Un gioco 3d in java.

Quant'è complicato fare un gioco in 3D? Dipende dal gioco. Volendo fare Unreal4000 o Crysis 5 quasi certamente molto complicato. Ma se svolazziamo un pochino più in basso potremmo avere delle piacevoli sorprese.

Il gioco.

Noi voliamo così bassi da bruciarci le penne sotto ai tacchi: FPS, un tizio in un'arena vaga sparacchiando a dei bersagli, fissi o mobili ma certamente non “camminanti”: la mancanza del paso doble deriva dalla totale inettitudine del sottoscritto nel creare un modello che cammina, salta, canta o balla. Tecnicamente non c'è alcun impedimento nell'inserire nella scena, con le librerie che useremo, un bel mostrone che agita i tentacoli.

Librerie, strumenti, risorse.

Java 6 JDK (http://www.oracle.com/technetwork/java/javase/downloads/index.html)
JMonkeyEngine 3 (http://jmonkeyengine.org/)
Blender 2.47 (http://www.blender.org/)
Audacity (http://audacity.sourceforge.net/)
http://www.freesound.org
http://www.cgtextures.com/
http://espeak.sourceforge.net/
Netbeans (http://netbeans.org/)

SO: un qualsiasi linux, windows o osx.

Co fu.

JMonkeyEngine3 è un insieme di api per programmare applicazioni 3d, include il motore di rendering più un insieme di accessori, dai caricatori di modelli alla gestione della fisica, che rende la creazione di un gioco 3d piuttosto rapida. Esiste anche un IDE, derivato da netbeans, che permette di creare l'applicazione con disinvoltura ancora maggiore (JMonkeyPlatform) che vi invito a provare.

Blender ci serve per creare livelli e modelli. Usiamo la versione 2.47 perchè l'ultima 2.5 beta al momento disponibile non integra (apparentemente) gli script per esportare i modelli in formati diversi da collada. Il formato di importazione preferito di JMonkeyEngine è ogre, supporta il wavefront obj per i modelli statici.
Usiamo Audacity per manipolare i suoni scaricati da freesound, nel caso in cui non coincidano esattamente con quello che ci serve (ad esempio perchè non sono dei loop oppure perchè ci interessa solo un pezzo di un traccia).
Usiamo ESpeak per generare vocalizzazioni. Non che siano particolarmente naturali ma, guarda caso, il nostro tizio è muto e pilota un robot che parla con voce...robotica. Che cu.., eh?

I siti freesound e cgtextures contengono... suoni e texture, in abbondanza.

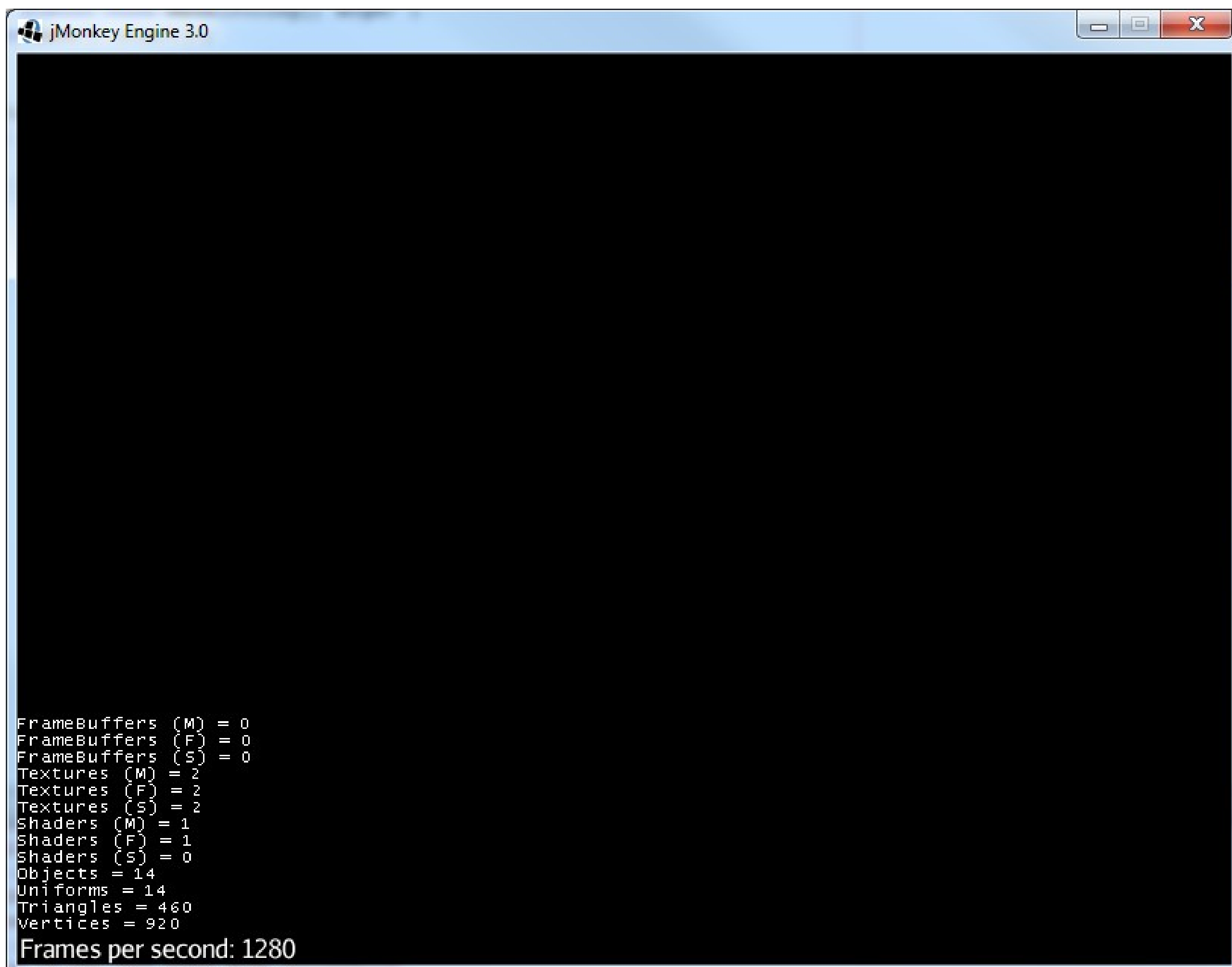
Pronti via.

Iniziamo con una bella finestrona JME3 vuota.

```
package game;

import com.jme3.app.SimpleApplication;

public class Game extends SimpleApplication {

    public static void main(String[] args) {
        new Game().start();
    }

    @Override
    public void simpleInitApp() {
    }
}
```

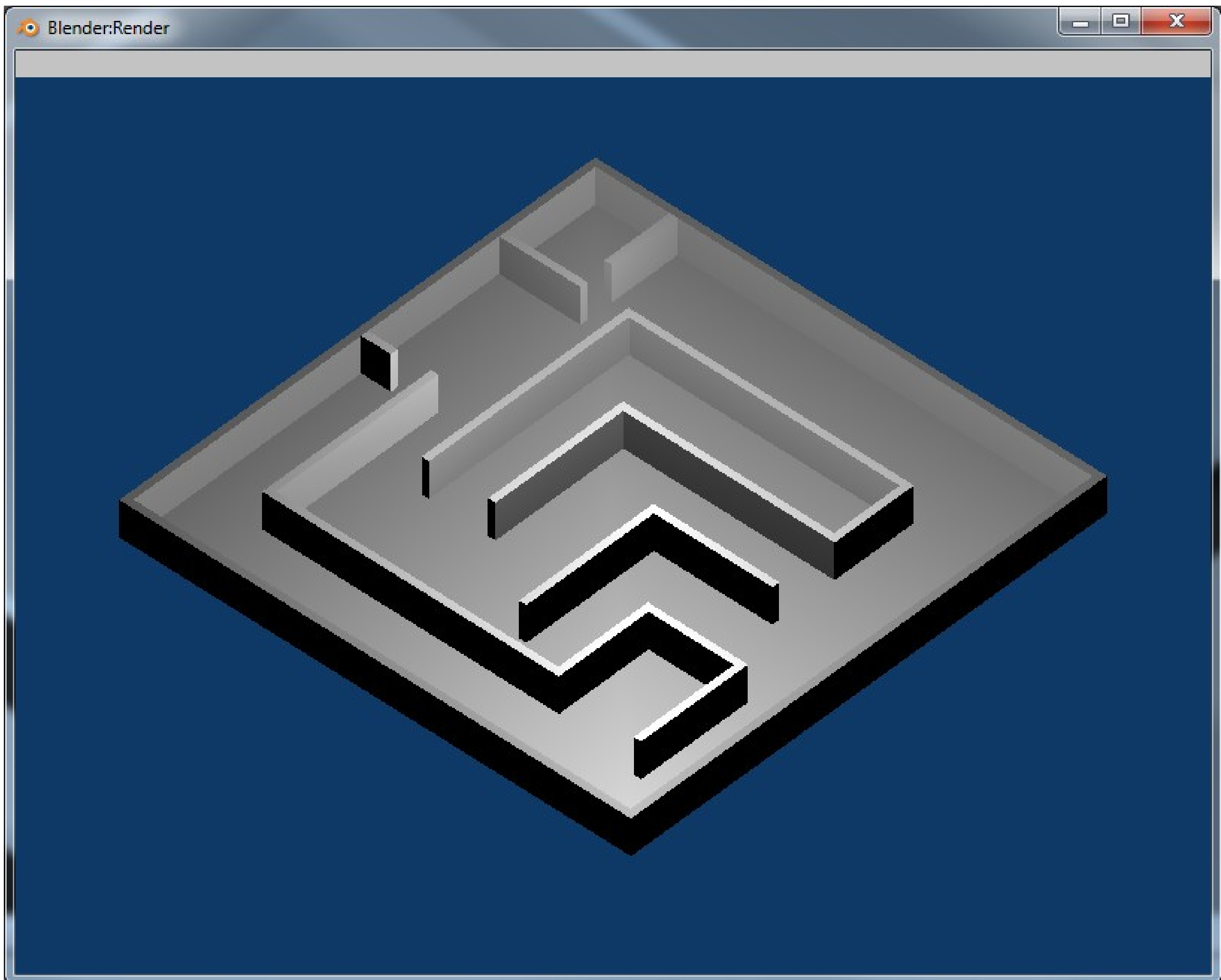
Eseguito, questo programma apre la finestra di configurazione del display:

Seguita dalla finestra contenente il programma:



Le scritte in basso a destra sono generate da due elementi utili a valutare cosa stia impegnando il motore di rendering e quali siano le performance attuali.

Ficchiamo qualcosa nel nostro gioco. Con blender, creiamo una geometria dimostrativa. In una decina di minuti produco il noto “livello del cassonetto”:



E' una sola geometria. Esporto questo pezzo d'antologia nel formato obj, in una cartella del mio progetto, diciamo la cartella “res”. Intendo per cartella del progetto un package nella cartella source di netbeans. Nella cartella res mi ritrovo dunque il file obj della geometria e il file mtl del material.

Carico il modello nel metodo simpleInitApp di Game. Il metodo, come intubile, è invocato una volta dal framework all'avvio del programma, quanto tutto e pronto per visualizzare i dati e subito prima che inizi il ciclo del motore di gioco.

Per caricare il livello, scrivo:

```
@Override
public void simpleInitApp() {
    Spatial livello = getAssetManager().loadModel("res/livello di prova.obj");
    getRootNode().attachChild(livello);
}
```

Il metodo getAssetManager resituisce l'istanza predefinita di AssetManager inizializzata da SimpleApplication. L'AssetManager è una specie di punto d'incontro delle varie cartelle, jar e ammennicoli impostabili nel programma e dalle quali possono essere caricate texture, modelli, suoni.

Il metodo loadModel carica i dati. Funziona per due ragioni: la prima è che il classpath è incluso tra i percorsi predefiniti dell'assetManager usato da SimpleApplication, la seconda è che esiste già un

caricatore predefinito per il formato obj in quello stesso AssetManager. Significa cioè che non posso dire tout-court, ad esempio:

getAssetManager().loadModel("qualsiasi percorso/file.qualsiasi formato");

Ci sono dei limiti. La seconda nuova linea inserisce il modello caricato nella radice della scena 3d:

getRootNode().attachChild(livello);

Il metodo getRootNode() restituisce l'istanza di Node che rappresenta la radice della scena 3d gestita da SimpleApplication, il metodo attachChild di Node inserisce un figlio in quel nodo. Il risultato dell'inserimento di un figlio nel nodo radice in questione è che il figlio sarà visualizzato sullo schermo. Il figlio può essere a sua volta un nodo eccetera eccetera, è il classico albero.

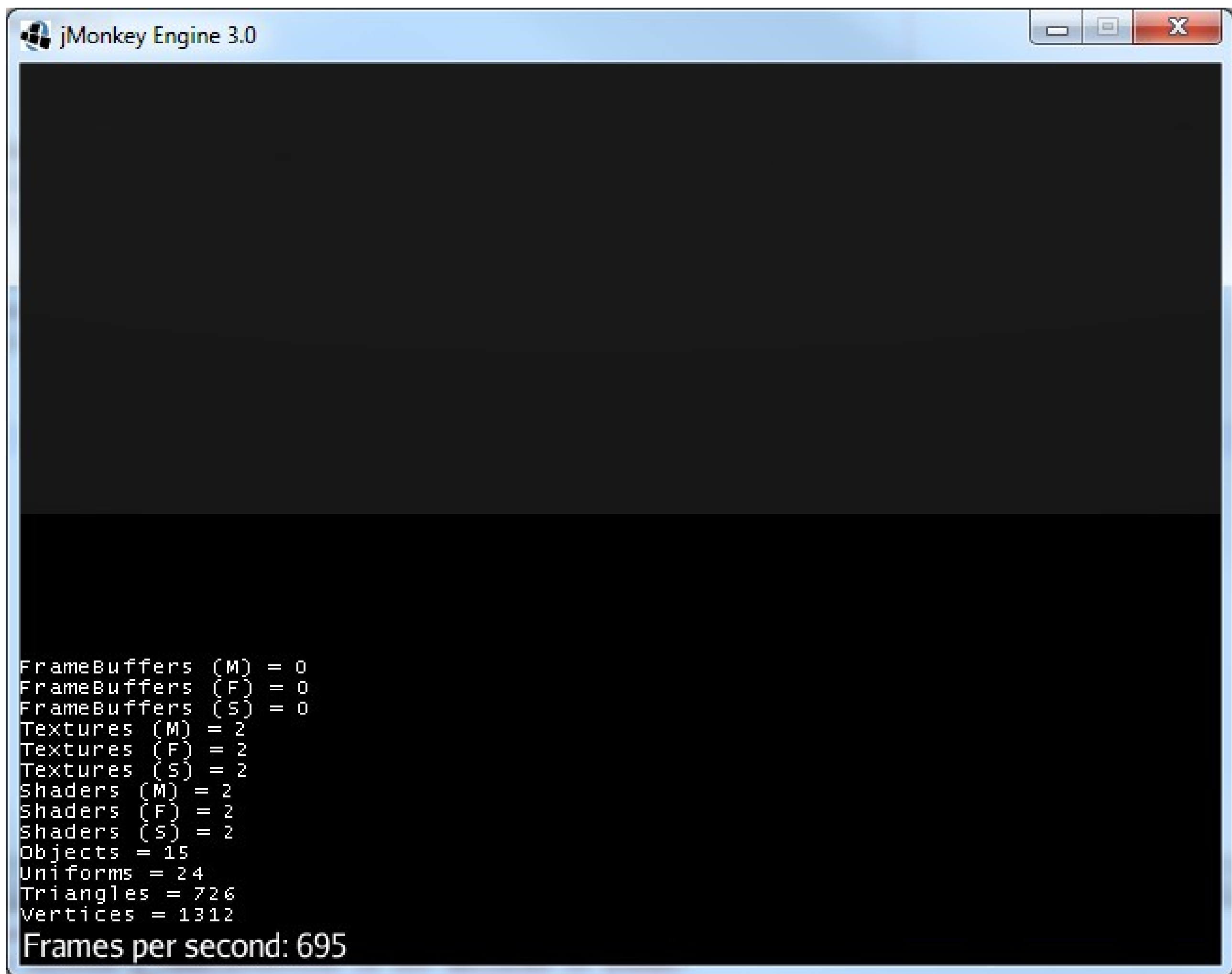
Se non è presente nel modello caricato, e nel nostro caso non lo è, allora occorre inserire almeno una lucetta nella scena, altrimenti non si vedrà un bel nulla.

```
@Override
public void simpleInitApp() {
    Spatial livello = getAssetManager().loadModel("res/livello di prova.obj");
    getRootNode().attachChild(livello);

    PointLight light = new PointLight();
    light.setPosition(new Vector3f(0, 100, 0));
    getRootNode().addLight(light);
}
```

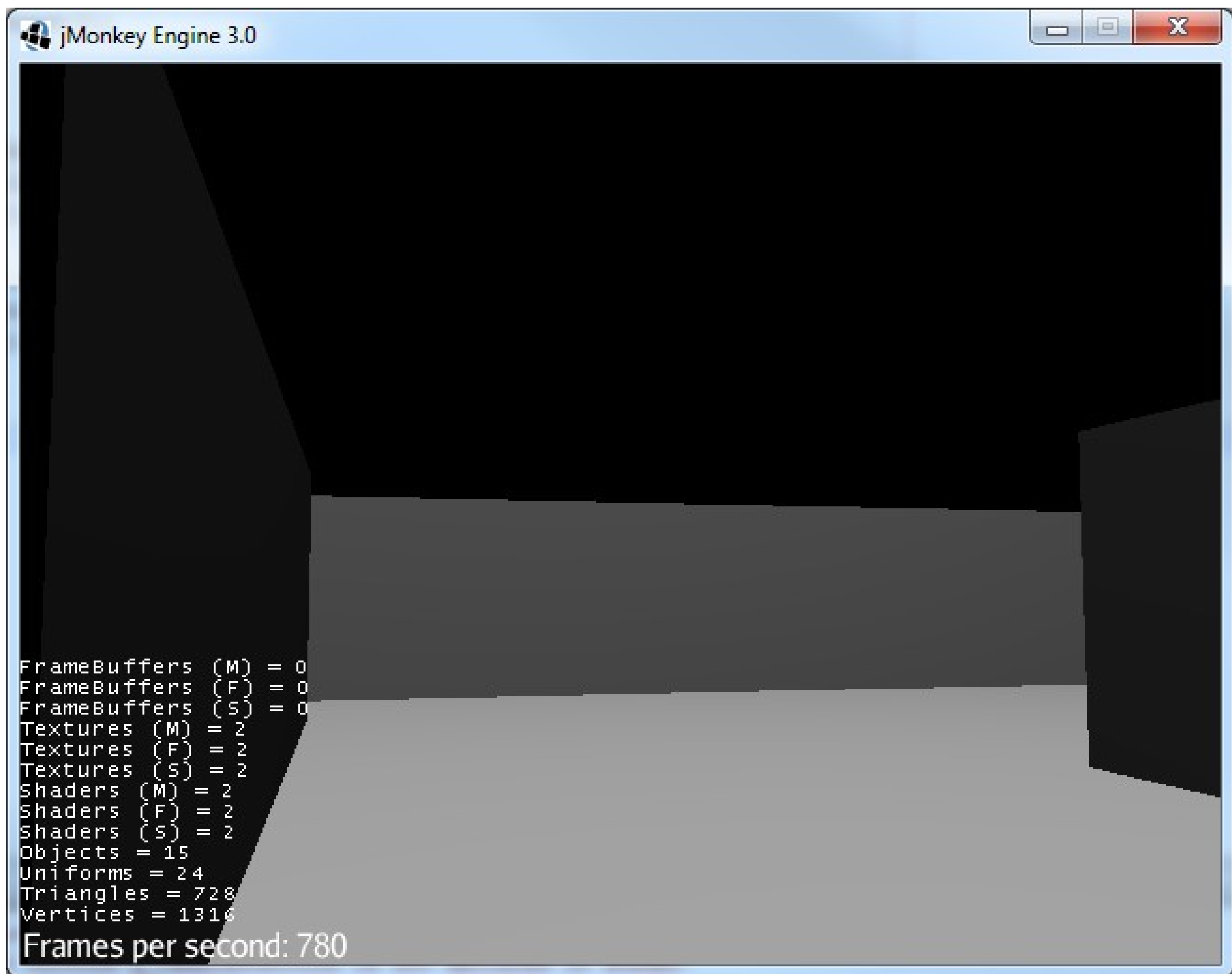
Le nuiove linee creano un punto luce o lo aggiungono alla radice della scena. Da notare che una luce illumina tutti gli elementi che appartengono al suo stesso ramo. Significa che se io dicessi:

```
Node scena = new Node("scena");
scena.attachChild(livello);
getRootNode().attachChild(scena);

Node luci = new Node("luci");
luci.addLight(luce);
getRootNode().attachChild(luci);
```

la luce non avrebbe effetto sulla scena, perchè l'illuminazione inizia da "luci" e prosegue lungo i figli di quel nodo mentre la scena appartiene ad un ramo fratello.
Ora abbiamo il modello nella scena. L'esecuzione genera quest'immagine:



La posizione iniziale della telecamera è a metà del pavimento, quindi l'avvio non è granchè. Si può muovere la vista usando i controlli predefiniti (WASD + il mouse). Spostandoci in giro si può vedere il livello da un altro punto di osservazione:



Fin qui niente di che. E neanche da qui in poi se è per questo ma arriva comunque una parte un pelo più interessante.

Muoversi nella scena.

JMonkeyEngine integra la gestione della fisica. Significa tante cose ma per un FPS ai minimi termini vuol dire muoversi nella scena senza attraversare i muri. Il bello è il come.

```
@Override
public void simpleInitApp() {
    Spatial livello = getAssetManager().loadModel("res/livello di prova.obj");
    getRootNode().attachChild(livello);

    PointLight light = new PointLight();
    light.setPosition(new Vector3f(0, 100, 0));
    getRootNode().addLight(light);

    //inizializzazione del simulatore fisico
    BulletAppState physics = new BulletAppState();
    getStateManager().attach(physics);

    //creazione della controparte fisica del modello
    float massaLivello = 0; //massa 0 produce un corpo statico
    RigidBodyControl rbLivello = new RigidBodyControl(massaLivello);
```

```
        livello.addControl(rbLivello);
        physics.getPhysicsSpace().add(rbLivello);
    }
```

Si crea questo BulletAppState, che gestisce e aggiorna il simulatore fisico, inserendolo in quello "state manager", che è una specie di lista di procedure invocate ad ogni ciclo del motore di gioco bla bla bla, non ce ne frega niente.

Una volta inizializzato questo simulatore, per ogni geometria (o per ogni nodo che contiene al suo interno delle geometri, cioè si può fare anche una volta per l'intera scena) si crea un "controller" (SpatialController) di tipo RigidBodyControl, lo si imposta come controllore per il nodo o la geometria di cui vogliamo produrre una controparte fisica e lo si aggiunge a quel getPhysicsSpace, affinchè il simulatore ne tenga conto durante il suo ciclo.

Da notare che se la massa usata per inizializzare il RigidBodyControl è zero allora il corpo è statico, nel senso che non reagisce alle forze dirette o indirette (cioè non cade per effetto della gravità e non si sposta per effetto degli impatti). In pratica gli altri corpi non statici possono picchiarci contro e quello resta fermo dov'è.

Compilando ed eseguendo non cambia nulla: c'è il muro ma manca la controparte fisica di chi non vuole attraversarlo.

Come si crea. I passaggi sono quattro. Si crea un volume che rappresenterà il giocatore, ad esempio una sfera. Si crea un CharacterControl usando quella sfera e lo si aggiunge al simulatore. Si crea un nodo da collegare a quel controllo e lo si inserisce nella scena. Si aggiorna periodicamente la posizione della telecamera facendole seguire quella del nodo.

```
package game;

import com.jme3.app.SimpleApplication;
import com.jme3.bullet.BulletAppState;
import com.jme3.bullet.collision.shapes.SphereCollisionShape;
import com.jme3.bullet.control.CharacterControl;
import com.jme3.bullet.control.RigidBodyControl;
import com.jme3.light.PointLight;
import com.jme3.math.Vector3f;
import com.jme3.scene.Node;
import com.jme3.scene.Spatial;

public class Game extends SimpleApplication {

    /* … omissis … */

    private final Node giocatore = new Node("camera");

    @Override
    public void simpleInitApp() {

        /* … omissis ...*/

        //creazione della controparte fisica della camera
        float raggio = 1,5f;
        SphereCollisionShape volumeCamera = new SphereCollisionShape(raggio);

        float altezzaPasso = 0.2f;
```

```
        CharacterControl ccCamera = new CharacterControl(volumeCamera, altezzaPasso);
        physics.getPhysicsSpace().add(ccCamera);

        giocatore.setLocalTranslation(0, 8, 0);
        giocatore.addControl(ccCamera);
        getRootNode().attachChild(giocatore);
    }

    @Override
    public void simpleUpdate(float tpf) {
        getCamera().setLocation(giocatore.getLocalTranslation());
    }
}
```

Nel metodo simpleUpdate, che il gioco invoca ciclicamente, assegnamo alla camera la posizione del nodo giocatore, a sua volta determinato dal simulatore fisico. Questa assegnazione fa sì che il movimento predefinito coi tasi WASD venga effettivamente annulato. Per muovere la camera dobbiamo applicare un impulso a ccCamera. Per un CharacterControl il movimento è controllato dal metodo setWalkDirection, se diciamo ccCamera.setWalkDirection(new Vector3f(1, 0, 0)), il simulatore farà muovere l'oggetto in direzione x positivo (salvo che abbia qualcosa tra i piedi, in quel caso tenterà di muoversi ma non ci riuscirà).
In pratica dobbiamo far sì che quando l'utente preme w, a, s, d sia invocato il metodo setWalkDirection di ccCamera, in modo che il simulatore applichi uno spostamento al nodo fantasma che rappresenta il giocatore col suo volume immaginario nell'universo fisico. Come si fa.
In jme3 la gestione dell'input può essere fatta usando un RawInputListener o creando delle associazioni tra un input ed un ascolatore di eventi. Nessuno dei due è particolarmente degno di nota, noi usiamo il secondo tipo per questo esempio, tanto poi dovremo scrivere un piccolo framework logico per gestire oltre all'input anche gli eventi logici che condurranno il gioco.
Sfruttiamo delle associazioni pulsante-etichetta già presenti in SimpleApplication e otteniamo quel che segue:

```
package game;

import com.jme3.app.SimpleApplication;
import com.jme3.bullet.BulletAppState;
import com.jme3.bullet.collision.shapes.SphereCollisionShape;
import com.jme3.bullet.control.CharacterControl;
import com.jme3.bullet.control.RigidBodyControl;
import com.jme3.input.controls.ActionListener;
import com.jme3.light.PointLight;
import com.jme3.math.Vector3f;
import com.jme3.scene.Node;
import com.jme3.scene.Spatial;

public class Game extends SimpleApplication {

    public static void main(String[] args) {
        new Game().start();
    }

    private final Node giocatore = new Node("camera");

    @Override
    public void simpleInitApp() {
        Spatial livello = getAssetManager().loadModel("res/livello di prova.obj");
        getRootNode().attachChild(livello);
```

```
    PointLight light = new PointLight();
    light.setPosition(new Vector3f(0, 100, 0));
    getRootNode().addLight(light);

    //inizializzazione del simulatore fisico
    BulletAppState physics = new BulletAppState();
    getStateManager().attach(physics);

    //creazione della controparte fisica del modello
    float massaLivello = 0; //massa 0 produce un corpo statico
    RigidBodyControl rbLivello = new RigidBodyControl(massaLivello);
    livello.addControl(rbLivello);
    physics.getPhysicsSpace().add(rbLivello);

    //creazione della controparte fisica della camera
    float raggio = 1.5f;
    SphereCollisionShape volumeCamera = new SphereCollisionShape(raggio);

    float altezzaPasso = 0.2f;
    ccCamera = new CharacterControl(volumeCamera, altezzaPasso);
    physics.getPhysicsSpace().add(ccCamera);

    giocatore.setLocalTranslation(0, 8, 0);//posizione iniziale del giocatore
    giocatore.addControl(ccCamera);
    getRootNode().attachChild(giocatore);

    /* onAction è eseguito da InputManager quando si verifica l'evento di
     basso livello associato ad una certa etichetta. L'actionlistener
     riceve lo stato del controllo e l'etichetta */
    ActionListener playerMoveListener = new ActionListener() {

        public void onAction(String name, boolean isPressed, float tpf) {
            if(name == "FLYCAM_StrafeLeft") {
                playerMotionVector.x = isPressed ? 1 : 0;
            } else if(name == "FLYCAM_StrafeRight") {
                playerMotionVector.x = isPressed ? -1 : 0;
            } else if(name == "FLYCAM_Forward") {
                playerMotionVector.z = isPressed ? 1 : 0;
            } else if(name == "FLYCAM_Backward") {
                playerMotionVector.z = isPressed ? -1 : 0;
            }
        }
    };
    getInputManager().addListener(playerMoveListener,
            "FLYCAM_StrafeLeft", //predefinito, associato al tasto A
            "FLYCAM_StrafeRight", //predefinito, associato al tasto D
            "FLYCAM_Forward", //predefinito, associato al tasto W
            "FLYCAM_Backward");//predefinito, associato al tasto S

    //questa associazioni sono predefinite in FlyByCamera, usata da
    //SimpleApplication per muovere la camera. Le rimuoviamo perchè non ci
    //servono. Lasciamo invece quelle che fanno "ruotare" la telecamera
    getInputManager().deleteMapping("FLYCAM_ZoomIn");
    getInputManager().deleteMapping("FLYCAM_ZoomOut");
    getInputManager().deleteMapping("FLYCAM_Rise");
    getInputManager().deleteMapping("FLYCAM_Lower");
}

private final Vector3f playerMotionVector = new Vector3f();//assoluto
```

```
    private final Vector3f vcache = new Vector3f();//usato per i conti intermedi
    private float playerSpeed = 0.25f;//velocità del giocatore
    private CharacterControl ccCamera;

    @Override
    public void simpleUpdate(float tpf) {
        vcache.set(playerMotionVector);
        getCamera().getRotation().multLocal(vcache).multLocal(playerSpeed);
        vcache.y = 0;//il giocatore non si può muovere lungo l'asse y
        ccCamera.setWalkDirection(vcache);
        getCamera().setLocation(giocatore.getLocalTranslation());
    }
}
```

Il meccanismo è questo: usiamo tre float, rappresentati da playerMotionVector, per stabilire la direzione del movimento del giocatore, in termini assoluti: destra, sinistra (x+ o x-), avanti o indietro (z+, z-). Cambiamo i valori di questo vettore quando l'utente preme o rilascia uno dei pulsanti associati alle etichette FLYCAM_Forward, FLYCAM_Backward, FLYCAM_StrafeLeft, FLYCAM_StrafeRight – cioè ai tasi w, s, d, a. Così l'utente preme A e playerMotionVector.x diventa 1, lo rilascia e lo stesso valore diventa 0, eccetera. Questo ci da la direzione come se la camera non avesse subito alcun cambiamento nella sua rotazione (cioè la direzione nello spazio di coordinate globale). Dopodichè la camera ha un suo orientamento, che l'utente cambia muovendo il mouse. Per tenere conto di questo orientamento diciamo:

```
    @Override
    public void simpleUpdate(float tpf) {
        vcache.set(playerMotionVector);
        getCamera().getRotation().multLocal(vcache)...eccetera
```

Cioè trasformiamo la direzione assoluta espressa nel vettore playermotionVector in direzione relativa all'orientamento della camera. Come dire che se "cammina" significa dire "muoviti in avanti" e chi deve camminare è girato verso la finestra allora "cammina" per lui significa "muoviti verso la finestra".
Il valore "vcache" ci serve per tenere separati il vettore assoluto da quello orientato. Il metodo getRotation di Camera (valore restituito da getCamera()) resituisce l'orientamento della camera, in forma di Quaternion. Il metodo multLocal di Quaterion applicato ad un vettore trasforma il vettore che riceve nella versione "ruotata" secondo l'orientamento espresso nel quaternione.
Il risultato del codice completo da ultimo proposto è che la nostra camera-giocatore è in grado di spostarsi all'interno del livello definito nel file wavefront/obj creato con blender, scivolando lungo le pareti, evitando di attraversare i muri e, se ce ne fossero, salendo le scale. Salire le scale significa che il motore fisico consente al volume fittizio del giocatore di superare dislivelli pari al valore altezzaPasso, da noi usato durante la costruzione dell'istanza ccCamera.

```
    @Override
    public void simpleInitApp() {
        ...
        float altezzaPasso = 0.2f;
        ccCamera = new CharacterControl(volumeCamera, altezzaPasso);
        physics.getPhysicsSpace().add(ccCamera);
        ...
```

Un po' di logica.

Scriviamo un piccolo framework logico da usare nel resto del programma. Il framework molto banalmente di eseguire dei compiti quando si verifcano dei fatti, ad esempio quando il giocatore entra nel volume di controllo di un power-up quel power-up è rimosso dal gioco ed il suo effetto acquisito. Oppure quando il giocatore spara a qualcosa, quel qualcosa è distrutto. Cose così. Ci sono un tot di framework del genere in giro (uno se non erro nella api di google) ma la faccenda è talmente corta che non serve andare a cercare chissà cosa. Il framework mima l'idea che nel gioco esistano delle condizioni al verificarsi delle quali sono attivate delle reazioni. Condizioni e reazioni possono essere combinate tra loro per generare condizioni che dipendono da più variabili o reazioni compose di più passaggi. Capita che il numero di condizioni e reazioni atomiche sia relativamente basso rispetto alla varietà di comportamenti che si possono ottenere combinandole. Servono cinque tasselli per comporre il puzzle (che metafora...).

Esiste un insieme di valori predefiniti usati dalle condizioni per determinare lo stato del gioco e dalle reazioni per alterarlo, chiamiamolo LogicEnvironment. Qui ci ficchiamo i riferimenti alla posizione della camera, allo stato dei controlli, alla linea temporale eccetera.

Esistono delle condizioni, convenzionalmente verificate quando una funzione astratta applicata al precedente insieme di valori restituisce true. Vale a dire che ogni condizioni è istanza di una classe che ha un metodo che restituisce true quando una certa operazione su un LogicEnvironment è true. Ad esempio una condizione che verifica se un tasto è premuto restituisce true quando dall'esame del LogicEnvironment risulti che quel tasto è stato premuto. Le condizioni possono essere combinate tramite un AND o un OR per generare condizioni complesse. Così se una certa azione dovesse capitare solo qualora l'utente abbia premuto il tasto x e un ipotetico valore delle statistiche sia maggiore di zero, la condizione sarà una cosa del tipo

Condizione c = new TastoPremuto(x).and(new ValoreMinoreDi(bla bla));

Esistono delle reazioni, definite come funzioni astratte applicate ad un LogicEnvironment, combinabili tra loro tramite un AND. Avendo le reazioni:

Reazione r0 = new MuoviInAvanti()
Reazione r1 = new Salta();

Dire:

Reazione r01 = r0.and(r1);

Significherà “muovi in avanti e salta”. Nulla di stratosferico.

Una condizione ed una reazione formano un evento attivabile che viene esaminato dopo essere stato aggiunto al LogicEnvironment. Cioè:

Condizione c = …
Reazione r = …
Evento e = new Evento(r, c);
logicEnv.add(e);

Significa che quando si verifica c sarà eseguito r.

Quinto pezzo, più importante di quanto si pensi, qualora una condizione o una reazione richieda un valore per operare, quel valore gli sarà fornito attraverso un wrapper mutabile (LogicParam<T>), cioè un puntatore al valore che dovrà effettivamente essere considerato. Significa che se ho una

condizione “tasto premuto” che verifica se un certo pulsante sia premuto, anziché crearla dicendo:

Condizione x = new TastoPremuto(KeyInput.KEY_W);

farò in modo di dover dire:

LogicParam<Integer> key = LogicParam<Integer>(KeyInput.KEY_W);
Condizione c = new TastoPremuto(key);

Il livello di indirezione introdotto dai parametri permette di creare una classe di reazioni, quelle che impostano il valore di un parametro arbitrario, che rende il sistema flessibile. Senza, in java, è tutto un fiorire di classi interne.

Codice

La condizione ha questa forma.

```
package jmex.logic;

/* Una condizione è qualcosa che può essere vero o falso, secondo una procedura
 astratta applicata ad un LogicEnvironment */
public abstract class LogicCondition {

   /* Verifica questa condizione. */
   public abstract boolean check(LogicEnvironment logics);

   /* Genera una nuova condizione che è vera se questa condizione e quella in
    argomento sono entrambe vere */
   public LogicCondition and(LogicCondition that) {
      return new And(this, that);
   }

   /* Genera una nuova condizione che è vera se questa condizione o quella in
    argomento sono vere */
   public LogicCondition or(LogicCondition that) {
      return new Or(this, that);
   }

   /* La condizione risultante dall'AND logico di due condizioni */
   private static class And extends LogicCondition {
      private final LogicCondition a;
      private final LogicCondition b;

      public And(LogicCondition a, LogicCondition b) {
         this.a = a;
         this.b = b;
      }

      @Override
      public boolean check(LogicEnvironment logics) {
         return a.check(logics) & b.check(logics);
      }
   }

   /* La condizione risultante dall'OR logico di due condizioni */
   private static class Or extends LogicCondition {
```

```
        private final LogicCondition a;
        private final LogicCondition b;

        public Or(LogicCondition a, LogicCondition b) {
            this.a = a;
            this.b = b;
        }

        @Override
        public boolean check(LogicEnvironment logics) {
            return a.check(logics) | b.check(logics);
        }
    }
}
```

Per creare una condizione si crea una sottoclasse di LogicCondition e si fornisce un'implementazione per il metodo check(LogicEnvironment).
La reazione è anche più corta:

```
package jmex.logic;

/* Una reazione è una qualche azione intrapresa a seguito del verificarsi di una
 condizione */
public abstract class LogicReaction {

    public abstract void act(LogicEnvironment logics);

    public LogicReaction and(LogicReaction that) {
        return new And(this, that);
    }

    /* La combinazione di due reazioni */
    private static class And extends LogicReaction {
        private final LogicReaction a, b;

        public And(LogicReaction a, LogicReaction b) {
            this.a = a;
            this.b = b;
        }

        public void act(LogicEnvironment logics) {
            a.act(logics);
            b.act(logics);
        }
    }
}
```

Le capsule per i parametri eventualmente necessari a condizioni e reazioni sono di questo tipo:

```
package jmex.logic;

public class LogicParam<T> {

    public static <T> LogicParam<T> wrap(T value) {
        return new LogicParam<T>(value);
    }
```

```
    public T value;

    public LogicParam(T value) {
        this.value = value;
    }
}
```

Questo permette di cambiare le carte in tavola a parità di condizioni e reazioni.
La coppia condizione-reazioni è rappresentata da un LogicTrigger:

```
package jmex.logic;

public class LogicTrigger {
    private final LogicParam<LogicCondition> condition;
    private final LogicParam<LogicReaction> reaction;

    public LogicTrigger(LogicParam<LogicCondition> condition, LogicParam<LogicReaction> reaction) {
        this.condition = condition;
        this.reaction = reaction;
    }

    public void test(LogicEnvironment logics) {
        if(condition.value != null && condition.value.check(logics)) {
            if(reaction.value != null) reaction.value.act(logics);
        }
    }
}
```

Infine arriva il LogicEnvironment, che è la classe più corposa. In verità non fa molto salvo tradurre l'input di basso livello in qualcosa di più commestibile e mantenere un paio di riferimenti generalmente disponibili in una SimpleApplication (la linea temporale a il renderer dei suoni).

```
package jmex.logic;

import com.jme3.audio.AudioRenderer;
import com.jme3.input.RawInputListener;
import com.jme3.input.event.JoyAxisEvent;
import com.jme3.input.event.JoyButtonEvent;
import com.jme3.input.event.KeyInputEvent;
import com.jme3.input.event.MouseButtonEvent;
import com.jme3.input.event.MouseMotionEvent;
import java.util.ArrayList;
import java.util.EnumMap;
import java.util.HashMap;
import java.util.Map;

public class LogicEnvironment implements RawInputListener {

    private double timeline = 0;
    private int mouseMotionDx, mouseMotionDy, mouseMotionDw;
    private final Map<Integer, Boolean> keyStates = new HashMap<Integer, Boolean>();
    private final Map<MouseState.MouseButton, MouseState> mouseButtonStates =
            new EnumMap<MouseState.MouseButton, MouseState>(MouseState.MouseButton.class);
    private final ArrayList<LogicTrigger> triggers = new ArrayList<LogicTrigger>();
    private AudioRenderer audioRenderer;

    /**
     * Instance initializer
     */
    public LogicEnvironment() {
```

```
}

public AudioRenderer getAudioRenderer() {
    return audioRenderer;
}

public void setAudioRenderer(AudioRenderer audioRenderer) {
    this.audioRenderer = audioRenderer;
}

public void addTrigger(LogicTrigger t) {
    triggers.add(t);
}

public LogicTrigger addTrigger(LogicCondition c, LogicReaction r) {
    LogicTrigger trigger = new LogicTrigger(
            new LogicParam<LogicCondition>(c),
            new LogicParam<LogicReaction>(r));
    triggers.add(trigger);
    return trigger;
}

public void beginInput() {
}

public void endInput() {
}

public void onJoyAxisEvent(JoyAxisEvent e) {
}

public void onJoyButtonEvent(JoyButtonEvent e) {
}

public void onMouseMotionEvent(MouseMotionEvent e) {
    updateMouseMotionState(e);
}

public void onMouseButtonEvent(MouseButtonEvent e) {
    MouseState mouseState = new MouseState(e.getButtonIndex(), e.isPressed(), e.getX(), e.getY());
    mouseButtonStates.put(mouseState.button, mouseState);
}

public void onKeyEvent(KeyInputEvent e) {
    keyStates.put(e.getKeyCode(), e.isPressed());
}

public MouseState getMouseButtonState(MouseState.MouseButton button) {
    return mouseButtonStates.get(button);
}

public boolean getKeyState(int keyCode) {
    Boolean state = keyStates.get(keyCode);
    return state == null ? false : state;
}

public void update(float tpf) {
    timeline += tpf;
    for (int i= 0; i < triggers.size(); i++) {
```

```
        triggers.get(i).test(this);
      }
      clearMouseMotionState();
    }

    public int getMouseMotionDw() {
      return mouseMotionDw;
    }

    public int getMouseMotionDx() {
      return mouseMotionDx;
    }

    public int getMouseMotionDy() {
      return mouseMotionDy;
    }

    public double getTimeline() {
      return timeline;
    }

    private void updateMouseMotionState(MouseMotionEvent e) {
      this.mouseMotionDx = e.getDX();
      this.mouseMotionDy = e.getDY();
      this.mouseMotionDw = e.getDeltaWheel();
    }

    private void clearMouseMotionState() {
      mouseMotionDx = mouseMotionDx = mouseMotionDw = 0;
    }

    public void removeTrigger(LogicTrigger value) {
      triggers.remove(value);
    }

    public void install(SimpleApplication app) {
      app.getInputManager().addRawInputListener(this);
      setAudioRenderer(app.getAudioRenderer());
    }
}
```

I metodi in rosso sono quelli usati rilevanti, gli altri sono serventi. Il sistema funziona così:

```
public class Game extends SimpleApplication {

        private final LogicEnvironment env = new LogicEnvironment();

        @Override
        public void simpleInitApp() {
                logics.install(this);

                ...ficca in env un po' di condizioni e reazioni
        }

        public void simpleUpdate(float tpf) {
                env.update(tpf);
        }
}
```

L'invocazione env.update causa l'analisi delle condizioni e l'eventuale esecuzione delle reazioni collegate. Naturalmente servono delle condizioni e delle reazioni precotte altrimenti tanto varrebbe

scrivere tutto nel simpleUpdate. Vediamo l'input reinterpretato. Ci serve una condizione che si attivi quando è premuto un tasto ed una che si attivi quando lo stesso è rilasciato. La condizione che verifica la pressione potrebbe essere:

```
package jmex.logic.basic;

import jmex.logic.LogicCondition;
import jmex.logic.LogicParam;
import jmex.logic.LogicEnvironment;

public class LCKeyPressed extends LogicCondition {

    private final LogicParam<Integer> key;
    private boolean wasDown;

    public LCKeyPressed(LogicParam<Integer> key) {
        this.key = key;
    }

    @Override
    public boolean check(LogicEnvironment logics) {
        return key.value != null && isPressed(logics);
    }

    private boolean isPressed(LogicEnvironment logics) {
        boolean isDown = logics.getKeyState(key.value);
        if(isDown) {
            return !wasDown ? wasDown = true : false;
        } else {
            return wasDown = false;
        }
    }
}
```

Mentre la condizione che verifica il rilascio sarebbe:

```
package jmex.logic.basic;

import jmex.logic.LogicCondition;
import jmex.logic.LogicParam;
import jmex.logic.LogicEnvironment;

public class LCKeyReleased extends LogicCondition {

    private final LogicParam<Integer> key;
    private boolean wasDown;

    public LCKeyReleased(LogicParam<Integer> key) {
        this.key = key;
    }

    @Override
    public boolean check(LogicEnvironment logics) {
        return key.value != null && isReleased(logics);
    }

    private boolean isReleased(LogicEnvironment logics) {
        boolean isDown = logics.getKeyState(key.value);
        if(!isDown && wasDown) {
            return !(wasDown = false);
        } else {
```

```
            return isDown ? !(wasDown = true) : false;
        }
    }
}
```

Per gestire il movimento della camera assegnato un certo valore ad una variabile che rappresenta la direzione del movimento. Ci serve una reazione che imposta un valore, banalmente:

```
aaaapackage jmex.logic.basic;

import jmex.logic.LogicParam;
import jmex.logic.LogicReaction;
import jmex.logic.LogicEnvironment;

public class LRSetValue<T> extends LogicReaction {

    private final LogicParam<T> variable;
    private final LogicParam<T> newValue;

    public LRSetValue(LogicParam<T> variable, LogicParam<T> newValue) {
        this.variable = variable;
        this.newValue = newValue;
    }

    @Override
    public void act(LogicEnvironment logics) {
        variable.value = newValue.value;
    }
}
```

Dopodichè avremo bisogno di applicare periodicamente la direzione di movimento al nodo fisico del giocatore e la posizione del nodo fisico alla camera. Predisponiamo una condizione “sempre”:

```
package jmex.logic.basic;

import jmex.logic.LogicCondition;
import jmex.logic.LogicEnvironment;

public class LCAlways extends LogicCondition {

    @Override
    public boolean check(LogicEnvironment logics) {
        return true;
    }
}
```

Il resto è traduzione. Quando l'utente preme W impostiamo a 1 la direzione Z del movimento del giocatore:

```
final LogicParam<Integer> keyUp = LogicParam.wrap(KeyInput.KEY_W);
final LogicParam<Float> playerDirZ = LogicParam.wrap(0f);
logics.addTrigger(
        new LCKeyPressed(keyUp),
        new LRSetValue<Float>(playerDirZ, LogicParam.wrap(1f)));
```

Quando l'utente preme S, impostiamo a -1 la stessa direzione:

```
final LogicParam<Integer> keyDown = LogicParam.wrap(KeyInput.KEY_S);
logics.addTrigger(
        new LCKeyPressed(keyDown),
        new LRSetValue<Float>(playerDirZ, LogicParam.wrap(-1f)));
```

Quando l'utente rilascia i tasti W o S, impostiamo a zero quel valore:

```
logics.addTrigger(
        new LCKeyReleased(keyUp).or(new LCKeyReleased(keyDown)),
```

```
        new LRSetValue<Float>(playerDirZ, LogicParam.wrap(0f)));
```

La pressione dei tasti A e D è collegata all'impostazione di una variabile dirX. Dopodichè non resta che introdurre una condizione che esegue sempre la sua reazione e inserire nella reazione i conti che prima facevamo nel metodo simpleUpdate. Alla fine della fiera, il codice diventa:

```
package game;

import com.jme3.app.SimpleApplication;
import com.jme3.bullet.BulletAppState;
import com.jme3.bullet.collision.shapes.SphereCollisionShape;
import com.jme3.bullet.control.CharacterControl;
import com.jme3.bullet.control.RigidBodyControl;
import com.jme3.input.KeyInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.light.PointLight;
import com.jme3.math.Quaternion;
import com.jme3.math.Vector3f;
import com.jme3.renderer.Camera;
import com.jme3.scene.Node;
import com.jme3.scene.Spatial;
import jmex.logic.LogicEnvironment;
import jmex.logic.LogicParam;
import jmex.logic.LogicReaction;
import jmex.logic.basic.LCAlways;
import jmex.logic.basic.LCKeyPressed;
import jmex.logic.basic.LCKeyReleased;
import jmex.logic.basic.LRSetValue;

public class Game extends SimpleApplication {

    public static void main(String[] args) {
        new Game().start();
    }

    private final Node giocatore = new Node("camera");
    private final LogicEnvironment logics = new LogicEnvironment();

    @Override
    public void simpleInitApp() {
        Spatial livello = getAssetManager().loadModel("res/livello di prova.obj");
        getRootNode().attachChild(livello);

        PointLight light = new PointLight();
        light.setPosition(new Vector3f(0, 100, 0));
        getRootNode().addLight(light);

        //inizializzazione del simulatore fisico
        BulletAppState physics = new BulletAppState();
        getStateManager().attach(physics);

        //creazione della controparte fisica del modello
        float massaLivello = 0; //massa 0 produce un corpo statico
        RigidBodyControl rbLivello = new RigidBodyControl(massaLivello);
        livello.addControl(rbLivello);
        physics.getPhysicsSpace().add(rbLivello);

        //creazione della controparte fisica della camera
        float raggio = 1.5f;
        SphereCollisionShape volumeCamera = new SphereCollisionShape(raggio);
```

```
float altezzaPasso = 0.2f;
final CharacterControl ccCamera = new CharacterControl(volumeCamera, altezzaPasso);
physics.getPhysicsSpace().add(ccCamera);

giocatore.setLocalTranslation(0, 8, 0);//posizione iniziale del giocatore
giocatore.addControl(ccCamera);
getRootNode().attachChild(giocatore);

getInputManager().deleteMapping("FLYCAM_ZoomIn");
getInputManager().deleteMapping("FLYCAM_ZoomOut");
getInputManager().deleteMapping("FLYCAM_Rise");
getInputManager().deleteMapping("FLYCAM_Lower");

//logics
logics.install(this);

final LogicParam<Integer> keyUp = LogicParam.wrap(KeyInput.KEY_W);
final LogicParam<Integer> keyDown = LogicParam.wrap(KeyInput.KEY_S);
final LogicParam<Integer> keyLeft = LogicParam.wrap(KeyInput.KEY_A);
final LogicParam<Integer> keyRight = LogicParam.wrap(KeyInput.KEY_D);

final LogicParam<Float> playerDirX = LogicParam.wrap(0f);
final LogicParam<Float> playerDirY = LogicParam.wrap(0f);
final LogicParam<Float> playerDirZ = LogicParam.wrap(0f);

final LogicParam<Float> playerSpeed = LogicParam.wrap(0.25f);

logics.addTrigger(
        new LCKeyPressed(keyUp),
        new LRSetValue<Float>(playerDirZ, LogicParam.wrap(1f)));
logics.addTrigger(
        new LCKeyPressed(keyDown),
        new LRSetValue<Float>(playerDirZ, LogicParam.wrap(-1f)));
logics.addTrigger(
        new LCKeyPressed(keyLeft),
        new LRSetValue<Float>(playerDirX, LogicParam.wrap(1f)));
logics.addTrigger(
        new LCKeyPressed(keyRight),
        new LRSetValue<Float>(playerDirX, LogicParam.wrap(-1f)));
logics.addTrigger(
        new LCKeyReleased(keyUp).or(new LCKeyReleased(keyDown)),
        new LRSetValue<Float>(playerDirZ, LogicParam.wrap(0f)));
logics.addTrigger(
        new LCKeyReleased(keyLeft).or(new LCKeyReleased(keyRight)),
        new LRSetValue<Float>(playerDirX, LogicParam.wrap(0f)));
logics.addTrigger(new LCAlways(), new LogicReaction() {

    private final Vector3f cache = new Vector3f();

    @Override
    public void act(LogicEnvironment logics) {
        Camera camera = Game.this.getCamera();
        Quaternion cameraRotation = camera.getRotation();
        cache.set(playerDirX.value, playerDirY.value, playerDirZ.value);
        cameraRotation.multLocal(cache).multLocal(playerSpeed.value);
        cache.y = 0;
        ccCamera.setWalkDirection(cache);
        cache.set(giocatore.getLocalTranslation());
```

```
            cache.y += 4f;//sposta la telecamera un po' più in alto (altrimenti ci guardiamo i tacchi)
            camera.setLocation(cache);
         }
      });
   }

   @Override
   public void simpleUpdate(float tpf) {
      logics.update(tpf);
   }
}
```

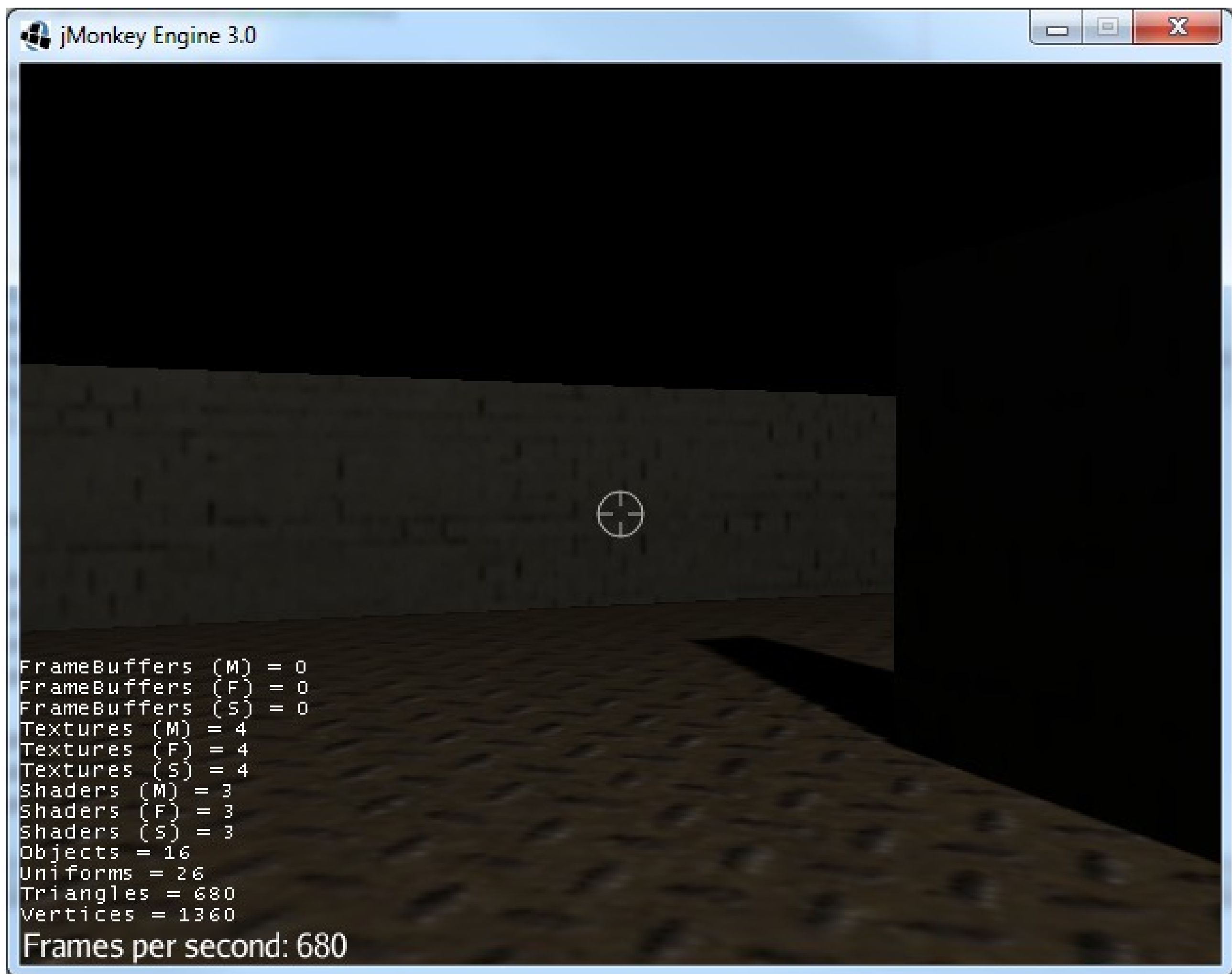
Mirino e ammennicoli.

Le API di JMonkeyEngine3 includono un gestore di ui, niftygui, di cui non sono un grandissimo fan quindi vi beccate l'alternativa. Una SimpleApplication ha un nodo, guiNode, il cui contenuto è proiettato in prospettiva ortogonale di fronte a tutto il resto. In pratica è come se ci fosse un grosso pannello trasparente di fronte allo schermo: tutto ciò che attacchiamo a guiNode diventa un'immagine piatta. La proiezione ortogonale "cambia" il sistema di coordinate nel senso che da coordinate relative all'universo 3d si passa ad un sistema di coordinate 2D in cui l'origine è l'angolo inferiore dello schermo e l'unità di riferimento è il punto del display (il pixel per intenderci). Se creo un cubo di 1x1x1 e lo piazzo nel rootNode, a schermo vedo un cubo le cui dimensioni variano a seconda della posizione della camera, se lo attacco al guiNode vedo un quadrato fisso nell'angolo inferiore sinistro dello schermo.
Supponiamo di avere quest'immagine nella cartella res del progetto:

Volendo usarla come mirino, nota la dimensione (32x32), si può dire, in simpleInitApp:

```
      //ui
      Picture mirino = new Picture("mirino");
      mirino.setWidth(32);
      mirino.setHeight(32);
      mirino.setTexture(assetManager, (Texture2D) assetManager.loadTexture("res/mirino.png"), true);
      mirino.setPosition(getCamera().getWidth() / 2 - 16, getCamera().getHeight() / 2 - 16);
      getGuiNode().attachChild(mirino);
```

E salta fuori:



En passant, per rimuovere le scritte in basso a sinistra si può scrivere, sempre nel simpleInitApp:

```
setDisplayFps(false);
setDisplayStatView(false);
statsView.removeFromParent();
fpsText.removeFromParent();
```

Un'utilità generale è l'uso di un Graphics2D per disegnare su una texture, applicata ad un Picture, il che permette di disegnare su un elemento 3D (nel caso della gui un 3D in proiezione ortogonale) usando un Graphics2D. Si passa da BufferedImage per fare prima:

```
package jmex.ui;

import com.jme3.texture.Image;
import com.jme3.texture.Texture2D;
import com.jme3.util.BufferUtils;
import java.awt.AlphaComposite;
import java.awt.Composite;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.nio.ByteBuffer;

public class PaintableTexture extends Texture2D {

    public static PaintableTexture newInstance(int w, int h) {
        final BufferedImage image = new BufferedImage(w, h, BufferedImage.TYPE_INT_ARGB);
```

```
        final ByteBuffer buffer = BufferUtils.createByteBuffer(w * h * 4);
        final Image textureImage = new Image(Image.Format.RGBA8, w, h, buffer);
        return new PaintableTexture(textureImage, image, buffer);
    }
    private final BufferedImage bufferedImage;
    private final Graphics2D graphics;
    private final ByteBuffer buffer;
    private final Image textureImage;

    protected PaintableTexture(Image image, BufferedImage awtImage, ByteBuffer buffer) {
        super(image);
        this.bufferedImage = awtImage;
        this.graphics = this.bufferedImage.createGraphics();
        this.buffer = buffer;
        this.textureImage = image;
    }

    public Graphics2D getGraphics() {
        return graphics;
    }

    public void clear() {
        Composite composite = graphics.getComposite();
        graphics.setComposite(AlphaComposite.Clear);
        graphics.setColor(new java.awt.Color(0, 0, 0, 0));
        graphics.fillRect(0, 0, getWidth(), getHeight());
        graphics.setComposite(composite);
    }

    public BufferedImage getBufferedImage() {
        return bufferedImage;
    }

    public int getWidth() {
        return getBufferedImage().getWidth();
    }

    public int getHeight() {
        return getBufferedImage().getHeight();
    }

    public void update() {
        final int w = getWidth();
        final int h = getHeight();
        final int pix = w * h;
        final BufferedImage img = getBufferedImage();
        buffer.clear();
        for (int i = 0; i < pix; i++) {
            int x = i % w;
            int y = i / w;
            int argb = img.getRGB(x, y);
            buffer.put((byte) ((argb >> 16) & 0xff));//r
            buffer.put((byte) ((argb >> 8) & 0xff));//g
            buffer.put((byte) ((argb) & 0xff));//b
            buffer.put((byte) ((argb >> 24) & 0xff));//a

        }
        buffer.flip();
        textureImage.setUpdateNeeded();
```

```
    }
}
```

Quella è la texture disegnabile, questo che segue è il picture che la usa:

```
package jmex.ui;

import com.jme3.asset.AssetManager;
import com.jme3.ui.Picture;
import java.awt.Graphics2D;

public class PaintablePicture extends Picture {

    public static PaintablePicture newInstance(AssetManager am, int w, int h) {
        return new PaintablePicture(am, PaintableTexture.newInstance(w, h));
    }

    private final PaintableTexture texture;

    public PaintablePicture(AssetManager am, PaintableTexture texture) {
        super("paintable picture", true);
        setWidth(texture.getBufferedImage().getWidth());
        setHeight(texture.getBufferedImage().getHeight());
        setTexture(am, texture, true);
        this.texture = texture;
    }

    public PaintableTexture getPaintableTexture() {
        return texture;
    }

    public Graphics2D getGraphics() {
        return texture.getGraphics();
    }

    public void flush() {
        texture.update();
    }

    public void clear() {
        texture.clear();
    }
}
```

A questo punto si può disegnare un po' di tutto. Ad esempio scrivendo:

```
        //prova disegno con graphics
        PaintablePicture label = PaintablePicture.newInstance(assetManager, 200, 50);
        label.setPosition(0, getCamera().getHeight() - 50);
        label.getGraphics().setPaint(java.awt.Color.PINK);
        label.getGraphics().setFont(new java.awt.Font("Serif", java.awt.Font.BOLD, 25));
        label.getGraphics().drawString("LIVELLO 001", 0, 25);
        label.flush();
        getGuiNode().attachChild(label);
```

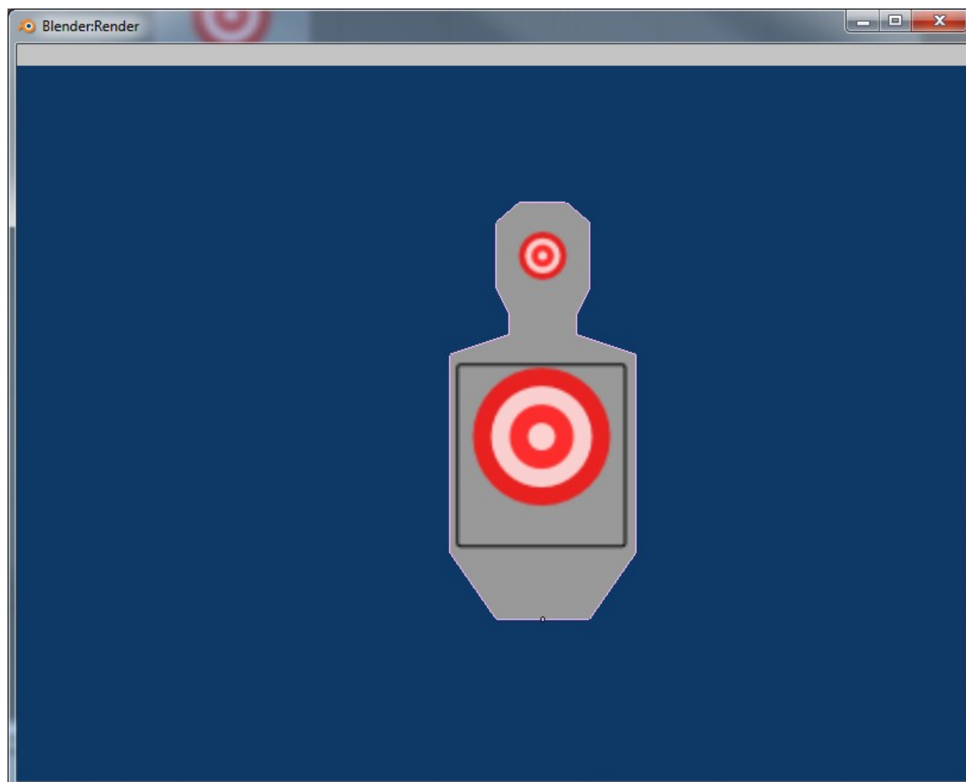
Otteniamo l'immagine che segue:

Occhio che il “flush” richiede una scansione del buffer pixel per pixel, il che lo rende un fps killer nel caso in cui lo si voglia usare per l'aggiornamento costante di immagini relativamente grosse. Tuttavia per piccoli elementi della gui (o piccole texture animate in 3D) è più che idoneo.

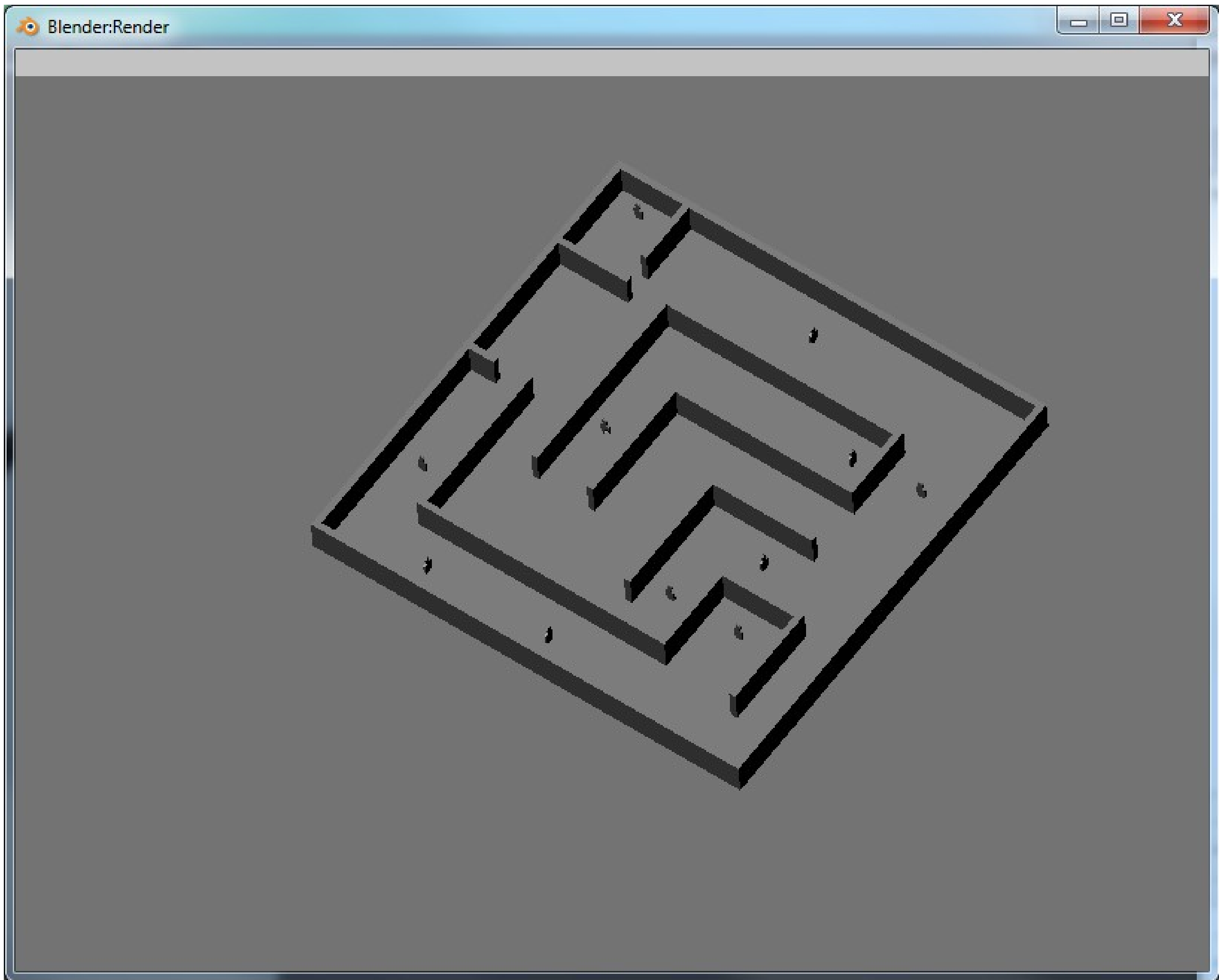
Spariamo a qualcosa.

Apriamo blender e il livello su cui stiamo lavorando.

Creiamo un bersaglio.



Alta scuola di grafica 3D. Pigliamo quest'affare e lo copiamo qua e la nel livello.



In blender, chiamiamo l'oggetto che rappresenta pareti e pavimento “room”, e i bersagli “target”. Salviamo il livello come file blender. Naturalmente va bene qualsiasi nome, purché serva all'identificazione.
Quello che facciamo adesso è sfruttura la struttura della scena per andare a “beccare” il tipo dei diversi elementi (due: la stanza che non fa niente e i bersagli a cui si spara).
Un piccolo problema deriva dal formato del file – il supporto di blender al formato ogre è quantomeno macchinoso, il formato obj supporta i gruppi ma sembra che il caricatore di jme3 non lo faccia o lo faccia secondo una sua logica. In linea generale è possibile usare JMonkeyPlatform per costruire la propria scena 3D combinando oggetti salvati singolarmente. C'è anche un caricatore per il formato .blend. Io carico direttamente i file di blender usando un importatore fai da te.
Caricato il modello, il primo passo è individuare gli elementi statici e generare la controparte fisica:

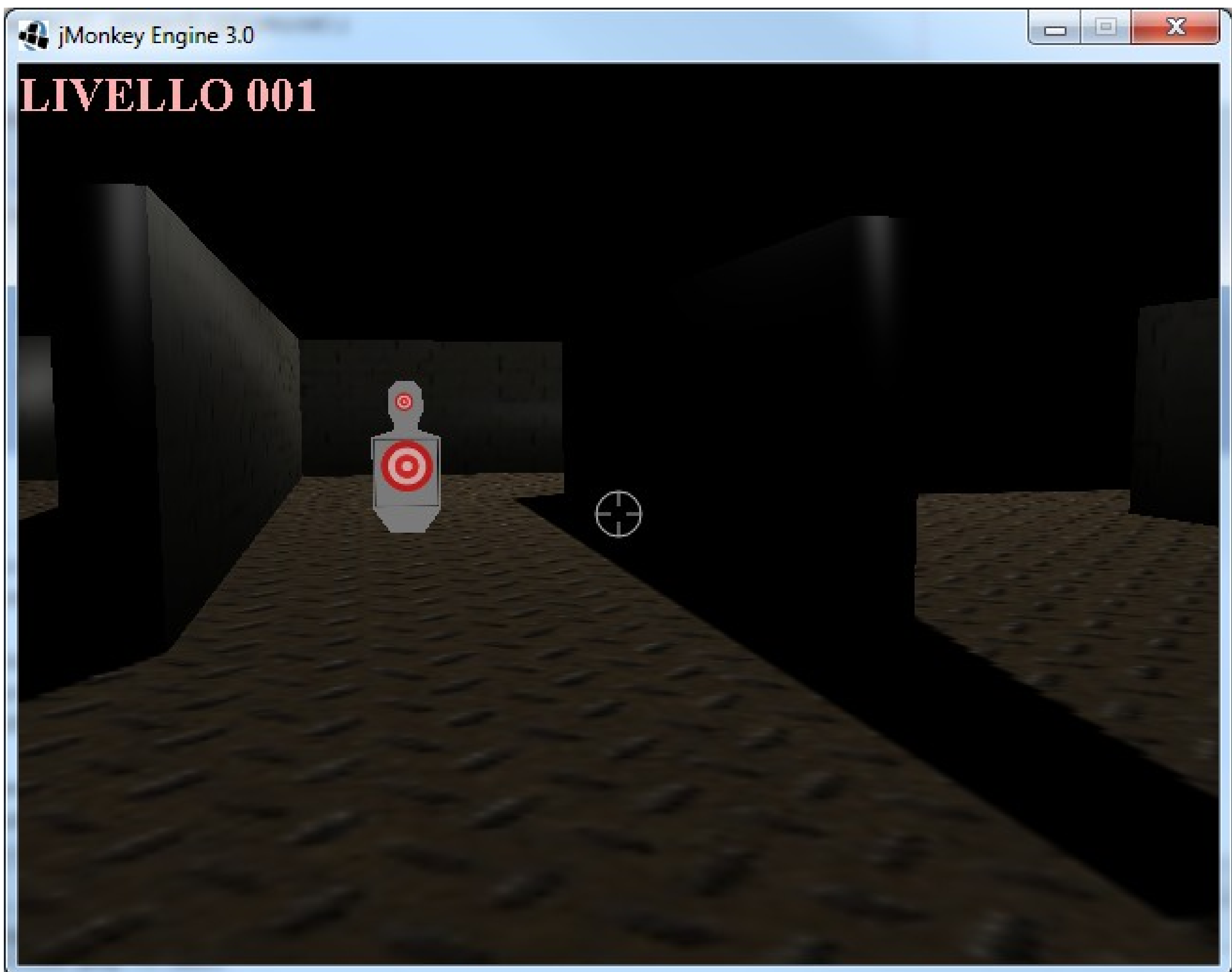
```
Spatial livello = getAssetManager().loadModel("res/livello di prova.blend");
getRootNode().attachChild(livello);
```

Noto che la geometria della stanza si chiama “room”, posso andare a pescarla per creare il controllo fisico, che mi permetterà di camminarci dentro, con un attraversamento:

```
livello.breadthFirstTraversal(new SceneGraphVisitor() {

    public void visit(Spatial spatial) {
```

```
            if(spatial.getName().equals("room")) {
                //creazione della controparte fisica del modello
                float massaLivello = 0; //massa 0 produce un corpo statico
                RigidBodyControl rbLivello = new RigidBodyControl(massaLivello);
                spatial.addControl(rbLivello);
                physics.getPhysicsSpace().add(rbLivello);
            }
        }
    });
```

A questo punto ho la mia scena 3d, composta quanto creato con blender, più la controparte fisica dell'elemento “room”. Ci sono anche i bersagli, non hanno un “RigidBody”, quindi ci si può camminare attraverso.



L'atto di sparare può essere riassunto come segue:

quando l'utente preme il pulsante sinistro del mouse
se l'utente ha ancora dei proiettili
    diminuisci il numero di proiettili di una unità
    riproduci il rumore dello sparo
    verifica se l'utente ha centrato un bersaglio

Nel framework logico proposto, la questione assume questa forma:

```
/* fire logic */
final LogicParam<MouseButton> fireButton = LogicParam.wrap(MouseButton.LEFT);
final LogicParam<Number> proiettili = LogicParam.<Number>wrap(10);
final LogicParam<Number> proiettiliPerColpo = LogicParam.<Number>wrap(-1);

final LogicCondition fireButtonPressed = new LCMousePressed(fireButton);
final LogicCondition proiettiliDisponibili =
    new LCGreaterThan(proiettili, LogicParam.<Number>wrap(0));

final LogicReaction riduciProiettili = new LRSum(proiettili, proiettiliPerColpo);
final LogicReaction suonoSparo = new LRDebug(LogicParam.wrap("suono sparo"));
final LogicReaction colpisciBersaglio = new LRDebug(LogicParam.wrap("colpisci bersaglio"));

logics.addTrigger(
    fireButtonPressed.and(proiettiliDisponibili),
    riduciProiettili.and(suonoSparo).and(colpisciBersaglio));
/* fire logic end */
```

I due LRDebug sono dei segnaposto. Il più semplice da definire è quello che riproduce il suono. JME3 integra delle api per la riproduzione di audio in 3D. Si carica un suono usando AssetManager e lo si associa ad un AudioNode. Dopodichè l'audio node può essere riprodotto così com'è (sono 2d) o inserito in una scena e riprodotto. La posizione del nodo nella scena determina la posizione del suono nello spazio. Se il nodo non è inserito nella scena, funziona come un audio clip e riproduce un suono aposizionale.

Supponiamo di avere un file “bang.wav” nella cartella res del progetto. Creiamo il nodo audio con:

```
final AudioNode bangNode = new AudioNode(getAssetManager(), "res/bang.wav");
```

La reazione che lo riproduce (suonoSparo) sarebbe:

```
final LogicReaction suonoSparo = new LogicReaction() {

  @Override
  public void act(LogicEnvironment logics) {
    logics.getAudioRenderer().playSourceInstance(bangNode);
  }
};
```

Per com'è fatto il “trigger”:

```
logics.addTrigger(
    fireButtonPressed.and(proiettiliDisponibili),
    riduciProiettili.and(suonoSparo).and(colpisciBersaglio));
```

Adesso sentiamo “bang” ogni volta che premiamo il pulsante sinistro del mouse se il numero di proiettili è maggiore di zero.

Passiamo al “colpisci bersaglio”. E' tutto precotto. Quel che facciamo è creare un raggio che va dal giocatore in avanti, verifichiamo se quel raggio interseca qualche elemento della scena, se esiste un'intersezione, quell'intersezione è la più vicina a noi ed è un bersaglio, abbiamo colpito qualcosa. Rimuoviamo quel qualcosa dalla scena e via.

```
final LogicReaction colpisciBersaglio = new LogicReaction() {
```

```
    @Override
    public void act(LogicEnvironment logics) {
      Camera camera = Game.this.getCamera();
      Node elements = Game.this.getRootNode();
      Ray ray = new Ray(camera.getLocation(), camera.getDirection());
      CollisionResults results = new CollisionResults();
      elements.collideWith(ray, results);
      if(results.size() > 0) {
        CollisionResult closestCollision = results.getClosestCollision();
        Geometry geometry = closestCollision.getGeometry();
        String nodeName = geometry.getParent().getName();
        if(nodeName.startsWith("target")) {
          geometry.getParent().removeFromParent();//hit
        }
      }
    }
  };
```

Qui prendo il nome del genitore della geometria perchè so che il caricatore dei file blender che ho scritto genera un nodo per ogni “mesh” di blender.

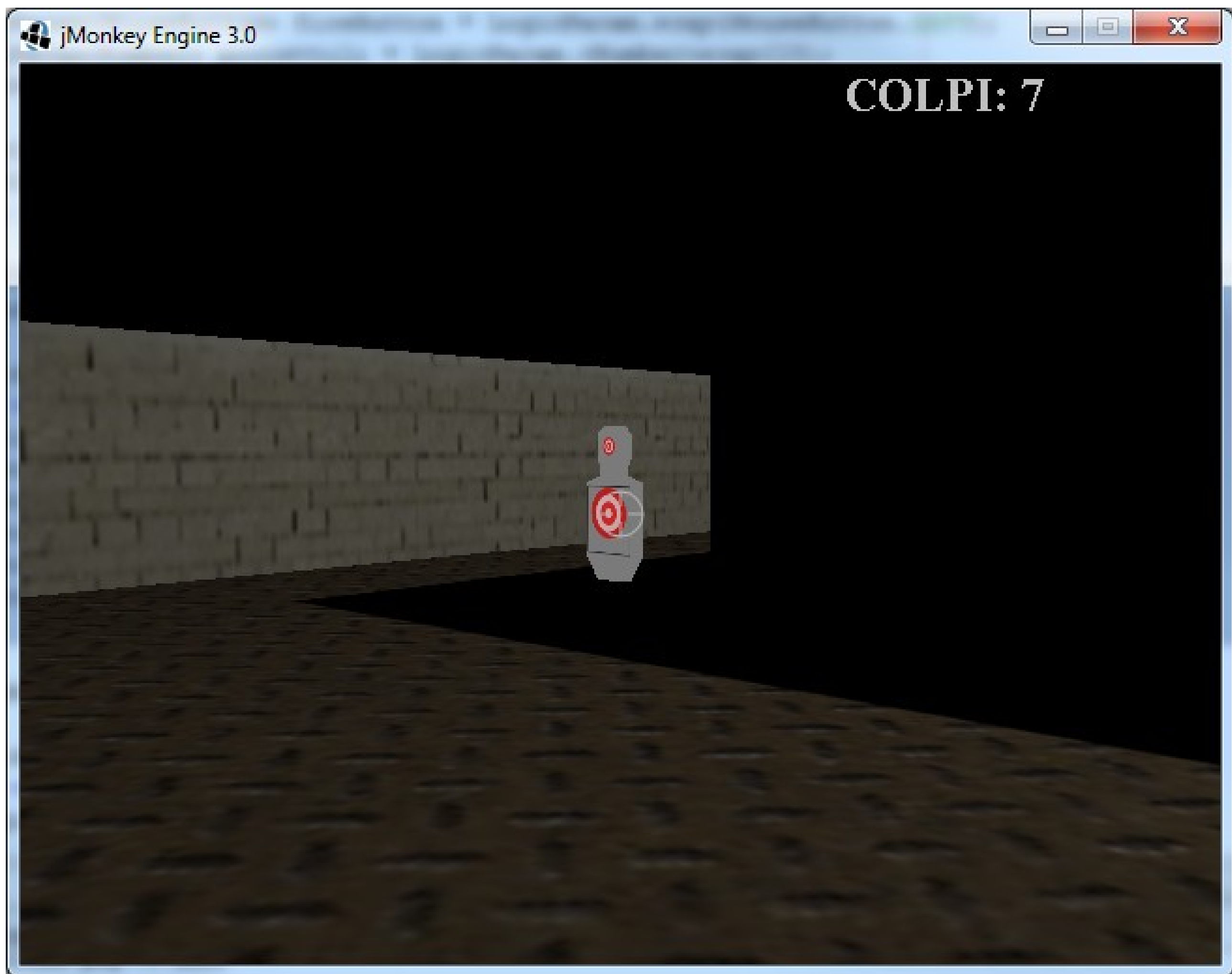
Supponiamo di voler aggiungere un contatore dei proiettili nella gui. Possiamo usare un PaintablePicture come “segnaposto”, dichiarato prima della logica dello sparo:

```
  /* hud */
  final PaintablePicture contatoreProiettili = PaintablePicture.newInstance(assetManager, 200, 50);
  contatoreProiettili.setPosition(getCamera().getWidth() - 200, getCamera().getHeight() - 50);
  contatoreProiettili.getGraphics().setPaint(java.awt.Color.LIGHT_GRAY);
  contatoreProiettili.getGraphics().setFont(new java.awt.Font("Serif", java.awt.Font.BOLD, 25));
  contatoreProiettili.getGraphics().drawString("COLPI: 10", 0, 25);
  contatoreProiettili.flush();
  getGuiNode().attachChild(contatoreProiettili);
  /* hud end */
```

Dopodichè “aggiungiamo” alla reazione “riduciProiettili” un'altra reazione che cambia la scritta mostrata in contatoreProiettili:

```
  final LogicReaction riduciProiettili = new LRSum(proiettili, proiettiliPerColpo)
      .and(new LogicReaction() {

    @Override
    public void act(LogicEnvironment logics) {
      String text = String.format("COLPI: %d", proiettili.value.intValue());
      contatoreProiettili.clear();
      contatoreProiettili.getGraphics().setPaint(Color.LIGHT_GRAY);
      contatoreProiettili.getGraphics().setFont(new Font("Serif", Font.BOLD, 25));
      contatoreProiettili.getGraphics().drawString(text, 0, 25);
      contatoreProiettili.flush();
    }
  });
```

E abbiamo la nostra scrittina.



Possiamo aggiungere il numero di bersagli totali e da colpire? Come no. Per contarli usiamo un visitor.

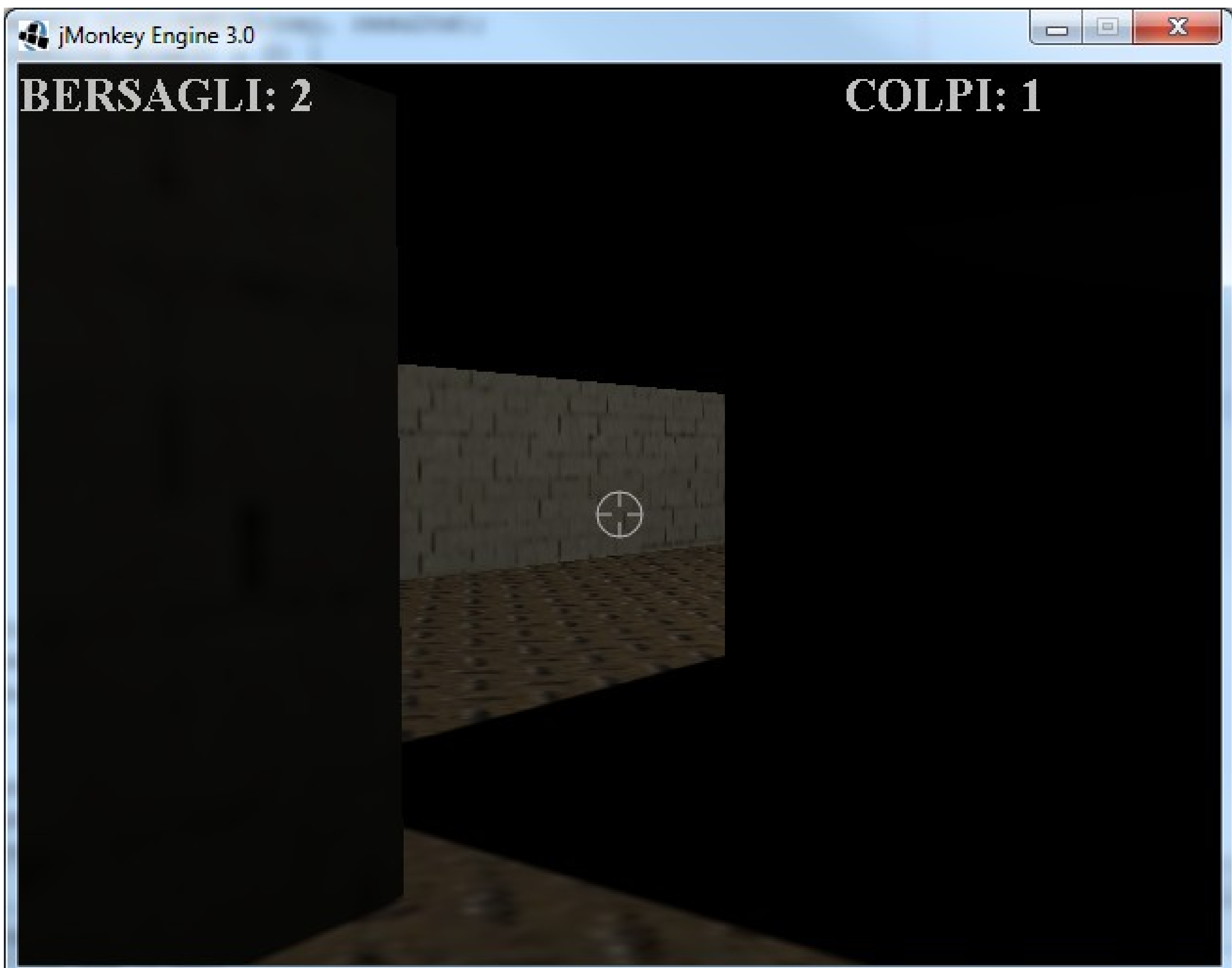
```
final LogicParam<Number> numeroBersagli = new LogicParam<Number>(0);
livello.breadthFirstTraversal(new SceneGraphVisitor() {

    public void visit(Spatial spatial) {
        String name = spatial.getName();
        if(name.startsWith("target")) {
            numeroBersagli.value = numeroBersagli.value.intValue() + 1;
        }
    }
});
```

Sempre basandoci sulla denominazione degli elementi. Aggiungiamo l'etichetta all'HUD:

```
final PaintablePicture contatoreBersagli = PaintablePicture.newInstance(assetManager, 200, 50);
contatoreBersagli.setPosition(0, getCamera().getHeight() - 50);
contatoreBersagli.getGraphics().setPaint(Color.LIGHT_GRAY);
contatoreBersagli.getGraphics().setFont(new Font("Serif", Font.BOLD, 25));
contatoreBersagli.getGraphics().drawString("BERSAGLI: "
        + numeroBersagli.value.intValue(), 0, 25);
contatoreBersagli.flush();
getGuiNode().attachChild(contatoreBersagli);
```

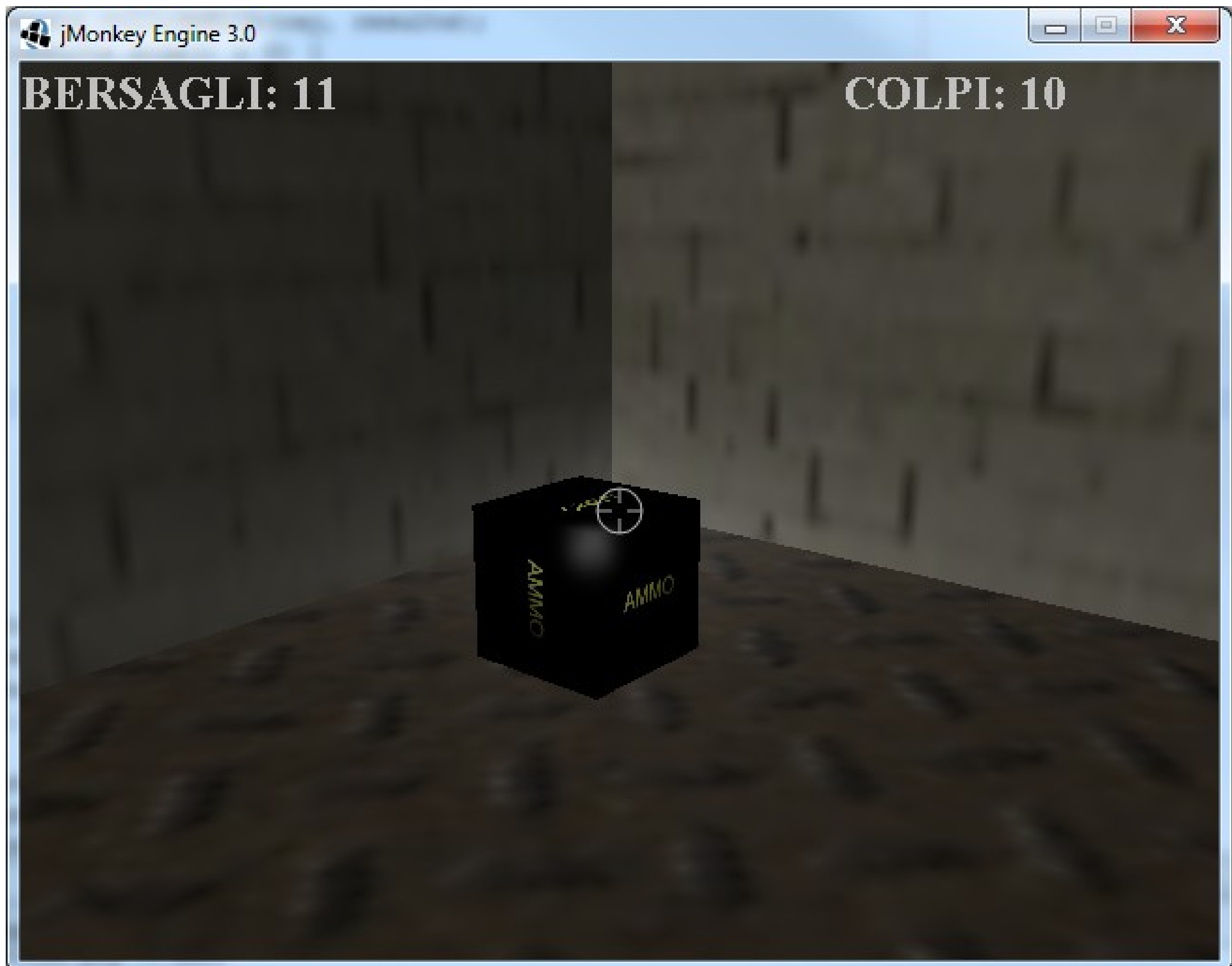
E la aggiorniamo come parte dell'azione colpisciBersaglio:

```
final LogicReaction colpisciBersaglio = new LogicReaction() {

    @Override
    public void act(LogicEnvironment logics) {
        Camera camera = Game.this.getCamera();
        Node elements = Game.this.getRootNode();
        Ray ray = new Ray(camera.getLocation(), camera.getDirection());
        CollisionResults results = new CollisionResults();
        elements.collideWith(ray, results);
        if(results.size() > 0) {
            CollisionResult closestCollision = results.getClosestCollision();
            Geometry geometry = closestCollision.getGeometry();
            String nodeName = geometry.getParent().getName();
            if(nodeName.startsWith("target")) {
                geometry.getParent().removeFromParent();//hit
                numeroBersagli.value = numeroBersagli.value.intValue() - 1;
                contatoreBersagli.clear();
                contatoreBersagli.getGraphics().setPaint(Color.LIGHT_GRAY);
                contatoreBersagli.getGraphics().setFont(new Font("Serif", Font.BOLD, 25));
                contatoreBersagli.getGraphics().drawString("BERSAGLI: "
                        + numeroBersagli.value.intValue(), 0, 25);
                contatoreBersagli.flush();
            }
        }
    }
};
```



Power-up (o pick up, quel che è).

Mettiamo nel nostro glorioso livello delle casse di munizioni. Vi risparmio il modello blender, sono quattro ignobili cubi che appaiono così:



Gli oggetti blender si chiamano "powerup.ammo.qualcosa". Raccattiamoli tutti in una lista. Abbiamo già un visitor che scandisce il contenuto del livello durante l'inizializzazione del gioco, lo ricicliamo:

```
final ArrayList<Spatial> ammoBoxes = new ArrayList<Spatial>();
final LogicParam<Number> numeroBersagli = new LogicParam<Number>(0);
livello.breadthFirstTraversal(new SceneGraphVisitor() {

    public void visit(Spatial spatial) {
        String name = spatial.getName();
        if(name.startsWith("target")) {
            numeroBersagli.value = numeroBersagli.value.intValue() + 1;
        } else if(name.startsWith("powerup.ammo")) {
            ammoBoxes.add(spatial);//
        }
    }
});
```

Ora on resta che verificare se la posizione del giocatore interseca in un qualsiasi istante il volume di uno degli "Spatial" che abbiamo incamerato in quella lista. In caso affermativo, aumentiamo il

numero dei colpi e rimuoviamo l'elemento dalla scena.

```
    /* power up logic */
    final LogicCondition pickupCondition = new LCAlways();
    final LogicReaction pickupAmmo = new LogicReaction() {

        @Override
        public void act(LogicEnvironment logics) {
            for (int i= 0; i < ammoBoxes.size(); i++) {
                Spatial box = ammoBoxes.get(i);
                BoundingVolume boxBounds = box.getWorldBound();
                Vector3f posizioneGiocatore = giocatore.getWorldTranslation();
                if(boxBounds.contains(posizioneGiocatore)) {
                    box.removeFromParent();
                    ammoBoxes.remove(i);
                    proiettili.value = proiettili.value.intValue() + 1;
                    Game.this.aggiornaContatoreProiettili(contatoreProiettili,
                            proiettili);
                    break;
                }
            }
        }
    };
    logics.addTrigger(pickupCondition, pickupAmmo);
    /* power up logic end */
```

Il metodo in rosso è questo:

```
private void aggiornaContatoreProiettili(PaintablePicture contatore, LogicParam<Number> proiettili) {
    String text = String.format("COLPI: %d", proiettili.value.intValue());
    contatore.clear();
    contatore.getGraphics().setPaint(Color.LIGHT_GRAY);
    contatore.getGraphics().setFont(new Font("Serif", Font.BOLD, 25));
    contatore.getGraphics().drawString(text, 0, 25);
    contatore.flush();
}
```

Che è poi la stessa cosa che abbiamo fatto nella reazione allo sparo: aggiorniamo l'etichetta. Volendo possiamo pescare il suono di un'arma ricaricata:

```
final AudioNode reloadAudio = new AudioNode(assetManager, "res/reload.wav");
```

E aggiungere prima del break nel codice di pickupAmmo:

```
logics.getAudioRenderer().playSource(reloadAudio);
```

L'effetto che ne risulta è un “click clock” quando passiamo sopra ad una cassa di munizioni mentre la gui registra l'aumento dei colpi disponibili.

Conclusione.

Incredibilmente non serve avere quattro lauree in geometria per girare in una stanza sparacchiando a destra e a sinistra. Se escludiamo la mancanza di figure animate (che il motore gestisce perfettamente, basta saperle creare con blender o altro ed esportarle nel formato ogre-xml) e una evidente carenza d'ispirazione artistica, per il resto jme3 con le due aggiunte presentate – peraltro neppure strettamente necessarie - sono più che sufficienti per scrivere un fps nel classico stile Quake (giri, raccogli, spari con armi diverse e fine della “meccanica” di gioco). Andate e

divertitevi.

Codice della classe principale.

```
package game;

import com.jme3.app.SimpleApplication;
import com.jme3.audio.AudioNode;
import com.jme3.bounding.BoundingVolume;
import com.jme3.bullet.BulletAppState;
import com.jme3.bullet.collision.shapes.SphereCollisionShape;
import com.jme3.bullet.control.CharacterControl;
import com.jme3.bullet.control.RigidBodyControl;
import com.jme3.collision.CollisionResult;
import com.jme3.collision.CollisionResults;
import com.jme3.input.KeyInput;
import com.jme3.light.PointLight;
import com.jme3.math.Quaternion;
import com.jme3.math.Ray;
import com.jme3.math.Vector3f;
import com.jme3.renderer.Camera;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.SceneGraphVisitor;
import com.jme3.scene.Spatial;
import com.jme3.texture.Texture2D;
import com.jme3.texture.plugins.AWTLoader;
import com.jme3.ui.Picture;
import java.awt.Color;
import java.awt.Font;
import java.util.ArrayList;
import java.util.logging.Level;
import java.util.logging.Logger;
import jme3blenderloader.BlenderFileLoader;
import jmex.logic.LogicCondition;
import jmex.logic.LogicEnvironment;
import jmex.logic.LogicParam;
import jmex.logic.LogicReaction;
import jmex.logic.MouseState.MouseButton;
import jmex.logic.basic.LCAlways;
import jmex.logic.basic.LCGreaterThan;
import jmex.logic.basic.LCKeyPressed;
import jmex.logic.basic.LCKeyReleased;
import jmex.logic.basic.LCMousePressed;
import jmex.logic.basic.LRSetValue;
import jmex.logic.basic.LRSum;
import jmex.ui.PaintablePicture;

public class Game extends SimpleApplication {

    public static void main(String[] args) {
        Logger.getLogger("com.jme3").setLevel(Level.WARNING);
        new Game().start();
    }

    private final Node giocatore = new Node("camera");
    private final LogicEnvironment logics = new LogicEnvironment();
```

```
@Override
public void simpleInitApp() {
    BlenderFileLoader.registerImageBasePath("res");
    getAssetManager().registerLoader(AWTLoader.class, "png");
    getAssetManager().registerLoader(BlenderFileLoader.class, "blend");

    Spatial livello = getAssetManager().loadModel("res/livello di prova.blend");
    getRootNode().attachChild(livello);

    PointLight light = new PointLight();
    light.setPosition(new Vector3f(0, 100, 0));
    getRootNode().addLight(light);

    //inizializzazione del simulatore fisico
    final BulletAppState physics = new BulletAppState();
    getStateManager().attach(physics);


    livello.breadthFirstTraversal(new SceneGraphVisitor() {

        public void visit(Spatial spatial) {
            if(spatial.getName().equals("room")) {
                //creazione della controparte fisica del modello
                float massaLivello = 0; //massa 0 produce un corpo statico
                RigidBodyControl rbLivello = new RigidBodyControl(massaLivello);
                spatial.addControl(rbLivello);
                physics.getPhysicsSpace().add(rbLivello);
            }
        }
    });

    //creazione della controparte fisica della camera
    float raggio = 1.5f;
    SphereCollisionShape volumeCamera = new SphereCollisionShape(raggio);

    float altezzaPasso = 0.2f;
    final CharacterControl ccCamera = new CharacterControl(volumeCamera, altezzaPasso);
    physics.getPhysicsSpace().add(ccCamera);

    giocatore.setLocalTranslation(0, 8, 0);//posizione iniziale del giocatore
    giocatore.addControl(ccCamera);
    getRootNode().attachChild(giocatore);

    getInputManager().deleteMapping("FLYCAM_ZoomIn");
    getInputManager().deleteMapping("FLYCAM_ZoomOut");
    getInputManager().deleteMapping("FLYCAM_Rise");
    getInputManager().deleteMapping("FLYCAM_Lower");

    setDisplayFps(false);
    setDisplayStatView(false);
    statsView.removeFromParent();
    fpsText.removeFromParent();

    //ui
    Picture mirino = new Picture("mirino");
    mirino.setWidth(32);
    mirino.setHeight(32);
    mirino.setTexture(assetManager, (Texture2D) assetManager.loadTexture("res/mirino.png"), true);
```

```
mirino.setPosition(getCamera().getWidth() / 2 - 16, getCamera().getHeight() / 2 - 16);
getGuiNode().attachChild(mirino);



//logics
logics.install(this);

final ArrayList<Spatial> ammoBoxes = new ArrayList<Spatial>();
final LogicParam<Number> numeroBersagli = new LogicParam<Number>(0);
livello.breadthFirstTraversal(new SceneGraphVisitor() {

    public void visit(Spatial spatial) {
        String name = spatial.getName();
        if(name.startsWith("target")) {
            numeroBersagli.value = numeroBersagli.value.intValue() + 1;
        } else if(name.startsWith("powerup.ammo")) {
            ammoBoxes.add(spatial);//
        }
    }
});

/* hud */
final PaintablePicture contatoreProiettili = PaintablePicture.newInstance(assetManager, 200, 50);
contatoreProiettili.setPosition(getCamera().getWidth() - 200, getCamera().getHeight() - 50);
contatoreProiettili.getGraphics().setPaint(java.awt.Color.LIGHT_GRAY);
contatoreProiettili.getGraphics().setFont(new java.awt.Font("Serif", java.awt.Font.BOLD, 25));
contatoreProiettili.getGraphics().drawString("COLPI: 10", 0, 25);
contatoreProiettili.flush();
getGuiNode().attachChild(contatoreProiettili);

final PaintablePicture contatoreBersagli = PaintablePicture.newInstance(assetManager, 200, 50);
contatoreBersagli.setPosition(0, getCamera().getHeight() - 50);
contatoreBersagli.getGraphics().setPaint(Color.LIGHT_GRAY);
contatoreBersagli.getGraphics().setFont(new Font("Serif", Font.BOLD, 25));
contatoreBersagli.getGraphics().drawString("BERSAGLI: "
        + numeroBersagli.value.intValue(), 0, 25);
contatoreBersagli.flush();
getGuiNode().attachChild(contatoreBersagli);

/* hud end */

/* fire logic */
final AudioNode bangNode = new AudioNode(getAssetManager(), "res/bang.wav");
final LogicParam<MouseButton> fireButton = LogicParam.wrap(MouseButton.LEFT);
final LogicParam<Number> proiettili = LogicParam.<Number>wrap(10);
final LogicParam<Number> proiettiliPerColpo = LogicParam.<Number>wrap(-1);

final LogicCondition fireButtonPressed = new LCMousePressed(fireButton);
final LogicCondition proiettiliDisponibili =
        new LCGreaterThan(proiettili, LogicParam.<Number>wrap(0));

final LogicReaction riduciProiettili = new LRSum(proiettili, proiettiliPerColpo)
        .and(new LogicReaction() {

    @Override
    public void act(LogicEnvironment logics) {
        String text = String.format("COLPI: %d", proiettili.value.intValue());
        Game.this.aggiornaContatoreProiettili(contatoreProiettili,
```

```
            proiettili);
    }
});
final LogicReaction suonoSparo = new LogicReaction() {

    @Override
    public void act(LogicEnvironment logics) {
        logics.getAudioRenderer().playSourceInstance(bangNode);
    }
};
final LogicReaction colpisciBersaglio = new LogicReaction() {

    @Override
    public void act(LogicEnvironment logics) {
        Camera camera = Game.this.getCamera();
        Node elements = Game.this.getRootNode();
        Ray ray = new Ray(camera.getLocation(), camera.getDirection());
        CollisionResults results = new CollisionResults();
        elements.collideWith(ray, results);
        if(results.size() > 0) {
            CollisionResult closestCollision = results.getClosestCollision();
            Vector3f contactPoint = closestCollision.getContactPoint();
            Geometry geometry = closestCollision.getGeometry();
            String nodeName = geometry.getParent().getName();
            if(nodeName.startsWith("target")) {
                geometry.getParent().removeFromParent();//hit
                numeroBersagli.value = numeroBersagli.value.intValue() - 1;
                contatoreBersagli.clear();
                contatoreBersagli.getGraphics().setPaint(Color.LIGHT_GRAY);
                contatoreBersagli.getGraphics().setFont(new Font("Serif", Font.BOLD, 25));
                contatoreBersagli.getGraphics().drawString("BERSAGLI: "
                        + numeroBersagli.value.intValue(), 0, 25);
                contatoreBersagli.flush();
            }
        }
    }
};

logics.addTrigger(
        fireButtonPressed.and(proiettiliDisponibili),
        riduciProiettili.and(suonoSparo).and(colpisciBersaglio));
/* fire logic end */

/* power up logic */
final AudioNode reloadAudio = new AudioNode(assetManager, "res/reload.wav");
final LogicCondition pickupCondition = new LCAlways();
final LogicReaction pickupAmmo = new LogicReaction() {

    @Override
    public void act(LogicEnvironment logics) {
        for (int i= 0; i < ammoBoxes.size(); i++) {
            Spatial box = ammoBoxes.get(i);
            BoundingVolume boxBounds = box.getWorldBound();
            Vector3f posizioneGiocatore = giocatore.getWorldTranslation();
            if(boxBounds.contains(posizioneGiocatore)) {
                box.removeFromParent();
                ammoBoxes.remove(i);
                proiettili.value = proiettili.value.intValue() + 1;
                Game.this.aggiornaContatoreProiettili(contatoreProiettili,
```

```
                    proiettili);
                logics.getAudioRenderer().playSource(reloadAudio);
                break;
            }
        }
    }
};
logics.addTrigger(pickupCondition, pickupAmmo);
/* power up logic end */

final LogicParam<Integer> keyUp = LogicParam.wrap(KeyInput.KEY_W);
final LogicParam<Integer> keyDown = LogicParam.wrap(KeyInput.KEY_S);
final LogicParam<Integer> keyLeft = LogicParam.wrap(KeyInput.KEY_A);
final LogicParam<Integer> keyRight = LogicParam.wrap(KeyInput.KEY_D);

final LogicParam<Float> playerDirX = LogicParam.wrap(0f);
final LogicParam<Float> playerDirY = LogicParam.wrap(0f);
final LogicParam<Float> playerDirZ = LogicParam.wrap(0f);

final LogicParam<Float> playerSpeed = LogicParam.wrap(0.25f);

logics.addTrigger(
        new LCKeyPressed(keyUp),
        new LRSetValue<Float>(playerDirZ, LogicParam.wrap(1f)));
logics.addTrigger(
        new LCKeyPressed(keyDown),
        new LRSetValue<Float>(playerDirZ, LogicParam.wrap(-1f)));
logics.addTrigger(
        new LCKeyPressed(keyLeft),
        new LRSetValue<Float>(playerDirX, LogicParam.wrap(1f)));
logics.addTrigger(
        new LCKeyPressed(keyRight),
        new LRSetValue<Float>(playerDirX, LogicParam.wrap(-1f)));
logics.addTrigger(
        new LCKeyReleased(keyUp).or(new LCKeyReleased(keyDown)),
        new LRSetValue<Float>(playerDirZ, LogicParam.wrap(0f)));
logics.addTrigger(
        new LCKeyReleased(keyLeft).or(new LCKeyReleased(keyRight)),
        new LRSetValue<Float>(playerDirX, LogicParam.wrap(0f)));
logics.addTrigger(new LCAlways(), new LogicReaction() {

    private final Vector3f cache = new Vector3f();

    @Override
    public void act(LogicEnvironment logics) {
        Camera camera = Game.this.getCamera();
        Quaternion cameraRotation = camera.getRotation();
        cache.set(playerDirX.value, playerDirY.value, playerDirZ.value);
        cameraRotation.multLocal(cache).multLocal(playerSpeed.value);
        cache.y = 0;
        ccCamera.setWalkDirection(cache);
        cache.set(giocatore.getLocalTranslation());
        cache.y += 4f;
        camera.setLocation(cache);
    }
});
}

@Override
```

```
    public void simpleUpdate(float tpf) {
        logics.update(tpf);
    }

    private void aggiornaContatoreProiettili(PaintablePicture contatore, LogicParam<Number> proiettili) {
        String text = String.format("COLPI: %d", proiettili.value.intValue());
        contatore.clear();
        contatore.getGraphics().setPaint(Color.LIGHT_GRAY);
        contatore.getGraphics().setFont(new Font("Serif", Font.BOLD, 25));
        contatore.getGraphics().drawString(text, 0, 25);
        contatore.flush();
    }
}
```