sarà invocata la funzione increaseHealth e il power-up farà il suo effetto.
Per aggiungere un tipo diverso di power-up basta definire la funzione appropriata e collegare il suo nome all'oggetto raccoglibile attraverso la proprietà pickup_callback.
Naturalmente si può procedere in modo diverso. Anzichè dare all'oggetto il nome della funzione si può usare l'oggetto come un mappa di attributi. Possiamo ad esempio stabilire, convenzionalmente, che ogni power-up abbia due attributi:

type-intero
value-intero

E che esistano alcuni valori predefiniti per type, ad esempio:

1 = VITA

2 = MUNIZIONE PER PISTOLA
3 = MUNIZIONE PER FUCILE
4 = MUNIZIONE PER FIONDA

e via discorrendo. Quando raccogliamo un oggetto andiamo a vedere di che tipo è e applichiamo il suo valore allo stato corrispondente. Come procediamo? Definiamo queste costanti in un modulo ad hoc per chiarezza d'intenti:

```
#game_constants.py
KEY_TYPE = "type"
KEY_VALUE = "value"

TYPE_UNDEFINED = 0
TYPE_LIFE = 1
TYPE_AMMO_GUN = 2
TYPE_AMMO_RIFLE = 3
TYPE_AMMO_SLING = 4
```

Definiamo una classe inventario che “conosca” i tipi di oggetto e reagisca di conseguenza:

```
#inventory.py
import game_constants as GCON


class Inventory:

    def __init__(self):
        self.data = {}
        self.pickCalls = {
            GCON.TYPE_UNDEFINED:self.doNothing,
            GCON.TYPE_LIFE:self.pickLife,
            GCON.TYPE_AMMO_GUN:self.pickGunAmmo,
            GCON.TYPE_AMMO_RIFLE:self.pickRifleAmmo,
            GCON.TYPE_AMMO_SLING:self.pickSlingAmmo}
        pass

    def doPickup(self, gameObject):
        type = gameObject.get(GCON.KEY_TYPE, GCON.TYPE_UNDEFINED)
        self.pickCalls.get(type, self.doNothing)(gameObject)
        pass

    def doNothing(self, gameObject):
        print("Unrecognized object's type")
        pass

    def pickLife(self, gameObject):
        print("pickup life")
        pass

    def pickGunAmmo(self, gameObject):
        print("pickup gun ammo")
        pass

    def pickRifleAmmo(self, gameObject):
        print("pickup rifle ammo")
        pass

    def pickSlingAmmo(self, gameObject):
        print("pickup sling ammo")
        pass
```

E usiamola nel metodo pickup del modulo principale:

```
import bge, fpsutils, inventory

inventory = inventory.Inventory()
player = bge.logic.getCurrentScene().objects["Player"]
navigator = fpsutils.FPSNavigator()
navigator.setHead("Player_head")
navigator.setBody("Player")
updatelist = [navigator]

def main(controller):
    for e in updatelist: e.update()
    pass


def pickup(controller):
    sensor = controller.sensors["Near"]
    if(sensor.positive):
        inventory.doPickup(sensor.hitObject)
        pass
    pass
```

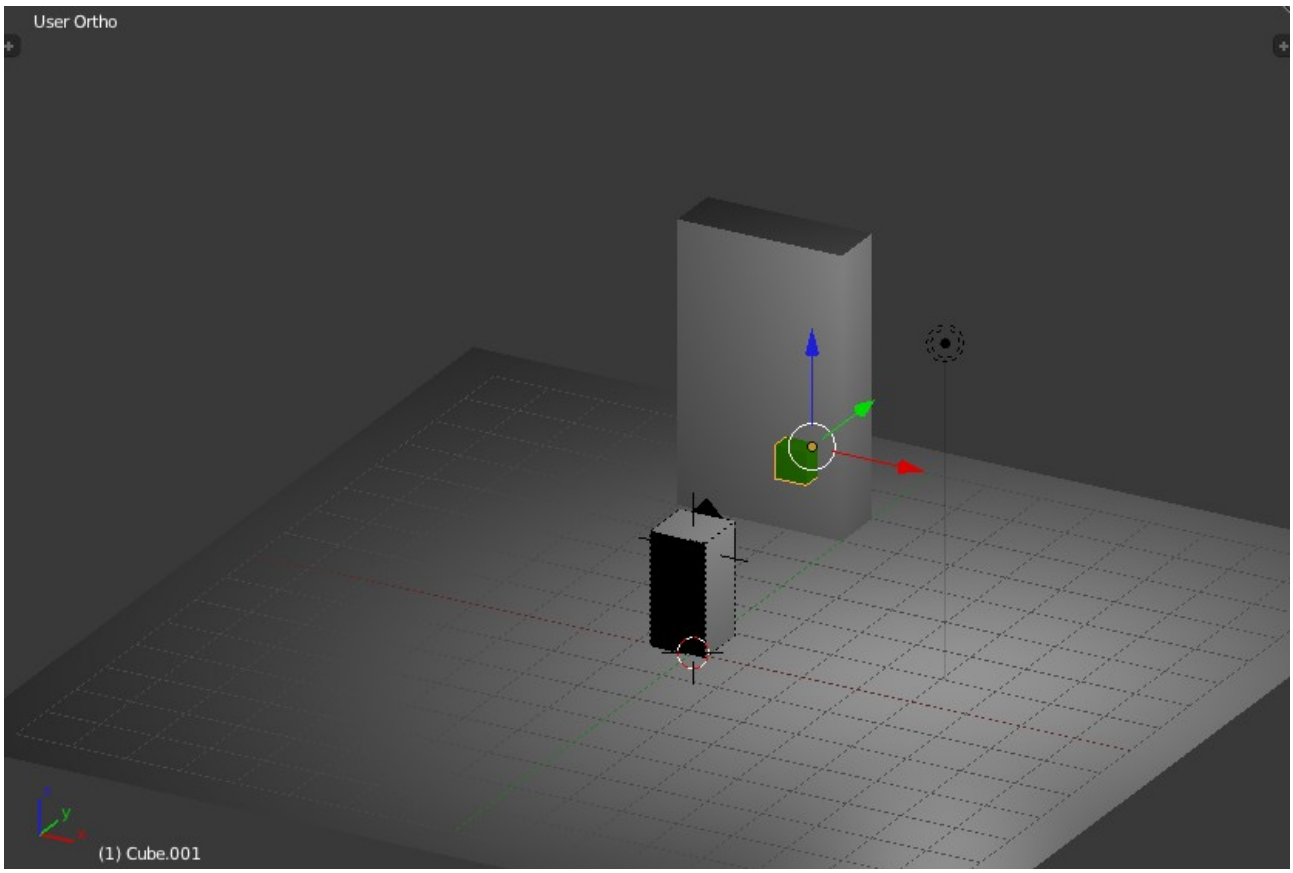
E via.

Pigia il pulsante, apri la porta.

I pulsanti sono simili ai power-up perchè l'utente deve trovarsi entro una certa distanza dall'oggetto per poter interagire. Sono diversi dai power-up perchè funzionano solo se l'utente “li sta guarndando” e preme un tasto. Non solo i pulsanti funzionano in questo modo. In generale potremmo pensare che esista una classe di oggetti “volontariamente interattivi”: non basta passarci sopra. Le condizioni che esprimono questa volontarietà sono tre:

vicino
lo guardo
premo il pulsante

Bene, iniziamo creando un bel muro con un pulsantone verde (occhio che qui Crisys 3 ci fa un baffo):



Per stabilire se l'utente sta guardando un oggetto X possiamo dire:

```
cam = bge.logic.getCurrentScene().active_camera
lookingAt = (cam.getScreenRay(0.5, 0.5, DISTANZA) == X)
```

dove DISTANZA è la soglia entro la quale deve trovarsi un oggetto per poter essere intercettato. Ci sono diversi effetti in un gioco che dipendono da ciò che l'utente sta “puntanto” in un dato istante e una routine per determinarlo può essere questa:

```
import bge, fpsutils

player = bge.logic.getCurrentScene().objects["Player"]
cam = bge.logic.getCurrentScene().active_camera
navigator = fpsutils.FPSNavigator()
navigator.mspeed = 0.02
navigator.setHead("Player_head")
navigator.setBody("Player")
updatelist = [navigator]

def main(controller):
    for e in updatelist: e.update()
    lookAt()
    pass

def doNothing(gameObject):
    print("undefined call")
    pass

def lookAt():
    gameObject = cam.getScreenRay(0.5, 0.5, 100)
    print("sto guardando:", gameObject)
    pass
```

Cosa fare quando si guardi ad un oggetto dipende da ciò che stiamo guardando. In base a cosa distinguere un oggetto da un altro è una questione di design. Supponiamo che alcuni oggetti abbiano una proprietà intera che ne qualifica il tipo e che 1 significhi “interruttore”, potremmo iniziare col dire:

```
def lookAt():
    gameObject = cam.getScreenRay(0.5, 0.5, 100)
    if gameObject != None:
        type = gameObject.get("type", None)
        doLookAt(type, gameObject)
        pass
    pass

TYPE_SWITCH = 1
def doLookAt(type, gameObject):
    if type == TYPE_SWITCH:
        print("Oh, un interruttore!")
        pass
    pass
```

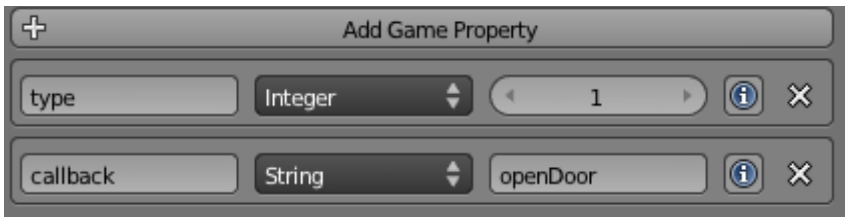
Se è un interruttore allora dobbiamo prima verificare la distanza:

```
def doLookAt(type, gameObject):
    if type == TYPE_SWITCH:
        doLookAtSwitch(gameObject)
        pass
    pass

def doLookAtSwitch(gameObject):
    distance = gameObject.getDistanceTo(player)
    if distance < 2:
        print("Oh, un interruttore vicino a me!")
        pass
    pass
```

A questo punto sappiamo che stiamo guarndando un interruttore e che quell'interruttore è vicino. Nota a margine: se l'avessimo metteremmo qui il codice per cambiare il puntatore da mirino a manina o quel che è. L'avremo, forse, quel dì in cui parleremo dell'interazione con l'HUD.
Vicini e guardanti, verifichiamo se siamo “prementi”:

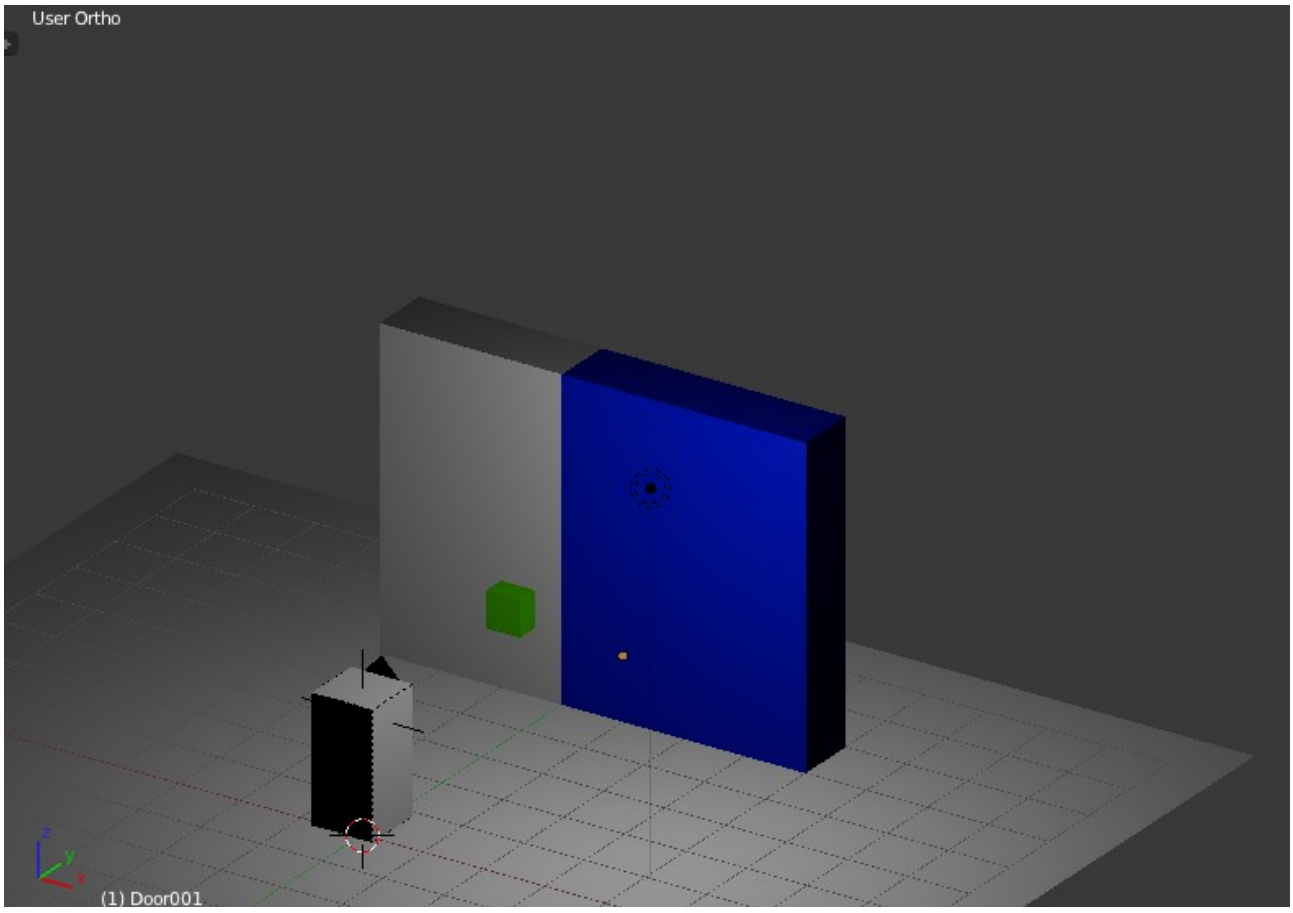
```
ACTION_KEY = bge.events.EKEY
def doLookAtSwitch(gameObject):
    distance = gameObject.getDistanceTo(player)
    inRange = distance < 2
    ePressed = fpsutils.isKeyDown(ACTION_KEY)
    if inRange and ePressed:
        print("Ti premo, oh vicino interruttore!")
        pass
    pass
```

Che fare ora? Dipende dall'interruttore. Diciamo che gli interruttori hanno una proprietà di nome callback che corrisponde ad una funzione del nostro main:




```
def doLookAtSwitch(gameObject):
    distance = gameObject.getDistanceTo(player)
    inRange = distance < 2
    ePressed = fpsutils.isKeyDown(ACTION_KEY)
    if inRange and ePressed:
        function = gameObject["callback"]
        globals()[function](gameObject)
        pass
    pass

def openDoor(gameObject):
    print("Adesso apro la porta...")
    pass
```

Aggiungiamo una porta alla scena:



Grafica già vista in Dead Space 2 naturalmente. Aggiungiamo alla porta un'animazione apri-chiudi. Qui vedetevela col grafico: io ho aggiunto un nodo vuoto Door001, genitore del cubo-porta, che mi fa da perno, l'animazione si chiama “open”, va dal frame 0 al frame 50. Le animzioni si possono

avviare con un'attuatore oppure col metodo

playAction

dei KX_GameObject – praticamente tutti gli oggetti gestiti dal game engine. Se io so che la porta di chiama Door001 e l'animazione della porta “open” e va dal frame 0 al 50 allora posso scrivere:

```
porta =bge.logic.getCurrentScene().objects[“Door001”]
porta.playAction(“open”, 0, 50)
```

per eseguirla. Diciamo per generalizzare un attimo che il mio interruttore ha tra le proprietà anche il nome del suo bersaglio:

Che si fa? Si scrive ad esempio:

```
def openDoor(gameObject):
    print("Adesso apro la porta...")
    targetName = gameObject["target"]
    scene = bge.logic.getCurrentScene()
    door = scene.objects[targetName]
    door.playAction("open", 0, 50)
    pass
```

Occhio qui alla precondizione: io so che l'animazione si chiama “open” e che va dal frame 0 al 50. Cosa capita? L'animazione parte ma la situazione è ancora grigia: openDoor è invocato ciclicamente fintantochè il pulsante E è premuto. Una volta che l'animazione è partita dovrei invece evitare questa ripetizione, altrimenti o non vedo la porta aprirsi perchè sto tenendo premuto il pulsante e l'animazione riparte da zero, o premo il pulsante quando l'animazione è a metà e la porta mi si richiude. Insomma, è un problema di stato. E qui possiamo pensare ad una valanga di soluzioni. Eccone una facile facile: una porta può trovarsi in quattro stati:

CLOSED
OPENED
CLOSING
OPENING

Lo stato predefinito è CLOSED. Se è OPENING o OPENED non ha senso aprirla. Se è CLOSED o CLOSING non ha senso chiuderla. Può essere chiusa se è OPENED e aperta se è CLOSED.

```
def openDoor(gameObject):
    print("Adesso apro la porta...")
    targetName = gameObject["target"]
    scene = bge.logic.getCurrentScene()
    door = scene.objects[targetName]
    status = door.get("status", "closed")
    if status == "closed":
```

```
        pass
    elif status == "closing":
        pass
    elif status == "opened":
        pass
    elif status == "opening":
        pass
    pass
```

Se è closed, avvio l'animazione e la imposto a opening:

```
def openDoor(gameObject):
    print("Adesso apro la porta...")
    targetName = gameObject["target"]
    scene = bge.logic.getCurrentScene()
    door = scene.objects[targetName]
    status = door.get("status", "closed")
    if status == "closed":
        status = "opening"
        door.playAction("open", 0, 50)
        pass
    elif status == "closing":
        pass
    elif status == "opened":
        pass
    elif status == "opening":
        pass
    door["status"] = status
    pass
```

E tanto mi basta per aprirla. Come faccio a capire quando è aperta? Che ha fatto la vecchia volpe calva nel main?

```
updatelist = [navigator]

def main(controller):
    for e in updatelist: e.update()
    lookAt()
    pass
```

Io posso infilare in quell'updatelist un “callback” che monitora lo stato dell'animazione e cambia quello della porta quando l'animazione si è conclusa:. Questa è la classe:

```
class DoorStatusUpdater:

    def __init__(self, door, newStatus, updateList):
        self.door = door
        self.newStatus = newStatus
        self.updateList = updateList
        pass

    def submit(self):
        self.updateList.append(self)

    def update(self):
        if(not self.door.isPlayingAction(0)):
            self.door["status"] = self.newStatus
            self.updateList.remove(self)
            pass
        pass
    pass
```

Quella classe dice: quando l'animazione corrente è terminata, cambia il valore della proprietà “status” di door in qualcos'altro ed eliminati dalla lista degli elementi “update”. E questo è l'uso:

```
def openDoor(gameObject):
    print("Adesso apro la porta...")
    targetName = gameObject["target"]
    scene = bge.logic.getCurrentScene()
    door = scene.objects[targetName]
    status = door.get("status", "closed")
    if status == "closed":
        status = "opening"
        door.playAction("open", 0, 50)
        DoorStatusUpdater(door, "opened", updatelist).submit()
        pass
    elif status == "closing":
        pass
    elif status == "opened":
        pass
    elif status == "opening":
        pass
    door["status"] = status
    pass
```

Qui “updatelist” è quella lista di cose che rispondono al messaggio “update” che il main esegue ciclicamente. Insomma, DoorStatusUpdater è un “poller”. Così se avessi un'azione “close” potrei dire:

```
elif status == “opened”:
    status=”closing”
    door.playAction(“close”, 0, 50)
    DoorStatusUpdater(door, “closed”, updatelist).submit()
    pass
```

e ottenere l'effetto contrario.

Pinzillacchere.