Scala tutorial pratico

Parte 5

Introduzione

Ri-partiamo con la creazione di un'applicazione del genere “oh che bello tanto lo so che non la finisco”, quelle robe destinate ad accumulare decine di migliaia di righe di codice scritte da noi, più un'altra milionata in librerie. Scala è interessante in questo contesto perchè tra tipi strutturali, conversioni implicite e condivisione dello spazio dei nomi permette una certa libertà nella manipolazione e nel collegamento di strutture già esistenti. Si può, in altri termini, buttare il cuore oltre l'ostacolo avendo ottime probabilità di recuperarlo nel caso in cui il corpo non riesca a seguirlo – un po' splatter come metafora?

Il programma.

Il programma che scriviamo è un editor 3D, nel senso di creazione di geometrie, materiali, animazioni, logica, insomma, tutto quello che può servire a creare un'ambiente tridimensionale dinamico e interattivo. Ovviamente le possibilità di portare a termine, da soli, un mattone del genere sono a dir poco remote, per scarsità dell'unica risorsa realmente scarsa: il tempo. Ciò nonostante, lanciarsi in questo genere di esercizi kamikaze ha un'utilità non trascurabile perchè la programmazione è forse una delle poche arti in cui si possono tentare imprese disperate senza che ciò comporti una virgola di disagio: non ci sono morti o feriti, non si buttano via soldi, ci si diverte e basta.

Struttura di base.

Il programma è diviso in tre pezzi: una libreria di base che contiene i moduli astratti, le implementazioni predefinite e i tipi di dato usati per la comunicazione tra i componenti dell'applicazione, una libreria che fornisce l'implementazione di base del motore 3D e un “lanciatore”, cioè un programma con una classe main che combina i moduli e avvia il programma. Il motore 3D è a piacere, nel senso che l'applicazione astrae il rendering attuale delegandolo ad un modulo esterno- per il test useremo le api JMonkeyEngine3 ma va bene uno qualsiasi.
Poiché non conosciamo il numero di moduli che comporranno l'applicazione né le loro caratteristiche concrete, stabiliamo un paio di forme di base che dicano tutto o niente.

```
package it.tukano.paf.core

object PafModuleMessage {

    val INIT = new PafModuleMessage
    val START = new PafModuleMessage
    val STOP = new PafModuleMessage
    val DESTROY = new PafModuleMessage
}

class PafModuleMessage {
}
```

Quattro messaggi predefiniti che definiscono il ciclo vitale di un modulo. Paf è il nome del programma.

```
package it.tukano.paf.core;
```

```
trait PafModule {

    protected var env: PafEnvironment = _

    def setEnvironment(env: PafEnvironment) {
        this.env = env
    }

    def parse(message: PafModuleMessage)
}
```

Il tipo dei moduli del programma. Qui c'è la dichiarazione del campo env con valore iniziale _:

```
    protected var env: PafEnvironment = _
```

Significa che il valore iniziale di `env` è il valore predefinito per le variabili del tipo `PafEnvironment`. E' `null`. L'uso di `null` è contrario al bon-ton di Scala, anche nella forma dell'inizializzazione di campo qui proposta. D'altronde, essendo io l'unico utente della libreria ed essendo env inizializzato in un ciclo, posso tranquillamente dare per scontato che, rispettando le precondizioni dettate dal ciclo vitale, non inconterò la mitica `NullPointerException`. Il tutto per risparmiarmi la noia di dover scrivere: `option.get`, `if(option.isDefined)` mille volte.

```
package it.tukano.paf.core

import collection.mutable.ListBuffer

class PafEnvironment {

    private var modules = ListBuffer[PafModule]()

    def register(m: PafModule*) {
        modules.appendAll(m)
    }

    def apply[T <: PafModule](c: Class[T]): T = {
        val module = modules.find(e => c.isAssignableFrom(e.getClass)).get
        c.cast(module)
    }

    def start() {
        import PafModuleMessage._

        modules.foreach(_.setEnvironment(this))
        modules.foreach(_.parse(INIT))
        modules.foreach(_.parse(START))
    }

    def stop() {
        import PafModuleMessage._

        modules.foreach(_.parse(STOP))
        modules.foreach(_.parse(DESTROY))
    }
}
```

Il registro. Una cornucopia di novità. Partiamo dal metodo register:

```
def register(m: PafModule*)
```

L'asterisco significa che `m` è una sequenza di zero, uno o più valori di tipo `PafModule`. Un vararg per

farla breve.
Il metodo `apply` cerca un modulo compatibile per tipo e lo restituisce, se c'è. Da notare che il metodo `apply` restituisce `null` nel caso in cui non si trovi un risultato: anche qui, è più scalesco dire `Option[T]` e restituire `None` ma, come per il campo `env` di prima, posso garantire a me stesso che non incapperò mai in un `null`. Essendo un `apply` potrà essere invocato nella forma istanza(), come fosse una sorta di metodo o di mappa.
I metodi `start` e `stop` hanno di particolare la dichiarazione import.

```
        import PafModuleMessage._
```

L'import appare in un blocco e funziona come un import usuale salvo il fatto che (in verità non è un eccezione ma la regola) esso è limitato all'ambito in cui compare, vale a dire nel corpo dei metodi `start` e `stop`. E' comodo, nel caso in cui ci si confronti con dei nomi iperusati (tipo `Node` o `Document` o `Message` eccetera), poter limitare al minimo indispensabile l'ambito di una dichiarazione di importazione. E' comodo anche quando si vogliano usare dei nomi dichiarati in un tipo – ed è la ragione per cui l'ho usato – poiché essendo l'inclusione visivamente prossima all'uso dei membri inclusi il significato dei nomi di quei membri è più agevolmente riconducibile al tipo in cui sono dichiarati. La stessa dichiarazione, in termini più generali, ha anche la particolarità di essere relativa. `PafModuleMessage` appartiene al package `it.tukano.paf.core`, dunque la sua inclusione pienamente qualificata è:

```
import it.tukano.paf.core.PafModuleMessage
```

Tuttavia, a differenza di Java, in Scala esiste una relazione gerarchica inclusiva tra i package. Significa che `it` è un package al cui interno è dichiarato `tukano`, al cui interno è dichiarato `paf`, dentro a cui troviamo `core` e infine `PafModuleMessage`. La dichiarazione import tiene conto di questa gerarchia così che se l'unità di compilazione – o meglio l'ambito – in cui mi trovo appartiene al package `x` allora la dichiarazione di importazione può far riferimenti ai membri di `x` e ai sottopackage di `x` usando il loro nome "semplice" (cioè senza la parte che qualifica l'appartenenza al package in cui già mi trovo).
L'ultimo pezzo della tabula rasa su cui lavoriamo è il lanciatore. Per il lanciatore creo un progetto a parte – IDEA non sembra digerire tanto bene i progetti multi-modulo con Scala, tende a dimenticarsi quali classi abbia sotto il naso – che dipende dalle classi di paf, Substance, JMonkeyEngine3. Il progetto contiene una sola classe:

```
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}

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(/*qui ci andranno le istanze dei moduli*/)
        env.start()
    }
}
```

Uso un progetto a parte perchè non posso usare un modulo dell'IDE – che sarebbe lo strumento ideale – e perchè non voglio che paf dipenda dalle api di implementazione del motore 3d.
Da notare che il thread predefinito del programma è l'edt. Ci sono delle interessanti questioni di threading che saltano fuori nell'uso di alcuni tra i più diffusi engine 3d per java (Ardor3D, JMonkeyEngine, Xith per dirne 3) o con le api Jogl.

Modulo 1, la finestra principale.

Creiamo anche un packagino a parte in cui definiamo alcune utilità del pigrone. Occhio al magheggio.

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

import java.awt.{Container, LayoutManager, Component}

trait ContainerExt extends Container {

    /** Imposta un layout e restituisce this */
    def withLayout(layout: LayoutManager) = {
        setLayout(layout)
        this
    }

    /** Aggiunge un set di coppie componente-posizione */
    def append(elements: (Component, AnyRef)*) = {
        elements.foreach(e => add(e._1, e._2))
        this
    }
}
```

Qui `ContainerExt` è un `trait` che ha tutti i metodi e i campi di `Container` più due aggiunte. I due metodi in più fanno cose note (uno imposta un layout, l'altro aggiunge dei componenti forniti come serie di coppie componente-posizione nel layout) e restituiscono `this`, così si possono concatenare le invocazioni. Lo scopo di questo trait è essere combinato con un contenitore, nella forma:

```
val panel = new JPanel with ContainerExt
```

Quel `with` combina il mio `ContainerExt` con un'istanza di `JPanel` e produce un `ContainerExt` che usa quel `JPanel`. L'uso a cui faccio riferimento è evidenziato in giallo nel codice del trait `ContainerExt`. A che pro 'sto casotto? `ContainerExt` permette di concatenare le invocazioni dei metodi di un contenitore, quindi posso dire, ad esempio:

```
val panel = (new JPanel with ContainerExt).
      withLayout(new BorderLayout).
      add(componente -> BordelLayout.WEST, componente -> BorderLayout.CENTER)
```

Si possono mescolare tutti i trait che vogliamo. Da notare che mentre in:

```
val panel = new JPanel with ContainerExt
```

o (è la stessa cosa):

```
val panel = (new JPanel with ContainerExt)
```

panel è di tipo `JPanel` E `ContainerExt`, per com'è fatta la concatenazione in `ContainerExt`, dicendo:

```
val panel = (new JPanel with ContainerExt).append(x -> y).append(z -> w)
```

panel non può che essere un `ContainerExt`, perchè i metodi append restituiscono `this`. Teniamolo a mente, a breve vediamo un altro truccone.
Infine un cenno sulla forma:

```
x -> y
```

L'operatore `->` genera una tupla (in questo caso una coppia) contenente gli elementi `x` e `y`.
Ora aggiungo un'altro `trait` “decorativo” al programma:

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

import java.awt.event.{WindowAdapter, WindowEvent}
import java.awt.{Window}

/** Una finestra con un paio di metodi in più */
trait WindowExt extends Window {

    /** invoca la funzione in argomento quando si chiude la finestra */
    def onClosing(fun: () => Unit) = {
        addWindowListener(new WindowAdapter {
            override def windowClosing(e: WindowEvent) = {
                fun()
            }
        })
        this
    }

    /** Assegna una dimensione alla finestra */
    def withSize(w: Int, h: Int) = {
        setSize(w, h)
        this
    }
}
```

Anche qui nient'altro che una concatenazione potenziale più uno shortcut per usare una funzione come fosse un `WindowListener` in attesa per un evento `WindowClosing`.
Posso combinare i due `trait`? Sì. E' qui che viene il bello. Posso cioè dire:

```
val panel = new JPanel with ContainerExt
val window = new JFrame with WindowExt with ContainerExt
```

Il primo valore è di tipo `JPanel` e `ContainerExt`. Il secondo è tipo `JFrame` e `WindowExt` e ContainerExt. Però c'è un problema: `window` è sia un `WindowExt` che un `ContainerExt`, oltre che un `JFrame`. Siccome i metodi di `WindowExt` e di `ContainerExt` sono fatti apposta per essere concatenati, uno vorrebbe dire:

```
window.metodoDiWindowExt.metodoDiContainerExt.metodoDiWindowExt
```

Ma se andiamo a vedere i metodi di `WindowExt` e `ContainerExt`, essi restituiscono `this` e il tipo in compilazione di `this` è `WindowExt` e `ContainerExt`, rispettivamente. Quind se su `window` invoco un metodo di `WindowExt` alla concatenazione successiva mi trovo per le mani un valore di tipo `WindowExt` e non posso chiaramente invocare su di esso un metodo di `ContainerExt`, anche se il riferimento in esecuzione lo ammetterebbe. Bene, il rimedio sta nello specificare che il tipo restituito dai metodi di `WindowExt` e `ContainerExt` è:

```
this.type
```

Questo consente al compilatore di inferire che l'invocazione avviene sul tipo di `window` e di mantenere il tipo durante la concatenazione. Modifico quindi sia `ContainerExt` che `WindowExt`:

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

import java.awt.{Container, LayoutManager, Component}

trait ContainerExt extends Container {

    /** Imposta un layout e restituisce this */
    def withLayout(layout: LayoutManager): this.type = {
        setLayout(layout)
        this
    }

    /** Aggiunge un set di coppie componente-posizione */
    def append(elements: (Component, AnyRef)*): this.type = {
        elements.foreach(e => add(e._1, e._2))
        this
    }
}

package it.tukano.paf.utils.swing

import java.awt.event.{WindowAdapter, WindowEvent}
import java.awt.{Window}

/** Una finestra con un paio di metodi in più */
trait WindowExt extends Window {

    /** invoca la funzione in argomento quando si chiude la finestra */
    def onClosing(fun: () => Unit): this.type = {
        addWindowListener(new WindowAdapter {
            override def windowClosing(e: WindowEvent) = {
                fun()
            }
        })
        this
    }

    /** Assegna una dimensione alla finestra */
    def withSize(w: Int, h: Int): this.type = {
        setSize(w, h)
        this
    }
}
```

Specificando `this.type` come tipo restituito posso “saltare da un tipo all'altro” nella concatenazione senza problemi.
E via col primo modulo (finalmente).

```
package it.tukano.paf.modules

import it.tukano.paf.core.PafModule
import util.logging.Loggerd

trait MainFrame extends PafModule with Logged {

}
```

Quindi `MainFrame` è un `trait` (tanto per cambiare) combinato con `PafModule` e `Logged`. `Logged` da al tipo `MainFrame` la capacità di sparare messaggi di log. Lo fa dotando il tipo di un metodo `log(String)`. `PafModule` è il nostro modulo base. Perchè uso un `trait` per `MainFrame`? Perchè un `trait` Scala ha proprietà più ampie di una classe Scala e io sto scrivendo un programma potenzialmente grande senza aver speso un minuto che sia uno nel ragionarci su: cerco insomma di tenere la coperta più lunga che posso. Come farò poi a istanziare il mio `trait MainFrame` quando ne avrò bisogno? Sempre con la composizione:

```
val istanza = new Object with MainFrame
```

In alternativa potrei anche dire:

```
val istanza = new MainFrame() {}
```

Cioè usare la forma della classe anonima ma non è idiomatico.
Orbene, che fa `MainFrame`. Fa la finestra principale, divisa nelle cinque porzioni classiche del `BorderLayout`, ognuna delle quali contiene un pannello a schede. I campi di `MainFrame` sono:

```
package it.tukano.paf.modules

import it.tukano.paf.core.{PafModuleMessage, PafModule}
import java.awt.BorderLayout
import util.logging.Logged
import it.tukano.paf.utils.swing.{ContainerExt, WindowExt}
import javax.swing.{JFrame, JPanel, JTabbedPane}

/** Finestra principale del programma */
trait MainFrame extends PafModule with Logged {

    private val westTab = new JTabbedPane()
    private val centerTab = new JTabbedPane()
    private val southTab = new JTabbedPane()
    private val eastTab = new JTabbedPane()
    private val northTab = new JTabbedPane()
    private val mainContainer = (new JPanel() with ContainerExt).
        withLayout(new BorderLayout()).
        append(westTab -> BorderLayout.WEST,
            eastTab -> BorderLayout.EAST,
            centerTab -> BorderLayout.CENTER,
            northTab -> BorderLayout.NORTH,
            southTab -> BorderLayout.SOUTH)

    private val window = (new JFrame() with WindowExt with ContainerExt).
        onClosing(closeApplication).
        withLayout(new BorderLayout).
        withSize(800, 600).
        append(mainContainer -> BorderLayout.CENTER)

    /** Invocato alla chiusura dell'applicazione */
    private def closeApplication() {
        env.stop()
    }

}
```

La finestra contiene il pannello principale che contiene a sua volta cinque contenitori a schede. Tramite `onClosing` colleghiamo la chiusura della finestra all'invocazione del metodo `closeApplication`. Il metodo `closeApplication` invoca `stop` su `env` – cioè sul gruppo di moduli

a cui appartiene anche la nostra finestra principale. Cosa succede quando invochiamo `stop()` su un `PafEnvironment`? L'ambiente reagisce invocando prima `stop` e poi `destroy` per ogni altro modulo, inclusa la nostra finestra. Nella stessa finestra intercetto il messaggio `destroy` e lo uso per seccare tutto il programma:

```
package it.tukano.paf.modules

import it.tukano.paf.core.{PafModuleMessage, PafModule}
import java.awt.BorderLayout
import util.logging.Logged
import it.tukano.paf.utils.swing.{ContainerExt, WindowExt}
import javax.swing.{JFrame, JPanel, JTabbedPane}

/** Finestra principale del programma */
trait MainFrame extends PafModule with Logged {

...omissis

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

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

    /** da definire */
    private def destroyWindow() { }

}
```

Lì `match` è una parola chiave che in scala denota il pattern matching. La forma è:

```
valore = espressione match {
      case pattern1 => espressione1
      case pattern2 => espressione2
      ...
      case patternN => espressioneN
}
```

Nel caso in questione non ci interessa il valore restituito comunque vale la pena di saperlo: volendo c'è. I pattern possibili sono una varietà ma in generale si possono verificare valori o tipi. Nel nostro caso verifichiamo se il valore di message equivale a quello del campo `DESTROY` di `PafModuleMessage`. Se così è, invochiamo il metodo `destroyWindow`. Chiudiamo il match con quel “caso”:

```
_ =>
```

Il simbolo di sottolineatura denota “in qualsiasi altro caso”. In Scala il pattern matching deve essere esaustivo, cioè deve sempre coprire tutti i possibili casi, altrimenti è rilasciata un'eccezione in esecuzione. Esiste anche la possibilità di far verificare al compilatore che un match copra tutti i valori possibili quando il tipo dell'espressione è un `case class` ma è un altro paio di maniche. Dunque quando il messaggio è `DESTROY` passiamo al metodo `destroyWindow`, sempre in `MainFrame`. Il metodo `destroyWindow` chiude la finestra e poi commette un crimine:

```
    /** Chiude la finestra e avvia un demone che ferma la jvm dopo 5 secondi */
```

```
    private def destroyWindow() {
        window.dispose()
        val vmkiller = new Thread {

            override def run = try {
                Thread.sleep(5000)
                System.exit(0)
            } catch {
                case ex: Throwable => log(ex.toString)
            }
        }
        vmkiller.setDaemon(true)
        vmkiller.start
    }
```

Dopo aver chiuso la finestra, parte un thread demone, cioè un thread la cui esecuzione non impedisce lo spegnimento della jvm. Il thread attende cinque secondi e, nel caso in cui la jvm sia sopravvissuta a quel termine, forza lo spegnimento del programma. La ragione sta in ciò che esistono almeno un paio di engine 3d Java (tra cui quello che useremo) che hanno il vizietto di creare dei thread virtualmente insopprimibili. Senza le maniere forti c'è il rischio che resti in vita una jvm per ogni sessione di esecuzione del programma, il che è una schifezza.
In definitiva, abbiamo una finestra principale su cui lavorare, qui riepilogata:

```
package it.tukano.paf.modules

import it.tukano.paf.core.{PafModuleMessage, PafModule}
import java.awt.BorderLayout
import util.logging.Logged
import it.tukano.paf.utils.swing.{ContainerExt, WindowExt}
import javax.swing.{JFrame, JPanel, JTabbedPane}

/** Finestra principale del programma */
trait MainFrame extends PafModule with Logged {

    private val westTab = new JTabbedPane()
    private val centerTab = new JTabbedPane()
    private val southTab = new JTabbedPane()
    private val eastTab = new JTabbedPane()
    private val northTab = new JTabbedPane()
    private val mainContainer = (new JPanel() with ContainerExt).
        withLayout(new BorderLayout()).
        append(westTab -> BorderLayout.WEST,
            eastTab -> BorderLayout.EAST,
            centerTab -> BorderLayout.CENTER,
            northTab -> BorderLayout.NORTH,
            southTab -> BorderLayout.SOUTH)

    private val window = (new JFrame() with WindowExt with ContainerExt).
        onClosing(closeApplication).
        withLayout(new BorderLayout).
        withSize(800, 600).
        append(mainContainer -> BorderLayout.CENTER)

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

        message match {
            case START => window.setVisible(true)
            case DESTROY => destroyWindow()
            case _ =>
        }
```

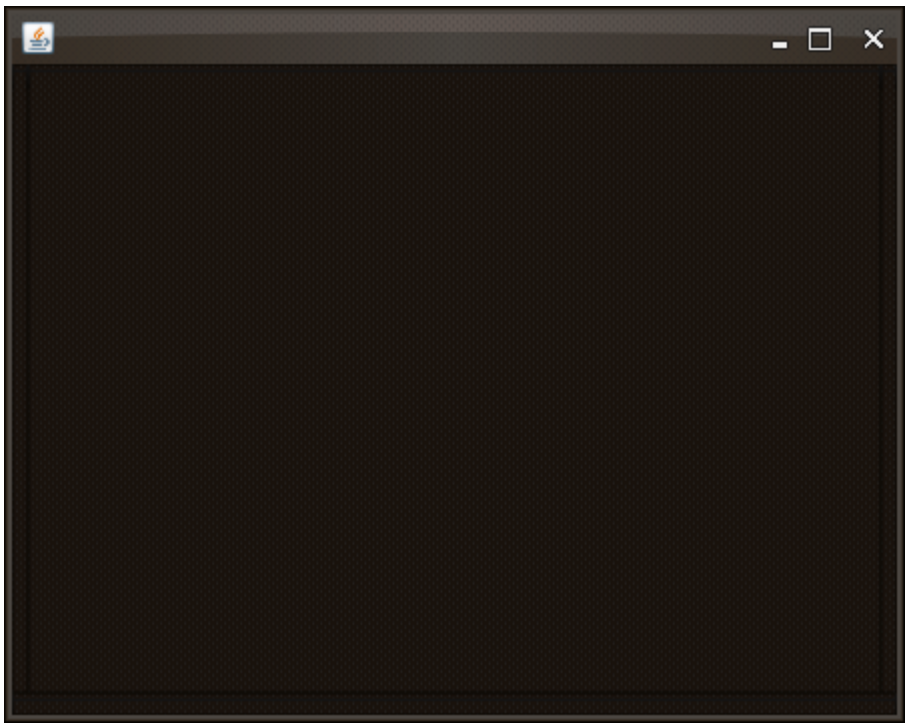
```
    }

    /** Invocato alla chiusura dell'applicazione */
    private def closeApplication() {
        env.stop()
    }

    /** Chiude la finestra e avvia un demone che ferma la jvm dopo 5 secondi */
    private def destroyWindow() {
        window.dispose()
        val vmkiller = new Thread {

            override def run = try {
                Thread.sleep(5000)
                System.exit(0)
            } catch {
                case ex: Throwable => log(ex.toString)
            }
        }
        vmkiller.setDaemon(true)
        vmkiller.start
    }
}
```

Inserimento del modulo MainFrame nell'applicazione di avvio.

Giochiamo la carta `MainFrame` nell'applicazione separata che avvia il nostro editor.

```
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

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)
        env.start()
    }
}
```

Substance è l'API di un LookAndFeel Swing. Avviando il programma che contiene il main (che dipende dal progetto paf e dai jar di Substance ma non sto qui ad annoiarvi), compare la nostra finestrella.



E qui chiudiamo. Al prossimo giro creiamo il modulo che gestisce il motore 3d e lo ficchiamo nella finestra.