Scala tutorial pratico

Parte 6

Introduzione

Continuiamo con lo sviluppo dell'applicazione introdotta nella parte 5. A mo' di riepilogo, al momento il programma è diviso in tre sotto-progetti: paf, il cuore del programma, e paflauncher, che avvia il programma. Adesso arriva il terzo (e ultimo).

Il modulo del motore 3D.

L'editor 3d ha bisogno di un motore 3d. Cose da pazzi. Il motore 3d è probabilmente l'unica cosa di cui effettivamente esisteranno più versioni quindi introduciamo un'astrazione nel nucleo del sistema, paf, per dedicare un terzo sotto-progetto ad una sua specifica implementazione, quella che userà JMonkeyEngine3. Aggiungiamo a `paf` il trait che segue:

```
package it.tukano.paf.modules

import it.tukano.paf.core.PafModule
import java.awt.Component

/** Astrazione del motore 3d*/
trait Engine3D extends PafModule {

    /** Componente che mostra il contenuto 3D */
    def displayComponent(): Component
}
```

Ribadiamo che un trait Scala è una definizione non istanziabile che può possedere membri astratti il cui carattere tipico è la capacità di essere combinata con altre definizioni.
`Engine3D` al momento è poco più di un segnaposto, vedremo di volta in volta cosa aggiungere. Creo un terzo progetto, jme3pafengine, per dare una definizione di `Engine3D` che usi le librerie JMonkeyEngine3:

```
package it.tukano.paf.engines.jme3

import it.tukano.paf.modules.Engine3D
import it.tukano.paf.core.PafModuleMessage

/** Modulo 3d che usa jme3 come motore grafico */
trait JME3PafEngine extends Engine3D {

    private val canvas = new Canvas3D

    /** Componente che mostra il contenuto 3d del programma */
    def displayComponent() = canvas;

    /** Riceve un messaggio dall'ambiente */
    def parse(message: PafModuleMessage) = {}
}
```

L'uso della API di jme3 non è esattamente lo scopo di questo tutorial e d'altronde è una libreria ancora in alpha ma, per chi ne avesse la curiosità, questa è la definizione di `Canvas3D`.

```
package it.tukano.paf.engines.jme3
```

```
import com.jme3.app.SimpleApplication
import java.awt.GridLayout
import javax.swing.JPanel
import com.jme3.system.JmeCanvasContext

/** Component awt incluso in un JPanel */
class Canvas3D extends JPanel(new GridLayout(1, 1)) {
    private var context: JmeCanvasContext = _

    /** Applicazione jme3 */
    private val application = new SimpleApplication {
        def simpleInitApp = {
            //statsView.removeFromParent
            //fpsText.removeFromParent
            flyCam.setEnabled(false)
            inputManager.setCursorVisible(true)
        }
    }

    /** Avvia l'applicazione jme3 quando il componente diventa attivo */
    override def addNotify() {
        application.createCanvas
        context = application.getContext.asInstanceOf[JmeCanvasContext]
        add(context.getCanvas)
        setMinimumSize(new Dimension(64, 64))
        application.startCanvas
        super.addNotify
    }

    /** Ferma l'applicazione jme3 quando il componente è rimosso */
    override def removeNotify() {
        remove(context.getCanvas)
        super.removeNotify
        application.destroy
    }
}
```

Ciò che mi preme sottolineare è l'interoperabilità tra le librerie, qualsiasi esse siano, della piattaforma Java ed il linguaggio Scala. Col progetto jme3pafengine bell'e compilato – col suo jar artifact – salto a paflauncher:

```
package it.tukano.paf.launcher

import it.tukano.paf.core.PafEnvironment
import org.pushingpixels.substance.api.SubstanceLookAndFeel
import org.pushingpixels.substance.api.skin.RavenSkin
import javax.swing.{JDialog, JFrame}
import it.tukano.paf.modules.MainFrame
import it.tukano.paf.engines.jme3.JME3PafEngine

object Main extends Runnable {

    def main(args: Array[String]) = java.awt.EventQueue.invokeLater(this)

    def run() {
        SubstanceLookAndFeel.setSkin(classOf[RavenSkin].getName)
        JFrame.setDefaultLookAndFeelDecorated(true)
        JDialog.setDefaultLookAndFeelDecorated(true)
        val env = new PafEnvironment()
        env.register(
            new Object with MainFrame,
            new Object with JME3PafEngine)
```

```
        env.start()
    }
}
```

Aggiungo il nuovo modulo come ho fatto per `MainFrame`. Anche `JME3PafEngine` è un trait, non posso istanziarlo tout-court ma posso combinarlo con un'istanza compatibile. Apriamo una parentesi? E apriamola.

Quando parlo di composizione dei trait intendo la forma:

```
val x = new istanza with IlMioTrait1 with IlMioTrait2 with Eccetera
```

Questo genera un'istanza del trait che è anche istanza di qualcos'altro. Ci sono altre due forme per ottenere un'istanza da un trait, la combinazione con un object (che è un singleton):

```
object x extends IlMioTrait1 with IlMioTrait2 with Eccetera
```

e l'istanziazione tramite sotto-tipo anonimo:

```
val x = new IlMioTrait1 with IlMioTrait2 with Eccetera {

}
```

Ovviamente se un trait ha dei membri astratti (metodi o campi), devo definirli altrimenti il compilatore, giustamente, si lagna.

```
trait T0 {
      def metodo(): Unit//astratto
}

val x = new Object with T0 //no, manca la definizione di metodo

val x = new Object with T0 {
      def metodo = ()
} //ok, ho definito il membro astratto
```

I trait partecipano poi ai rapporti di derivazione nelle classi e nei trait:

```
class X extends UnTrait with UnAltroTrait {

}

trait Y extends UnTrait with UnAltroTrait {

}
```

Si tratta sempre di combinare delle definizioni, solo che qui non ci sono istanze finchè non si creano da qualche altra parte.
C'è poi un'altra carineria dei trait, nella forma:

```
trait X { this: Y =>

}
```

Questa roba significa che il trait `X` è combinabile, sia per derivazione che per composizione, con tipi che siano almeno degli `Y` (lo fa stabilendo che il tipo di `this` dev'essere almeno `Y`). Esempio:

```
trait Trait  { this: Container =>

}

class C extends Trait //non va bene, chi è il container?

class C extends Panel with Trait//ok, c'è il container

val x = new Object with Trait //no

val x = new Panel with Trait//si
```

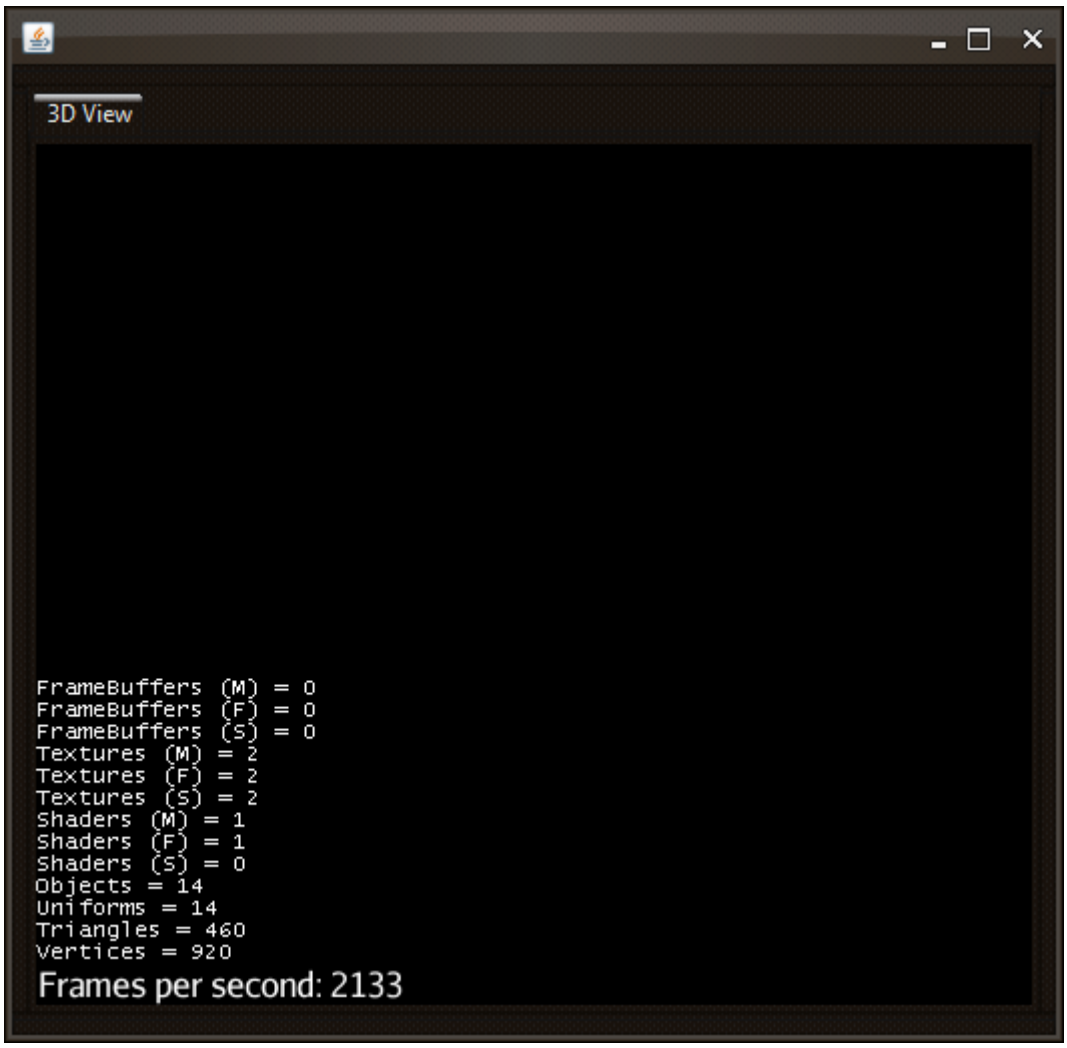
Sa un po' di curiosità da almanacco dell'enigmista – e sinceramente credo che abbia uno senso più pregnante quando si inizia a pasticciare coi tipi generici – ma è un modo per dire "ok, questo trait è l'appendice di un'altra cosa".

Chiusa parentesi.

Abbiamo inserito nel lanciatore il modulo del motore 3d. Il tipo `Engine3D` definisce un metodo che restituisce un componente. Ovvio che quel componente vada infilato da qualche parte. Lo mettiamo al centro della finestra principale. Per farlo modifichiamo `MainFrame` (in paf):

```
//MainFrame.scala
    /** Esamina i messaggi inviati dall'ambiente a questo modulo */
    def parse(message: PafModuleMessage) = {
        import PafModuleMessage._

        message match {
            case START => installModuleComponentsAndShow()
            case DESTROY => destroyWindow()
            case _ =>
        }
    }

    private def installModuleComponentsAndShow() {
        val engineDisplay = env(classOf[Engine3D]).displayComponent
        if(engineDisplay.getName == null) engineDisplay.setName("3D View")
        centerTab.addTab(engineDisplay.getName, engineDisplay)
        window.setVisible(true)
    }
```

La funzione `classOf[T]` è la controparte Scala di `T.class`, con `T` nome di un tipo (classe, trait, object). L'invocazione `env(classOf[Engine3D])` corrisponde all'invocazione `env.apply(classOf[Engine3D])`, perchè l'`apply` si può omettere – ok, è l'ultima volta che lo dico, promesso. Compilando tutto, creando i jar necessari a paflauncher ed eseguendo otteniamo questa finestra:



Gestione della vista: preliminari.

Aggiungiamo un modulo per controllare le impostazioni della vista sulla scena 3d: posizione, rotazione, con o senza griglie, con o senza assi eccetera. Mutuando il lookAt di OpenGl, possiamo definire una "vista" come un punto a partire dal quale se ne guarda un altro. Per farlo servono due punti e vettore (il vettore decide se sto guardando con la testa dritta, piegata o coi piedi per aria). Creiamo una classe per punti e vettori.

```
package it.tukano.paf.math

class NTuple3(val x: Number, val y: Number, val z: Number)
```

Gli argomenti di un costruttore scala diventano automaticamente campi di quella classe. Quando si dichiara un argomento in un costruttore le opzioni sono tre: o metto `var`, o metto `val` o non metto niente.

```
class C(x: Number)

class C(val x: Number)

class C(var x: Number)
```

Non mettere alcunchè equivale a scrivere private val, cio dichiaro un campo non riassegnabile accessibile solo nella classe `C` e nel companion object di `C` (se gliene do uno). Se uso `var` il valore

del campo (pubblico) è variabile. Usando `val` o `var` è possibile specificare il modificatore d'accesso:

```
class C(private var x: Number)
class C(protected val x: Number)
eccetera
```

Dunque `NTuple3` è una classe con tre campi pubblici, immutabili, di tipo `Number`, inizializzati nel costruttore.
Introduciamo un tipo che rappresenti il tratto comune tra tutti gli elementi di una scena 3d.

```
package it.tukano.paf.scene

/** Elemento di una scena 3d */
trait SceneElement {

    /** identificatore univoco dell'elemento */
    val uid: String

    /** nome dell'elemento */
    val name: String
}
```

Il campo uid serve per identificare l'elemento, il campo name come etichetta per la gui. I dati per stabilire un punto di osservazione della scena 3d potrebbero essere questi:

```
package it.tukano.paf.scene

import it.tukano.paf.math.NTuple3

class ViewElement(val name: String,
                  val uid: String,
                  val position: NTuple3,
                  val rotation: NTuple3) extends SceneElement
```

Tutto `val`, tutto immutabile quindi. La gui lavora nel thread EDT, il motore grafico nel suo thread, meno si cambia e meglio è. Ovviamente c'è un prezzo da pagare: se da un lato possiamo inviare e ricevere dati senza che questo comporti la necessità di sincronizzare gli accessi alla memoria dall'altro ogni variazione dei dati comporta la creazione di un nuovo oggetto. Il costo è tuttavia sostenibile perchè il motore 3d userà (sperabilmente) i suoi “primitivi” per la rappresentazione grafica: i nostri servono solo per lo scambio di dati – che avviene nei tempi della gui.
Aggiungiamo al motore un metodo per l'impostazione della vista corrente:

```
package it.tukano.paf.modules

import it.tukano.paf.core.PafModule
import java.awt.Component
import it.tukano.paf.scene.ViewElement

/** Astrazione del motore 3d*/
trait Engine3D extends PafModule {

    /** Componente che mostra il contenuto 3D */
    def displayComponent(): Component

    /** Imposta la vista corrente */
    def setCurrentView(e: ViewElement): Unit
}
```

E ne diamo una definizione in jme3pafengine:

```
package it.tukano.paf.engines.jme3

import it.tukano.paf.modules.Engine3D
import it.tukano.paf.core.PafModuleMessage
import it.tukano.paf.scene.ViewElement

/** Modulo 3d che usa jme3 come motore grafico */
trait JME3PafEngine extends Engine3D {

    private val canvas = new Canvas3D

    /** Componente che mostra il contenuto 3d del programma */
    def displayComponent() = canvas;

    /** Riceve un messaggio dall'ambiente */
    def parse(message: PafModuleMessage) = {}

    def setCurrentView(view: ViewElement) = {

    }
}
```

Che ci scriviamo. Bisogna tradurre i dati di view in qualcosa di comprensibile a jme3. Il punto non è come ma dove farlo. Il motore ha un suo thread, quindi dobbiamo fare un salto. In primis, cambio una virgola in Canvas3D, aggiugendo:

```
package it.tukano.paf.engines.jme3

import com.jme3.app.SimpleApplication
import javax.swing.JPanel
import com.jme3.system.JmeCanvasContext
import java.awt.{Dimension, GridLayout}
import java.util.concurrent.Callable

/** Component awt incluso in un JPanel */
class Canvas3D extends JPanel(new GridLayout(1, 1)) {

...omissis...

    /** Invoca una funzione nel thread di jme */
    def call(fun: (SimpleApplication) => Any) {
        application.enqueue(new Callable[Unit] {
            def call = fun(application);
        })
    }
}
```

Il metodo `call` riceve una funzione che fa qualcosa con un `SimpleApplication` e, tramite una coda integrata in jme3, fa sì che quella funzione sia invocata nel thread che gestisce il ciclo di rendering del motore grafico. Con questo call possiamo dire:

```
package it.tukano.paf.engines.jme3

import it.tukano.paf.modules.Engine3D
import it.tukano.paf.core.PafModuleMessage
import it.tukano.paf.scene.ViewElement

/** Modulo 3d che usa jme3 come motore grafico */
trait JME3PafEngine extends Engine3D {
```

```
    private val canvas = new Canvas3D

    /** Componente che mostra il contenuto 3d del programma */
    def displayComponent() = canvas;

    /** Riceve un messaggio dall'ambiente */
    def parse(message: PafModuleMessage) = {}

    /** Imposta la posizione della vista */
    def setCurrentView(view: ViewElement) = canvas.call{ app =>
        val cam = app.getCamera
        cam.setLocation(view.position)
        val quaternion = new Quaternion()
        quaternion.fromAngles(
            view.rotation.x.floatValue,
            view.rotation.y.floatValue,
            view.rotation.z.floatValue)
        cam.setRotation(quaternion)
    }
}
```

Possiamo? C'è un intoppo. JME3 lavora coi suoi tipi, noi coi nostri. Il valore di view.position è di tipo NTuple3. Le api di jme3 richiedono un Vector3f. Non c'è alcuna relazione tra NTuple3 e Vector3f tuttavia noi possiamo istruire il compilatore scala a tradurre l'uno nell'altro dichiarando una conversione implicita. Lo facciamo in un tipo a parte per radunare in un punto tutte le funzioni di conversione che, magari, potranno servirci anche in altre classi.

```
package it.tukano.paf.engines.jme3

import com.jme3.math.Vector3f
import it.tukano.paf.math.NTuple3

trait PafJmeConversions {

    implicit def NTuple3ToVector3f(tuple: NTuple3): Vector3f = {
        new Vector3f(tuple.x.floatValue, tuple.y.floatValue, tuple.z.floatValue)
    }
}
```

Incrociamo l'ambito di `JME3PafEngine` con quello della nostra conversione implicita con un'estensione:

```
package it.tukano.paf.engines.jme3

import it.tukano.paf.modules.Engine3D
import it.tukano.paf.core.PafModuleMessage
import it.tukano.paf.scene.ViewElement

/** Modulo 3d che usa jme3 come motore grafico */
trait JME3PafEngine extends Engine3D with PafJmeConversions {
```

Vale a dire che sbattiamo sotto al naso del compilatore l'esistenza di questa conversione da `NTuple3` a `Vector3f` nella parte del programma dove effettivamente usiamo un `NTuple3` come fosse un `Vector3f`. E il gioco è fatto: il codice di prima

```
    /** Imposta la posizione della vista */
    def setCurrentView(view: ViewElement) = canvas.call{ app =>
        val cam = app.getCamera
        cam.setLocation(view.position)
```

```
        val quaternion = new Quaternion()
        quaternion.fromAngles(
            view.rotation.x.floatValue,
            view.rotation.y.floatValue,
            view.rotation.z.floatValue)
        cam.setRotation(quaternion)
    }
```

da errato diventa corretto. D'altronde, per quanto siano belle e utili, io in queste conversioni implicite vedo un problema di comprensibilità. Il fatto che un enunciato richieda un tipo A e io gli passi un B senza che questo comporti alcun lamento del compilatore lascia perplessi a meno che non sia evidente il richiamo alla conversione. D'altronde star lì a invocare esplicitamente il metodo che trasforma un NTuple3 in un Vector3f è chiaramente una ripetizione: una volta che il lettore del codice sa che esiste una funzione che può convertire, perchè annoiarlo. Dichiarando le conversioni in un tipo a parte ed usandolo come genitore della classe in cui uso la conversione può essere un rimedio:

```
trait JME3PafEngine extends Engine3D with PafJmeConversions {
```

nel senso che chi legge si accorge che è in ambito un qualcosa che fa delle conversioni tra i tipi e quindi può immaginare che ci sia qualcosa che trasforma un `NTuple3` in un `Vector3f` anche senza andare a cercarlo. Il problema è che la nostra annotazione compare solo nell'intestazione del sorgente. Si può fare di meglio incrociando due cose: le dichiarazioni di importazione “locali”, che abbiamo già visto nelle puntate precedenti, e i `package object`, che vediamo adesso. Otteniamo questo:

```
package it.tukano.paf.engines.jme3

import it.tukano.paf.modules.Engine3D
import it.tukano.paf.core.PafModuleMessage
import it.tukano.paf.scene.ViewElement

/** Modulo 3d che usa jme3 come motore grafico */
trait JME3PafEngine extends Engine3D {

    private val canvas = new Canvas3D

    /** Componente che mostra il contenuto 3d del programma */
    def displayComponent() = canvas;

    /** Riceve un messaggio dall'ambiente */
    def parse(message: PafModuleMessage) = {}

    /** Imposta la posizione della vista */
    def setCurrentView(view: ViewElement) = canvas.call{ app =>
        import conversions._

        val cam = app.getCamera
        cam.setLocation(view.position)
        val quaternion = new Quaternion()
        quaternion.fromAngles(
            view.rotation.x.floatValue,
            view.rotation.y.floatValue,
            view.rotation.z.floatValue)
        cam.setRotation(quaternion)
    }
}
```

Lo otteniamo aggiungendo al package `it.tukano.paf.engines.jme3` un nuovo tipo:

```
package it.tukano.paf.engines.jme3

import it.tukano.paf.math.NTuple3
import com.jme3.math.Vector3f

package object conversions {

    implicit def NTuple3ToVector3f(tuple: NTuple3): Vector3f = {
        new Vector3f(tuple.x.floatValue, tuple.y.floatValue, tuple.z.floatValue)
    }
}
```

Olalà, una novità. Un `package object` è un modo per dichiarare delle funzioni e dei valori come membri diretti di un package anziché come membri di un tipo (`class`, `trait`, `object`) appartenente a quel package. Per la parte che interessa a noi (cioè a me, voi mi venite dietro), vale solo per poter etichettare un pezzetto del codice in modo tale da rendere evidente che c'è qualcosa al lavoro nell'ombra ma i package object sono usati, ad esempio, per dichiarare le funzioni matematiche delle api di scala (c'è un `package object` di nome `math` che contiene le funzioni `abs`, `floor`, il valore di pi greco eccetera).
Ora siamo in grado di spostare la vista sulla nostra scena 3d. Non che sia molto utile perchè non abbiamo un bel nulla da vedere ma pian piano arriveremo anche a quello. Passiamo al modulo di controllo.

Gestione della vista: il modulo di controllo.

Il modulo nudo e crudo potrebbe essere questo:

```
package it.tukano.paf.modules

import it.tukano.paf.core.{PafModuleMessage, PafModule}
import javax.swing.JPanel
import it.tukano.paf.utils.swing.ContainerExt

/** Controlli della vista sulla scena 3D */
trait ViewController extends PafModule {

    /** Componente dell'interfaccia */
    private val component = new JPanel with ContainerExt

    /** Restituisce il componente dell'interfaccia */
    def displayComponent = component;

    /** Gestisce i messaggi inviati dall'ambiente a questo modulo */
    def parse(message: PafModuleMessage) = {

    }
}
```

Lo installiamo così com'è nel lanciatore:

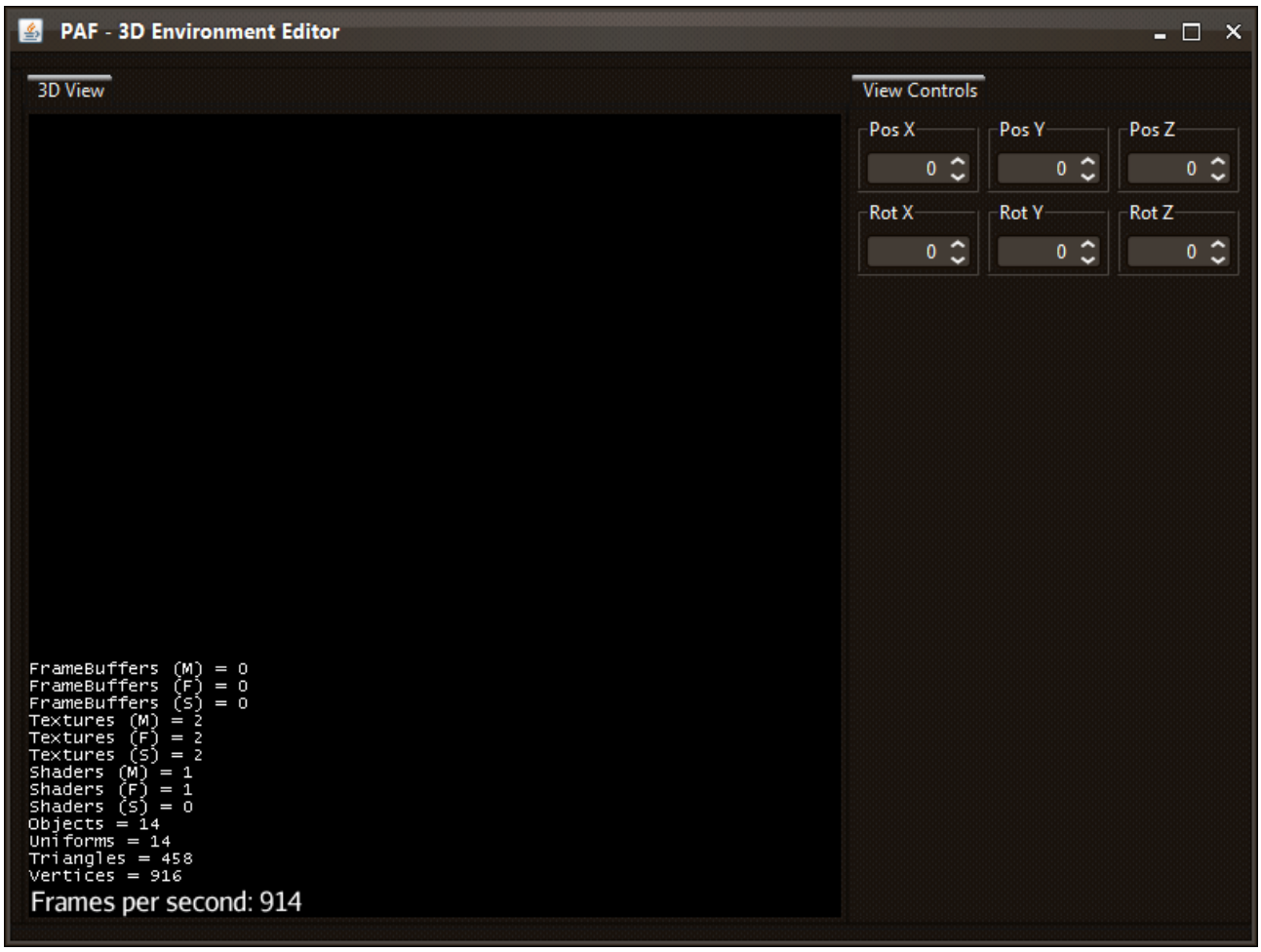
```
package it.tukano.paf.launcher

import it.tukano.paf.core.PafEnvironment
import org.pushingpixels.substance.api.SubstanceLookAndFeel
import org.pushingpixels.substance.api.skin.RavenSkin
import javax.swing.{JDialog, JFrame}
import it.tukano.paf.engines.jme3.JME3PafEngine
import it.tukano.paf.modules.{ViewController, MainFrame}
```

```
object Main extends Runnable {

    def main(args: Array[String]) = java.awt.EventQueue.invokeLater(this)

    def run() {
        SubstanceLookAndFeel.setSkin(classOf[RavenSkin].getName)
        JFrame.setDefaultLookAndFeelDecorated(true)
        JDialog.setDefaultLookAndFeelDecorated(true)
        val env = new PafEnvironment()
        env.register(
            new Object with MainFrame,
            new Object with JME3PafEngine,
            new Object with ViewController)
        env.start()
    }
}
```

e nella finestra principale:

```
//MainFrame.scala
...omissis...

    private def installModuleComponentsAndShow() {
        val viewControlDisplay = env(classOf[ViewController]).displayComponent
        val engineDisplay = env(classOf[Engine3D]).displayComponent
        if(engineDisplay.getName == null) engineDisplay.setName("3D View")
        eastTab.addTab(viewControlDisplay.getName, viewControlDisplay)
        centerTab.addTab(engineDisplay.getName, engineDisplay)
        window.setVisible(true)
    }
...omissis...
```

Il modulo di controllo della visuale contiene inizialmente due campi per determinare posizione e rotazione della telecamera. Visivamente è quello a sinistra nella figura che segue:



Il suo codice usa un tot di classi intermedie. La prima è:

```
package it.tukano.paf.utils.swing

import javax.swing.{BorderFactory, JPanel}
import java.awt.{Component, GridLayout}

/** JPanel con un bordo titolato ed un layout predefinito ad una cella */
class TitledContainer(title: String) extends JPanel(new GridLayout(1, 1)) {

    setBorder(BorderFactory.createTitledBorder(title))

    /** Costruttore alternativo */
    def this(title: String, content: Component) {
        this(title)
        add(content)
    }
}
```

Qui vediamo all'opera un costruttore principale, un super-costruttore ed un costruttore alternativo. Le classi Scala hanno un costruttore principale definito nell'intestazione della classe stessa

```
class TitledContainer(title: String)
```

dopo il nome. Se una classe ne estende un'altra, la classe estesa ha più di un costruttore e la sottoclasse deve scegliere quale tra questi deva essere invocato all'atto dell'inizializzazione, l'invocazione del super-costruttore prescelto è fatta sempre nell'intestazione:

```
extends JPanel(new GridLayout(1, 1))
```

In questo caso il super-costruttore richiamato è il costruttore di `JPanel` che accetta un `LayoutManager` come parametro. Oltre al costruttore principale, una classe Scala può avere più costruttori secondari, nella forma:

```
def this(lista parametri) = {
    invocazione del costruttore principale
    altri enunciati
}
```

oppure

```
def this(lista parametri) {
    invocazione del costruttore principale
    altri enunciati
}
```

I costruttori in questione sono definiti secondari in quanto obbligati a fornire i parametri necessari all'esecuzione del costruttore principale. Vale a dire che se ho una classe con un costruttore principale che richiede quattro interi:

```
class Classe(x: Int, y: Int, z: Int, w: Int) {
}
```

Posso introdurre un costruttore secondario che accetta una stringa:

```
class Classe(x: Int, y: Int, z: Int, w: Int) {

    def this(s: String) {
    }
}
```

La cui prima istruzione deve necessariamente essere l'invocazione del costruttore principale, e quindi devo alla peggio inventarmi dei valori per i parametri che quel costruttore richiede:

```
class Classe(x: Int, y: Int, z: Int, w: Int) {

    def this(s: String) {
        this(0, 0, 0, 0)
        ...altro...
    }
}
```

Tornando a `TitledContainer`, il suo scopo è, banalmente, quello di circondare un altro componente con un titolo. La classe è usata dal componente che gestisce la modifica di un valore `NTuple3`:

```
package it.tukano.paf.utils.swing

import javax.swing.{JSpinner, SpinnerNumberModel, JPanel}
import it.tukano.paf.math.NTuple3
import java.awt.GridLayout

/** Editor per valori di tipo NTuple3 */
class NTuple3Editor(val labelX: String = "X",
                    val labelY: String = "Y",
                    val labelZ: String = "Z") {
```

```
    /** Metodo d'appoggio, genera un modello numerico per uno spinner */
    private def newModel() = new SpinnerNumberModel(0.0, null, null, 1.0)

    /** Metodo d'appoggio, genera uno spinner con un modello numerico */
    private def DoubleSpinner() = new JSpinner(newModel())

    /** Controlli dei valori x, y, z*/
    private val xEditor = DoubleSpinner()
    private val yEditor = DoubleSpinner()
    private val zEditor = DoubleSpinner()

    /** Contenitore dei controlli, applicano un titolo intorno ai componenti */
    private val xComponent = new TitledContainer(labelX, xEditor)
    private val yComponent = new TitledContainer(labelY, yEditor)
    private val zComponent = new TitledContainer(labelZ, zEditor)

    /** Il pannello che appare nella gui */
    val component = (new JPanel with ContainerExt).
        withLayout(new GridLayout(1, 3)).
        append(xComponent).
        append(yComponent).
        append(zComponent)

    /** Imposta il valore corrente dell'editor */
    def setEditedValue(e: NTuple3) {
        xEditor.setValue(e.x.doubleValue)
        yEditor.setValue(e.y.doubleValue)
        zEditor.setValue(e.z.doubleValue)
    }

    /** Restituisce il valore corrente dell'editor */
    def getEditedValue() = {
        new NTuple3(xEditor.getValue.asInstanceOf[Number],
            yEditor.getValue.asInstanceOf[Number],
            zEditor.getValue.asInstanceOf[Number])
    }
}
```

La novità in questa classe è l'uso di valori predefiniti per gli argomenti di un costruttore – vale anche per i metodi.

```
class NTuple3Editor(val labelX: String = "X",
                    val labelY: String = "Y",
                    val labelZ: String = "Z") {
```

Il costruttore ha tre parametri, di tipo `String`, ciascuno dei quali ha un valore predefinito. Il valore predefinito è applicato nel caso in cui l'utente – della classe, cioè il programmatore – non specifichi un valore diverso. Significa che posso invocare il costruttore scrivendo:

```
new NTuple3Editor()
```

usando quindi i valori predefiniti `X`, `Y`, `Z` per i parametri `labelX`, `labelY` e `labelZ`, oppure scrivendo:

```
new NTuple3Editor("a", "b", "c")
```

assegnando i valori `a`, `b`, `c` ai parametri `labelX`, `labelY`, `labelZ`, oppure scrivendo:

```
new NTuple3Editor(labelY = "pippo")
```

assegnando il valore `pippo` al parametro `labelY` e usando i valori predefiniti per i parametri `labelX` e `labelZ`.

Due NTuple3Editor sono usati dal controllo della vista, uno per la posizione e uno per la rotazione:

```
package it.tukano.paf.modules

import it.tukano.paf.core.{PafModuleMessage, PafModule}
import javax.swing.JPanel
import it.tukano.paf.utils.swing.{NTuple3Editor, ContainerExt}
import java.awt.{GridLayout, BorderLayout}

/** Controlli della vista sulla scena 3D */
trait ViewController extends PafModule {

    /** Editor rotazione-posizione */
    private val locationEditor = new NTuple3Editor("Pos X", "Pos Y", "Pos Z")
    private val rotationEditor = new NTuple3Editor("Rot X", "Rot Y", "Rot Z")

    /** Contenitore degli editor */
    private val editorContainer = (new JPanel with ContainerExt).
        withLayout(new GridLayout(2, 1)).
        append(locationEditor.component).
        append(rotationEditor.component)

    /** Componente dell'interfaccia */
    val displayComponent = (new JPanel with ContainerExt).
        withName("View Controls").
        withLayout(new BorderLayout).
        append(editorContainer -> BorderLayout.NORTH).
        withPreferredSize(250, 200)

    /** Gestisce i messaggi inviati dall'ambiente a questo modulo */
    def parse(message: PafModuleMessage) = {
        import PafModuleMessage._

        message match {
            case START => setDefaultViewPosition()
            case _ =>
        }
    }

    /** All'avvio imposta la posizione predefinita della vista */
    def setDefaultViewPosition() {
        //todo
    }
}
```

Qui c'è poco da dire, salvo forse notare che abbiamo aggiunto un paio di metodi a `ContainerExt`, uno imposta il nome del componente, uno imposta la dimensione preferita, un altro per aggiungere un singolo componente senza specificare i limiti del layout.
Introduciamo un decoratore per il tipo `Frame`:

```
package it.tukano.paf.utils.swing

import java.awt.Frame

/** Decorazioni per il tipo Frame */
trait FrameExt extends Frame {
```

```
    /** Imposta il titolo della finestra */
    def withTitle(s: String): this.type = {
        setTitle(s)
        this
    }
}
```

Da usare in MainFrame per impostare il titolo:

```
    private val window =
        (new JFrame() with WindowExt with FrameExt with ContainerExt).
        onClosing(closeApplication).
        withLayout(new BorderLayout).
        withSize(800, 600).
        withTitle("PAF - 3D Environment Editor").
        append(mainContainer -> BorderLayout.CENTER)
```

Onde evitare equivoci, non è necessario far 'sto giro per impostare il titolo di una finestra, è solo una mia personale passione per la concatenazione delle invocazioni, sfruttata per mostrare una caratteristica del linguaggio – la composizione o, per dirla alla Brachese, la composizionalità.
Il tutto genera l'interfaccia che abbiamo visto.
Ora facciamo fare qualcosa all'interfaccia. Quando l'utente cambia i valori degli spinner, il sistema sposta la visuale 3d. Le azioni da compiere sono relativamente banali: quando l'utente cambia un valore in uno degli spinner, creiamo un nuovo `ViewElement` coi valori contenuti nell'editor e lo passiamo al modulo del motore 3d tramite il suo metodo `setCurrentView`. Il primo passo è intercettare gli eventi prodotti dagli `NTuple3Editor`:

```
package it.tukano.paf.utils.swing

import javax.swing.{JSpinner, SpinnerNumberModel, JPanel}
import it.tukano.paf.math.NTuple3
import java.awt.GridLayout
import javax.swing.event.{ChangeEvent, ChangeListener}

/** Editor per valori di tipo NTuple3 */
class NTuple3Editor(val labelX: String = "X",
                    val labelY: String = "Y",
                    val labelZ: String = "Z") {

    /** Metodo d'appoggio, genera un modello numerico per uno spinner */
    private def newModel() = new SpinnerNumberModel(0.0, null, null, 1.0)

    /** Ascoltatori di evento di quest'editor */
    private var listeners = List[ChangeListener]()

    /** Ascoltatori di evento degli spinner */
    private val spinnerListener: ChangeListener = new ChangeListener {
        def stateChanged(e: ChangeEvent) = if(!listeners.isEmpty) {
            val editedValue = getEditedValue
            val event = new ChangeEvent(this)
            System.out.println("notify listeners...")
            listeners.foreach(listener => listener.stateChanged(event))
        }
    }

    /** Metodo d'appoggio, genera uno spinner con un modello numerico */
    private def DoubleSpinner() = {
        val spinner = new JSpinner(newModel())
        spinner.addChangeListener(spinnerListener)
        spinner
    }
```

```
    /** Controlli dei valori x, y, z*/
    private val xEditor = DoubleSpinner()
    private val yEditor = DoubleSpinner()
    private val zEditor = DoubleSpinner()

    /** Contenitore dei controlli, applicano un titolo intorno ai componenti */
    private val xComponent = new TitledContainer(labelX, xEditor)
    private val yComponent = new TitledContainer(labelY, yEditor)
    private val zComponent = new TitledContainer(labelZ, zEditor)

    /** Aggiunge un ascoltatore di eventi all'editor */
    def addChangeListener(listener: ChangeListener): this.type = {
        listeners = listeners :+ listener
        this
    }

    /** Rimuove un ascoltatore di eventi dall'editor */
    def removeChangeListener(listener: ChangeListener): this.type = {
        listeners = listeners.filterNot(_ == listener)
        this
    }

    /** Il pannello che appare nella gui */
    val component = (new JPanel with ContainerExt).
        withLayout(new GridLayout(1, 3)).
        append(xComponent).
        append(yComponent).
        append(zComponent)

    /** Imposta il valore corrente dell'editor */
    def setEditedValue(e: NTuple3) {
        detachSpinnerListener()
        xEditor.setValue(e.x.doubleValue)
        yEditor.setValue(e.y.doubleValue)
        zEditor.setValue(e.z.doubleValue)
        attachSpinnerListener()
    }

    /** Rimuove l'ascoltatore di eventi dagli spinner */
    def detachSpinnerListener() {
        xEditor.removeChangeListener(spinnerListener)
        yEditor.removeChangeListener(spinnerListener)
        zEditor.removeChangeListener(spinnerListener)
    }

    /** Aggiunge l'ascoltatore di eventi agli spinner */
    def attachSpinnerListener() {
        xEditor.addChangeListener(spinnerListener)
        yEditor.addChangeListener(spinnerListener)
        zEditor.addChangeListener(spinnerListener)
    }

    /** Restituisce il valore corrente dell'editor */
    def getEditedValue() = {
        new NTuple3(xEditor.getValue.asInstanceOf[Number],
            yEditor.getValue.asInstanceOf[Number],
            zEditor.getValue.asInstanceOf[Number])
    }
}
```

E' swing classico in salsa scala. Le modifiche evidenziate rendono `NTuple3Editor` una sorgente di eventi `ChangeEvent`, generati in risposta ad analoghi eventi prodotti dagli spinner. I metodi

`attachSpinnerListener` e `detachSpinnerListener` sono usati all'interno di `setEditedValue` per evitare che gli spinner producano un evento – e di conseguenza lo faccia l'editor – quando il componente `ViewController` assegna programmaticamente un valore ai suoi componenti. Il modulo `ViewController` dichiara un `ChangeListener` e lo aggancia ai due editor:

```
… omissis …
/** Controlli della vista sulla scena 3D */
trait ViewController extends PafModule {

    /** Ascoltatore degli eventi generati dagli editor */
    private val editorChangeListener = new ChangeListener {
        def stateChanged(e: ChangeEvent) = {
            updateEngineViewPosition()
        }
    }

    /** Editor rotazione-posizione */
    private val locationEditor = new NTuple3Editor("Pos X", "Pos Y", "Pos Z").
        addChangeListener(editorChangeListener)
    private val rotationEditor = new NTuple3Editor("Rot X", "Rot Y", "Rot Z").
        addChangeListener(editorChangeListener)

… omissis …

    /** Invocato per aggiornare la posizione della vista nel motore 3d */
    def updateEngineViewPosition() {
        val pos = locationEditor.getEditedValue
        val rot = rotationEditor.getEditedValue
        val view = new ViewElement("default", "0", pos, rot)
        env(classOf[Engine3D]).setCurrentView(view)
    }
```

Nell'implementazione del modulo del motore 3d cambiamo la definizione del metodo `setViewPosition` per tener conto della rappresentazione in gradienti:

```
...jme3pafengine...
    /** Imposta la posizione della vista */
    def setCurrentView(view: ViewElement) = canvas.call{ app =>
        import conversions._

        val pos: Vector3f = view.position
        val rot: Vector3f = view.rotation
        rot.multLocal(FastMath.DEG_TO_RAD)//da gradienti a radianti
        val cam = app.getCamera
        cam.setLocation(pos)
        val quaternion = new Quaternion()
        quaternion.fromAngles(rot.x, rot.y, rot.z)
        cam.setRotation(quaternion)
    }
...jmepafengine
```

E aggiungiamo un cubo “di debug” (se non vediamo un fico secco è difficile verificare che la vista funzioni):

```
...jme3pafengine...
    /** Riceve un messaggio dall'ambiente */
    def parse(message: PafModuleMessage) = {
        import PafModuleMessage._

        message match {
            case START => createDebugBox()
```

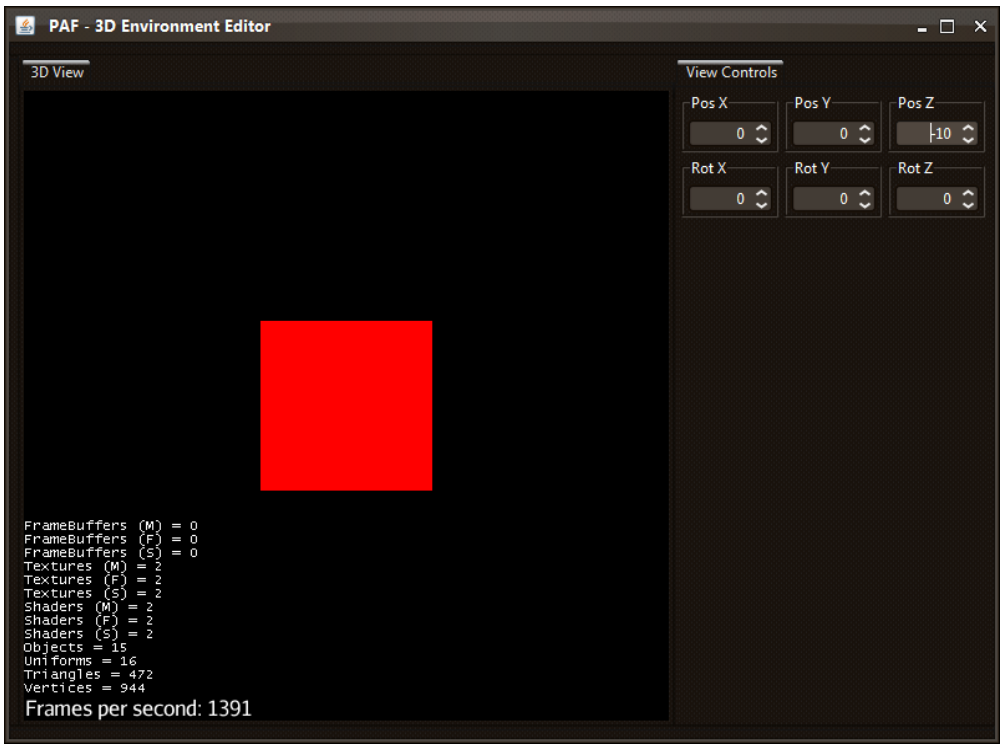
```
            case _ =>
        }
    }

    /** temporaneo, per verificare la posizione della vista*/
    def createDebugBox() = canvas.call { app =>
        val box = new Box(1, 1, 1)
        val material = app.getAssetManager.loadMaterial(
            "Common/Materials/RedColor.j3m")
        val geom = new Geometry("debug box", box)
        geom.setMaterial(material)
        geom.setModelBound(new BoundingBox)
        app.getRootNode.attachChild(geom)
    }
...jme3pafengine...
```

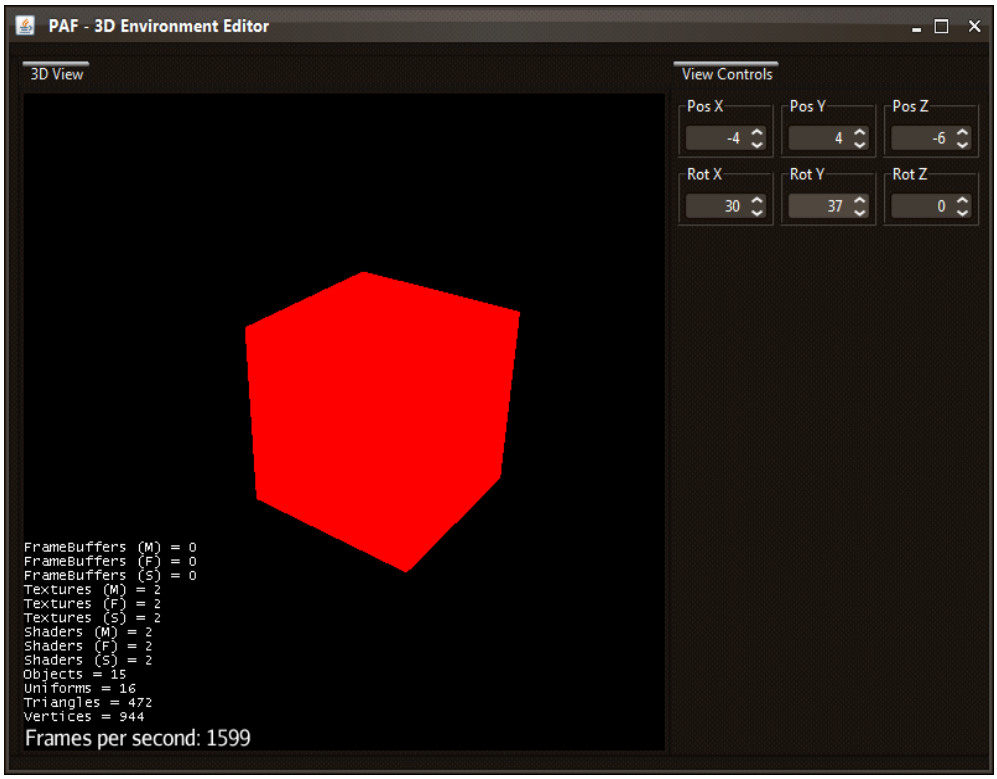
Ultima cosa da fare è impostare la vista predefinita quando il programma si avvia. Il modulo `ViewController` sa che il programma parte quando riceve un messaggio `START`, quindi all'arrivo di quel messaggio impostiamo alcuni valori sull'editor e notifichiamo il cambiamento al modulo del motore grafico:

```
...viewcontroller...
    /** Gestisce i messaggi inviati dall'ambiente a questo modulo */
    def parse(message: PafModuleMessage) = {
        import PafModuleMessage._

        message match {
            case START => setDefaultViewPosition()
            case _ =>
        }
    }

    /** All'avvio imposta la posizione predefinita della vista */
    def setDefaultViewPosition() {
        val pos = new NTuple3(0,0,-10)
        locationEditor.setEditedValue(pos)
        updateEngineViewPosition()
    }
...viewcontroller...
```

In tutto questo implicitamente assumiamo che l'editor 3d lavori nello stesso spazio di coordinate di jme3 (che è un destrorso cioè, prendendo come riferimento l'utente che guarda il monitor, `y` cresce verso l'alto, `x` verso sinistra, `z` verso il monitor): è una decisione di comodo ma di cui tener conto nel caso si voglia passare ad un diverso engine.

Alla fine della fiera, otteniamo il nostro sistema di controllo di base della vista.





Un po' di più.

Messo così il controllo della visuale è chiaramente di una tristezza senza fine. Sbizzarriamoci aggiungendo:

un po' di visuali predefinite tra cui scegliere (dall'alto, dal basso, di qui, di lì eccetera)
la possibilità di creare delle posizioni predefinite e di richiamarle

Passare da una vista a più viste comporta la creazione di qualche nuovo elemento e la modifica di

parte di ciò che è già stato scritto.
Creiamo un modulo che generi gli identificatori univoci degli elementi della scena.

```
package it.tukano.paf.modules

import it.tukano.paf.core.{PafModuleMessage, PafModule}
import java.util.concurrent.atomic.AtomicLong

/** Genera degli uid univoci (per sessione) */
trait UIDGenerator extends PafModule {
    private var lastUid = new AtomicLong(0)

    /** Genera un nuovo uid */
    def generateNewUid() = {
        val uid = lastUid.getAndAdd(1)
        String.format("%016X", uid.asInstanceOf[AnyRef])
    }

    def parse(message: PafModuleMessage) = {}
}
```

Gli uid servono per identificare gli elementi salvati in un file quando entrano in ballo dei riferimenti incrociati e per far capire all'engine quali elementi grafici siano associati alle controparti logiche – cioè quelle gestite dal programma. Il modulo creato è registrato in paflauncher, vi risparmio la tiritera.
Per il resto si tratta di aggiungere qualche componente a `ViewController` e un paio di metodi per gestirli.

```
package it.tukano.paf.modules

import it.tukano.paf.core.{PafModuleMessage, PafModule}
import it.tukano.paf.scene.ViewElement
import it.tukano.paf.math.NTuple3
import javax.swing.event.{ChangeEvent, ChangeListener}
import it.tukano.paf.utils.swing.{ButtonExt, TitledContainer, NTuple3Editor,
    ContainerExt}
import java.awt.{GridLayout, BorderLayout}
import javax.swing.{JOptionPane, JButton, BoxLayout, Box, JComboBox,
    DefaultComboBoxModel, JPanel}
import java.awt.event.{ActionListener, ActionEvent}
/** Controlli della vista sulla scena 3D */
trait ViewController extends PafModule {

    /** Tipo degli elementi mostrati dalla casella combinata delle viste */
    private class ViewBoxItem(view: ViewElement) {
        override def toString = view.name
        val uid = view.uid
    }

    /** Ascoltatore degli eventi generati dagli editor */
    private val editorChangeListener = new ChangeListener {
        def stateChanged(e: ChangeEvent) = {
            updateEngineView()
        }
    }

    /** Editor rotazione-posizione */
    private val locationEditor = new NTuple3Editor("Pos X", "Pos Y", "Pos Z").
        addChangeListener(editorChangeListener)
    private val rotationEditor = new NTuple3Editor("Rot X", "Rot Y", "Rot Z").
        addChangeListener(editorChangeListener)
```

```
/** Editor viste */
private val viewChooserModel = new DefaultComboBoxModel
private val viewChooser = new JComboBox(viewChooserModel)
viewChooser.addActionListener(new ActionListener {
    def actionPerformed(e: ActionEvent) {
        val uid = viewChooser.getSelectedItem.asInstanceOf[ViewBoxItem].uid
        currentView = viewSet(uid)
        locationEditor.setEditedValue(currentView.position)
        rotationEditor.setEditedValue(currentView.rotation)
        env(classOf[Engine3D]).setCurrentView(currentView)
    }
})

private val removeViewButton = (new JButton("Delete") with ButtonExt).
    scaleFont(0.8).tip("Remove the selected view").
    onAction(removeCurrentView)
private val markViewButton = (new JButton("Mark") with ButtonExt).
    scaleFont(0.8).tip("Create a new view").
    onAction(markCurrentView)
private val updateViewButton = (new JButton("Update") with ButtonExt).
    scaleFont(0.8).tip("Update the current view").
    onAction(resetCurrentView)
private val buttonsContainer = (new JPanel with ContainerExt).
    withLayout(new GridLayout(1, 3)).
    append(markViewButton).append(updateViewButton).append(removeViewButton)
private val viewIndexContainer = (new JPanel with ContainerExt).
    withLayout(new BorderLayout).
    append(buttonsContainer -> BorderLayout.NORTH).
    append(viewChooser -> BorderLayout.CENTER)

/** Contenitore degli editor */
private val editorContainer = (new Box(BoxLayout.Y_AXIS) with ContainerExt).
    append(TitledContainer("Available Views", viewIndexContainer)).
    append(locationEditor.component).
    append(rotationEditor.component)

/** Componente dell'interfaccia */
val displayComponent = (new JPanel with ContainerExt).
    withName("View Controls").
    withLayout(new BorderLayout).
    append(editorContainer -> BorderLayout.NORTH).
    withPreferredSize(250, 200)

/** Insieme delle viste */
private var viewSet = Map[String, ViewElement]()

/** Vista corrente */
private var currentView: ViewElement = _

/** Rimuove la vista selezionata nell'editor */
private def removeCurrentView() {
    if(viewChooserModel.getSize > 1) {
        viewChooserModel.removeElementAt(viewChooser.getSelectedIndex)
    }
}

/** Reimposta i valori della vista selezionata nell'editor */
private def resetCurrentView() {
    val pos = locationEditor.getEditedValue
    val rot = rotationEditor.getEditedValue
    currentView = currentView.set(pos, rot)
    registerView(currentView)//aggiorna la mappa interna
}
```

```
/** Crea una nuova vista con i dati dell'editor */
private def markCurrentView() {
    val frame = JOptionPane.getFrameForComponent(displayComponent)
    val name = JOptionPane.showInputDialog(frame,
        "Insert a name for the view")
    if(name != null && !name.isEmpty) {
        val uid = env(classOf[UIDGenerator]).generateNewUid
        val pos = locationEditor.getEditedValue
        val rot = rotationEditor.getEditedValue
        val view = ViewElement(name, uid, pos, rot)
        registerView(view)
        viewChooserModel.insertElementAt(new ViewBoxItem(view), 0)
        viewChooser.setSelectedIndex(0)
    }
}

/** Gestisce i messaggi inviati dall'ambiente a questo modulo */
def parse(message: PafModuleMessage) = {
    import PafModuleMessage._

    message match {
        case START => createDefaultViewPosition()
        case _ =>
    }
}

/** All'avvio imposta la posizione predefinita della vista */
private def createDefaultViewPosition() {
    val uidgen = env(classOf[UIDGenerator])
    val pos = new NTuple3(0,0,-10)
    val rot = new NTuple3(0,0,0)
    val id = uidgen.generateNewUid
    val frontView = new ViewElement("Front", id, pos, rot)
    registerView(frontView)
    currentView = frontView
    viewChooserModel.addElement(new ViewBoxItem(frontView))
    locationEditor.setEditedValue(pos)
    rotationEditor.setEditedValue(rot)
    env(classOf[Engine3D]).setCurrentView(frontView)

    //crea le altre viste predefinite
    val topView = ViewElement("Top", uidgen.generateNewUid,
        NTuple3(0, 10, 0), NTuple3(90, 0, 0))
    val bottomView = ViewElement("Bottom", uidgen.generateNewUid,
        NTuple3(0, -10, 0), NTuple3(-90, 0, 0))
    val rightView = ViewElement("Right", uidgen.generateNewUid,
        NTuple3(-10, 0, 0), NTuple3(0, 90, 0))
    val leftView = ViewElement("Left", uidgen.generateNewUid,
        NTuple3(10, 0, 0), NTuple3(0, -90, 0))
    val backView = ViewElement("Back", uidgen.generateNewUid,
        NTuple3(0, 0, 10), NTuple3(0, -180, 0))
    List(topView, bottomView, rightView, leftView, backView).foreach{ e =>
        registerView(e)
        viewChooserModel.addElement(new ViewBoxItem(e))
    }
}

/** Registra o aggiorna l'elemento e nella mappa delle viste */
private def registerView(e: ViewElement) {
    viewSet = viewSet + (e.uid -> e)
}

/** Applica la vista corrente al motore grafico */
```

```
    private def updateEngineView() {
        val view = currentView.set(locationEditor.getEditedValue,
            rotationEditor.getEditedValue)
        env(classOf[Engine3D]).setCurrentView(view)
    }
}
```

Risulta (con una skin diversa, Graphite) l'interfaccia che segue: