

PSTL – FRP et voyage dans le temps

Guillaume Hivert, Jordi Bertran de Balanda

25 Mai 2016

Contents

1	Introduction	2
1.1	Sujet original	2
1.2	Outils utilisés	2
2	Ambitions	3
2.1	Functional Reactive Programming	3
2.2	Gestion du modèle	4
2.3	Live Coding	4
2.4	Abstraction graphique	5
2.5	Modifications graphiques minimales	5
3	Questions rencontrées	6
3.1	Signaux & Callback Hell	6
3.2	Gestion du modèle	7
4	Embla	8
4.1	Vue d'ensemble	8
4.1.1	Architecture	8
4.1.2	Exécution	13
5	Extensions	14
5.1	Acyclisme du graphe de signaux	14
5.2	Hierarchie des zones de dessin	14
5.3	Voyage dans le temps	14
6	Bibliographie	15

1 Introduction

Ce rapport présente le travail effectué pour l’UE projet PSTL (4I508), réalisé sous la direction de Frédéric Peschanski.

1.1 Sujet original

Le titre du sujet de départ du projet était “FRP et voyage dans le temps”. Ce projet avait pour but d’explorer l’approche de programmation Functional Reactive Programming. Au fil du déroulement du projet, nous avons été amenés à privilégier certains aspects du projet au profit d’autres.

Pour rappel, les tâches du projet étaient les suivantes:

- L’intégration de signaux de premier ordre, avec les combinateurs adéquats pour fournir à l’utilisateur un large panel de possibilités quant à la description des comportements
- La construction d’un modèle basé sur des structures immutables
- Les mises à jour construisant les vues à partir du modèle, gérées de façon à émuler l’approche d’ELM et de React.

1.2 Outils utilisés

- Clojure
- Leiningen
- Boot
- Cider
- GLFW
- LWJGL
- PNGDecoder
- Eclipse/Emacs/Atom
- Slack
- GitHub: <https://github.com/jbertran/embla>

2 Ambitions

Pour explorer le sujet, nous avons décidé de nous donner comme but d’arriver à fournir un moteur de rendu qui mette en jeu autant de concepts évoqués ci-après que possible.

2.1 Functional Reactive Programming

Le terme Functional Reactive Programming (FRP ci-après) décrit un paradigme de programmation qui a pour intention d’offrir une manière déclarative de créer des systèmes réactifs. Lorsqu’on parle de systèmes réactifs dans ce cadre, on vise plus particulièrement les interfaces utilisateur graphiques (GUI) représentant une scène évoluant en fonction d’entrées provenant du monde extérieur.

Il est important de noter l’aspect *déclaratif* du modèle offert par le paradigme FRP dont ELM, l’exemple sur lequel se base notre approche, fait partie. Bien que les frameworks pour GUI usuels soient déclaratifs dans le sens général, la définition d’un nouvel élément à afficher à l’écran nécessite que soient décrites de manière liées la façon dont l’élément interagit avec le reste de la scène.

L’idée de la FRP est de proposer une manière de déclarer les comportements qui permettent de voir les valeurs mutables - positions, scores... - plutôt comme des valeurs *variant dans le temps*, ce qui est particulièrement adapté pour l’interface graphique appliquée au jeu vidéo. Ceci permet de travailler de manière beaucoup plus aisée et logique avec des langages fonctionnels “purs”, où le concept de valeur mutable est généralement proscrit.

Dans notre projet, nous ne limitons pas formellement à la FRP dite classique, qui était à l’origine destinée à l’animation, mais elle montre parfaitement le changement de paradigme entre la manière déclarative usuelle de coder une animation, et la manière proposée par la FRP en général. On dégage de cette manière, dans la FRP classique, deux types de valeurs: les comportements et les événements.

Les *comportements* servent de remplacement pour valeurs mutables auxquelles un utilisateur d’une librairie de haut niveau actuelle (Qt, Swing...) est habitué.

```
Behavior 'a = Time -> 'a
```

Les *événements* représentent, eux, une suite d’événements discrets avec (éventuellement) une information sur le temps auquel ils se déclenchent.

```
Event 'a = [(Time, 'a)]
```

On peut, à partir de ces deux types de valeurs, être très expressif quand aux comportements des objets décrits par ces signaux.

2.2 Gestion du modèle

Afin de pouvoir utiliser un modèle FRP, il est nécessaire de créer un monde de signaux de premier ordre. Ces signaux représentent l'intégralité des événements de l'application.

Dans un monde impératif, les signaux produisent une information — information reçue par des fonctions abonnées à ces signaux. Ces fonctions vont alors agir en conséquence sur le modèle par une suite d'effet de bord pour modifier l'état du modèle. Le modèle change constamment.

A l'inverse, dans un monde purement fonctionnels, le modèle ne doit pas pouvoir être modifié. Chaque fonction, lors de son exécution, peut accéder au modèle, mais elle ne peut pas le modifier. Chaque fonction va alors recréer un modèle complet, correspondant au nouvel état du modèle. Chaque modèle est donc immuable, et il est possible de parcourir les différents états de ceux-ci.

Dans un jeu vidéo, le modèle représente le monde en lui-même ; et les signaux, les différents événements ayant lieu lors du déroulement du jeu. Lorsque le joueur appuie sur une flèche du clavier pour faire avancer son personnage, le signal correspondant émet l'information correspondante. Les fonctions abonnées à ce signal vont alors recréer un nouvel état de jeu — à l'aide d'un nouveau modèle — correspondant à ce qui se déroule : en l'occurrence, l'avancée du personnage. On disposera alors de deux modèles distincts, l'un représentant le monde au temps t , le second au temps $t+1$. Il est alors possible d'effectuer un “voyage dans le temps”.

2.3 Live Coding

Dans un souci de simplicité, et de coller à l'esprit Clojure, l'un des buts de ce projet était également de fournir une interface interactive, type REPL (Read-Eval-Print-Loop), afin de pouvoir développer dynamiquement. Dans un paradigme de développement classique, l'utilisateur écrit son programme à l'aide d'un éditeur de texte ou d'un autre outil, puis le compile et l'exécute (ou l'interprète immédiatement). La compilation peut alors relever des bogues, ainsi que l'exécution. Le développeur retourne alors à son éditeur de texte pour déboguer son programme. Dans un paradigme de live coding, l'utilisateur est amené à écrire son programme dans un éditeur de texte, puis l'exécuter immédiatement. Le code écrit va ainsi être exécuté à la volée, et les conséquences sont immédiatement visibles. Il n'y a plus de séparation entre la phase d'écriture et la phase d'exécution : les deux sont intimement liées. La mise en place d'un REPL permet d'abonder en ce sens : une fois le programme lancé, une boucle d'interaction s'affiche, permettant de rentrer des commandes et de continuer à développer le programme, même si celui-ci est encore en train de fonctionner. Ce concept ressemble fortement au débogueur inclus par défaut dans la majorité des distributions de Common

Lisp, capable d'interrompre le programme au premier bogue pour réécrire le code dynamiquement.

2.4 Abstraction graphique

L'ambition principale étant de fournir un moteur de rendu *utilisable*, une des contraintes est de fournir un résultat dont l'emploi soit facile. On désire donc éviter de conserver le niveau d'abstraction que fournit OpenGL, c'est-à-dire de réaliser une surcouche relativement épaisse d'OpenGL qui nous permette d'ignorer les difficultés de gestion de la machine à états (définition des buffers, passage de flottants correctement ordonnés à la carte graphique, modes de dessin correspondants, etc...) pour fournir un mode de fonctionnement plus immédiatement abordable, avec des formes simples et des mécanismes d'affichage simplifiés à l'extrême. On s'établit donc comme but d'abstraction graphique d'émuler une partie du comportement d'une bibliothèque relativement bas niveau comme la SDL.

Ce parti pris est pour nous un compromis raisonnable entre la trop grande complexité d'OpenGL pour l'utilisateur final, et le trop grand nombre de parti pris sur l'approche d'affichage et de gestion d'une bibliothèque de GUI haut niveau comme par exemple Qt ou Swing. Il faut noter qu'en ce sens, nous voulons éviter au maximum d'imposer à l'utilisateur quoi que ce soit pour le comportement des éléments qu'il veut afficher à l'écran, de manière à lui laisser le soin de lui-même décrire les comportements qu'il désire. Ceci lui laisse une charge plus importante que dans un moteur plus haut niveau, mais lui donne également une plus grande liberté d'action.

2.5 Modifications graphiques minimales

Dans un souci de performances, il se révèle plus intéressant de placer les données graphiques dans la mémoire du GPU au démarrage du programme (ou d'un niveau par exemple, dans le cas d'un jeu), puis de ne plus avoir à y toucher : on minimise l'utilisation du bus mémoire pour faire transiter des données pouvant rapidement atteindre plusieurs centaines de Mio dans le cas de jeux haute définition ; et on réutilise un maximum les données en place dans la mémoire. De plus, les jeux réutilisent souvent les mêmes textures et les mêmes objets (un ennemi peut apparaître plusieurs fois, idem pour les arbres et autres éléments du décor).

Pour faciliter cela, on souhaite donc charger et modifier des éléments le moins souvent possible au niveau de la carte graphique. Or, les différents modèles immutables contiennent tous l'information complète de la scène de jeu. Pour s'en abstraire, il a été envisagé d'effectuer un diff, à l'instar de React. Entre deux modèles, il est possible de comparer les différences entre eux, et en retenir uniquement l'information utile : ce qui a changé. Ce qui n'a pas changé n'a

nul besoin d'être modifié, et ce qui a été changé va être modifié sur la carte graphique. L'état des objets qui ont été modifié est donc la seule information réellement utile.

3 Questions rencontrées

3.1 Signaux & Callback Hell

La création d'un monde de signaux de premier ordre amène divers problèmes inhérents à la plateforme utilisée. Dans un programme Clojure, les signaux sont représentés sous formes de `channel`. Un canal supporte un nombre indéfinis d'écrivains et de lecteurs, mais également de lectures ou d'écritures. Ils fonctionnent comme une file d'attente : lorsqu'un écrivain écrit une valeur à l'intérieur de celui-ci, la valeur se place en attente. Dès qu'une valeur est en attente, si un lecteur peut la lire, il va alors la consommer, et la faire disparaître. Dans un monde de signaux, il serait souhaitable que lorsqu'un signal important, comme le mouvement par exemple, émette une information, toutes les fonctions de notre choix puisse intercepter cette information pour l'utiliser. Toutefois, au vu du modèle, la première fonction lisant la valeur privera les autres fonctions de celle-ci. Il faut donc une couche relai pour faire le lien entre la valeur du signal émis, et la réception de celle-ci par toutes les fonctions.

Pour arriver à ce résultat, il faut donc associer, à chaque signal, une liste de signaux récepteurs. A chaque émission d'une information sur le signal, cette même information est dupliquée dans les signaux récepteurs :

```
(defn broadcast-all
  "Transfer the value from the channel to all functions which need it."
  []
  (loop [[sig & signals] custom-signals]
    (let [input (second sig)
          output (second (rest sig))]
      (go-loop []
        (let [msg (<! input)]
          (map #(go (>! % msg)) output))
          (recur))
        (if-not (empty? signals)
          (recur signals))))))
```

Toutefois, pour qu'une fonction puisse s'abonner au signal qui l'intéresse, il a fallu également écrire une macro permettant d'effectuer ce processus sans que l'utilisateur ait à s'en soucier :

```
(defmacro defsigf
  "Define a function registered to the signal."
  [name & code]
```

```

(let [channel (chan)]
  (signal-register name channel)
  `(go-loop []
    (let [~'msg (<! channel)]
      ~@code
      (recur))))

```

Cela permet à l'utilisateur, à l'intérieur de sa fonction abonné au signal qui l'intéresse, d'utiliser la variable `msg`, qui contient le message qui l'intéresse. Ainsi, chaque fonction définie par l'utilisateur peut s'abonner à un signal sans que cela ne consume les informations de ce signal au profit d'une autre fonction.

Une telle décision permet également de ne pas tomber dans le piège d'un callback hell. Une première solution envisagée était de pouvoir abonner différentes fonctions anonymes de callback à un signal, et lorsque celui-ci obtenait une information, il exécutait séquentiellement les fonctions une par une avec l'information en question.

```

(go-loop []
  (let [msg (<! signal)]
    (doseq [fun functions] (func msg))))

```

Dans cet exemple, lorsque signal reçoit une information, les fonctions de la liste functions sont exécutés séquentiellement.

En plus d'annihiler la possibilité de concurrence — puisqu'une fois l'information envoyée aux signaux abonnés, le scheduler se charge de répartir les calculs sur les différentes fonctions — cela peut rapidement aboutir à du “code spaghetti” avec des callback très nombreux. Par exemple, les codes Javascript de callback hell sont nombreux. La décision d'éviter de tomber dans cet écueil avec plusieurs signaux a donc été prise.

3.2 Gestion du modèle

Un problème majeur s'est posé lors de la gestion du modèle : puisque chaque fonction, une fois son signal reçu crée un nouveau modèle qui remplace l'ancien, comment s'assurer que les fonctions ne travaillent pas sur le même modèle, mais produisent bien différents modèles dans le temps. La décision de mettre un sémaphore sur le modèle a été prise. Ainsi, chaque fonction doit essayer de prendre le “contrôle” du modèle avant de pouvoir en créer un nouveau qui viendra le remplacer. Si cette fonction est en train de calculer le nouveau modèle, aucune fonction ne peut lire le modèle actuel. Elles devront attendre que la fonction de calcul ait fini de son travail et ait remplacé le modèle pour pouvoir agir. Ainsi, on peut s'assurer que le modèle est bien modifié à chaque fois, et qu'aucune modification ne se perd dans le temps.

4 Embla

Contrairement à d'autres langages comme Java, Clojure est très peu verbeux. De plus, les noms des projets ont rarement un rapport avec ce qu'il représente, mais se doivent d'être reconnaissables et faciles à retenir, comme Leiningen, Herbert, Alia, ou Catacumba. Pour le projet, Embla a été le nom retenu. Embla et Ask ("aulne" et "frêne") sont la première femme et le premier homme créés par Odin et ses frères Vili et Vé dans la mythologie nordique. Embla représente donc la naissance des êtres humains, tout comme elle représente la source de tout jeu OpenGL dans notre projet.

4.1 Vue d'ensemble

4.1.1 Architecture

Notre application se divise en quatre parties distinctes.

- Du côté Clojure:
 - La définition des macros permettant à l'utilisateur de construire le modèle et d'interagir avec celui-ci.
 - La définition des primitives de traitement des signaux.
- Du côté Java :
 - La définition du modèle structuré, prenant la forme d'un arbre de formes (les primitives de dessin en deux dimensions : rectangles, triangles, sprites...).
 - Le pendant OpenGL du modèle, sous la forme d'un dictionnaire identifiant Embla / instance de classe forme OpenGL, qui ne sert qu'à retenir les identifiants nécessaires pour redessiner les formes géométriques à partir des données déjà présentes sur la carte graphique.

4.1.1.1 Modèle

Le modèle — en Java — est représenté sous forme d'arbre n-aire. La racine de l'arbre est invariable, et représente son point d'entrée. Chaque noeud dispose ensuite de n fils, puisque le monde peut être composé d'autant de personnages ou d'éléments que l'on souhaite sur une même surface. En effet, chaque élément du jeu est représenté par un noeud de l'arbre. Un personnage, un élément du jeu, ou un élément de décor sera représenté par un noeud.

Dans un jeu à défilement horizontal, on peut imaginer que le noeud de l'arbre aura deux fils : le ciel et le sol. Le sol aura tous les objets reposant sur le sol comme fils, alors que le ciel aura comme fils toutes les objets reposant dans le ciel. Cela permet également de monter au niveau de détail désiré : un personnage peut avoir divers objets, chacun représenté par un fils. Et chaque objet peut lui-même avoir différentes caractéristiques.

Enfin, les possibilités de modularité sont nombreuses : les noeuds de bases permettent de créer n'importe quel élément. En héritant de ces noeuds, on peut créer de nouveaux éléments, et lui attribuer n'importe quel caractéristique. On peut donc obtenir de nouveaux personnages, de nouveaux ennemis, de nouveaux décors, etc... Puisque le jeu obtenu sera en 2D, la class Sprite peut représenter n'importe quel élément.

Afin de pouvoir représenter et manipuler les données comme un arbre, un jeu de macros a été configuré. Ces dernières permettent d'effectuer la majeure partie du travail de l'utilisateur en dissimulant les appels aux classes et fonctions correspondantes. Il est ainsi possible de définir un arbre simple :

```
(defgom "World"
  (defrect {:x 0 :y 0 :length 400 :width 200 :id "sky"})
  (defrect {:x 0 :y 200 :length 400 :width 100 :id "earth"}))
```

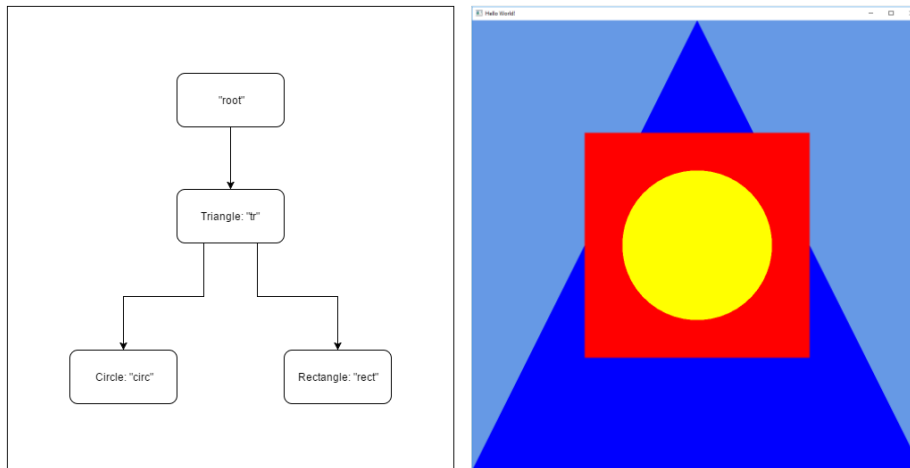


Figure 1: Exemple d'affichage à partir du modèle

4.1.1.2 Signaux

Les signaux se décomposent en trois parties : les signaux de bases, les signaux composés, et les fonctions abonnés aux signaux.

Les signaux de bases — ou primitives — sont les briques de base du jeu. Le temps ou les entrées utilisateurs, notamment sont des primitives du jeu. L'utilisateur peut en créer, mais ne peut pas les détruire, et elles seront tout le temps disponibles. Des signaux comme la vie de son personnage ou les collisions peuvent être implémentés. Cela permet par la suite de bâtir de nouveaux signaux plus complexes.

Les signaux composés sont des signaux prenant en entrée deux signaux, et les combinant pour n'en former plus qu'un. Pour combiner ces deux signaux, une

fonction se charge de réceptionner les informations des signaux en entrée, de calculer ce qui est nécessaire, puis d'émettre ce résultat sur le canal de sortie. Cela permet de bâtir le monde selon ses besoins, et de composer différents éléments pour en former un nouveau. Ainsi, deux signaux de collisions en entrée peuvent calculer si une collision a lieu par exemple, et émettre le signal correspondant.

Enfin, les fonctions abonnés aux signaux correspondent à la fin de la chaîne : une fois les signaux bâtis et fonctionnels, l'utilisateur peut y greffer des fonctions. Ces dernières vont lire le contenu de ces signaux, et agir en conséquence, pour créer un nouveau modèle. Elles sont donc le coeur du moteur, puisque ce sont elles qui modifient l'état du jeu. Ces fonctions peuvent faire ce qu'elles souhaitent, mais elles ne doivent pas créer de nouveaux signaux, ou réémettre sur les signaux de bases. Si elles émettent sur les signaux de base, le graphe de signaux devient cyclique, et le moteur peut tourner en boucle.

4.1.1.3 Vue

La "vue" Embla, ou encore la partie du moteur responsable de l'affichage proprement dit, est réalisée en Java plutôt qu'en Clojure pour faciliter l'utilisation de LWJGL.

Les formes mises à disposition pour l'utilisateur sont implémentées sous la forme de sous-classes de **GLShape**, une classe abstraite contenant la majorité de la logique de gestion de la forme par OpenGL: * L'attribution des identifiants de VAO et VBO à la création de l'instance * La logique de mise à jour des buffers sur la carte graphique * La logique de transformation du repère classique orthonormé avec (0,0) situé sur le coin en haut à gauche avec lequel l'utilisateur définit les positions, en coordonnées du repère que l'on utilise dans OpenGL, avec (-1,-1) en bas à gauche et (1,1) en haut à droite.

La gestion de ces formes est à vocation *strictement interne* à Embla et doit correspondre en intégralité au modèle (en tout cas en termes de présence/absence de noeuds), de telle sorte que la gestion d'erreurs à ce niveau produit tout simplement des **RuntimeException**. En effet, ce genre d'erreur implique que l'utilisateur ait appelé des fonctions Clojure directement au lieu d'utiliser les macros qui lui sont fournies - qui elles garantissent la modification adéquate de la vue après avoir été traitées -, et on peut se permettre de terminer violemment le programme dans ce cas.

La classe **GameEngine**, dans laquelle se trouve la boucle de rendu, représente l'endroit d'où un programme utilisant Embla est lancé. Elle contient toute la logique de création de la fenêtre dans laquelle on affiche le modèle.

4.1.1.3.1 OpenGL

Le fonctionnement d'OpenGL est comparable à celui d'une machine à états. Pour interagir avec des données spécifiques sur la carte graphique, il faut mettre

la machine à états OpenGL dans l'état correspondant. En particulier, en ce qui concerne l'optimisation des transferts CPU/GPU, il est nécessaire de lier les buffers de flottants correspondant à nos données à la machine OpenGL avant de réaliser les opérations de dessin. Ceci nécessite de conserver les identifiants.

Le dessin d'une forme simple se déroule comme ceci sur OpenGL:

```
// Lier le shader program à la machine
GL20.glUseProgram(shader_progid);
// Lier l'ID du VAO enregistrant les buffers de la forme
GL30.glBindVertexArray(vao_shapeid);
// Lier l'ID du buffer positions à la machine
GL20.glEnableVertexAttribArray(0);
GL11.glDrawArrays(GL11.GL_TRIANGLE_FAN, 0, summit_count);
GL20.glDisableVertexAttribArray(0);
// Lier l'ID du buffer couleur à la machine
GL20.glEnableVertexAttribArray(1);
GL11.glDrawArrays(GL11.GL_COLOR_ARRAY, 0, 1);
GL20.glDisableVertexAttribArray(1);
// Déliaer le VAO de la machine
GL30.glBindVertexArray(0);
// Déliaer le shader program de la machine
GL20.glUseProgram(0);
```

Cette logique que l'on désire cacher à l'utilisateur se trouve dans la fonction de rendu propre à chaque sous-classe de GLShape

4.1.1.3.2 Gestion des formes

Nos formes OpenGL servent uniquement à identifier les buffers présents sur la carte graphique, et à s'y référer pour chaque demande de rendu. Les objets implémentant l'interface IGLShape contiennent quatre opérations capitales pour la gestion des formes:

- `<position/color>ToVBO` traduisent:
 - les coordonnées 2D (x, y) sur la projection vue par l'utilisateur (dont nous discutons plus haut) en coordonnées flottantes à 4 dimensions sur la projection gérée par la machine OpenGL.
 - les couleurs fournies par le modèle (concrètement de type java.AWT) en flottants représentant les 4 composantes d'une couleur RGBA.
- `bind<Color/Coordinates>` permettent de fournir à OpenGL un nouveau buffer position ou couleur, modifier en place les buffers de la carte graphique, et ainsi modifier la couleur ou la position de la forme.
- `toProjection` propose un accès après construction de l'objet à la logique de calcul des buffers qui doivent être transférés dans la carte graphique (notamment position et couleur). Cette opération est nécessaire pour obtenir la modification en place de ces buffers, au lieu d'en recréer de

toutes pièces.

- **propagate** réalise l'appel à `toProjection` correspondant aux arguments de la classe concrète implémentant `IGLShape`, de manière à reconstruire les buffers adéquats sur la carte graphique à partir des informations véhiculées par le noeud du modèle passé en argument.

4.1.1.3.3 Boucle de rendu

Une boucle d'affichage OpenGL procède de la manière suivante à chaque tour de boucle:

1. Créer un buffer vide, le remplir d'une couleur de base prédéfinie.
2. Placer dans ce buffer les informations relatives à ce qu'il faut afficher.
3. Echanger le buffer actuellement affiché avec le buffer préparé.

OpenGL requiert donc intrinsèquement de redessiner la scène à chaque tour de boucle, ce qui fait que notre approche de minimisation des opérations d'affichage doit avoir pour but de minimiser les transferts vers la carte graphique en vérifiant quels objets ont changé dans la scène, et ne modifier que ceux-ci sur la carte graphique. Autrement dit, on néglige le coût des appels de dessin au profit des

Son mode de fonctionnement est de vérifier la présence de changements fournis après le parcours du modèle par les signaux, et répercuter ces modifications sur les buffers de la carte graphique. On peut ensuite afficher la scène correctement, en parcourant l'arbre du modèle. Le rendu au fil du parcours de l'arbre nous permet de garantir automatiquement les superpositions des formes en fonction de la profondeur des formes.

On vérifie au passage si notre liste d'objets OpenGL concorde avec notre arbre de formes du modèle, ce qui nous permet d'éviter les comportements indéfinis causés par une éventuelle modification directe du modèle par l'utilisateur, en dehors du cadre du DSL qui lui est fourni.

```
try {
    HashMap<String, Model> modelchanges = changes.get();
    for (Model modelch : modelchanges.values()) {
        GLShape s = glShapes.get(modelch.ID);
        if (s != null)
            s.propagate(modelch);
        else
            throw new RuntimeException("Attempted to propagate changes to GLShape
                                     unknown to the engine");
    }
} finally {
    // Changes are propagated, now draw the world
    changes = Optional.empty();
    draw_model_item(world);
}
```

La variable globale qui contient les changements de l'ancien modèle au nouveau est mise à jour de manière asynchrone par les parcours du modèle suite à la réception d'un signal. Cette variable fait également office de 'file d'attente'. En effet, si plusieurs signaux causent des modifications du modèle, et que ces modifications ne sont pas propagées dans la partie OpenGL du modèle avant l'arrivée d'un autre signal, le remplacement simple causerait un décalage entre la vue et le modèle jusqu'à la propagation réussie des modifications pour le noeud de modèle concerné.

4.1.2 Exécution

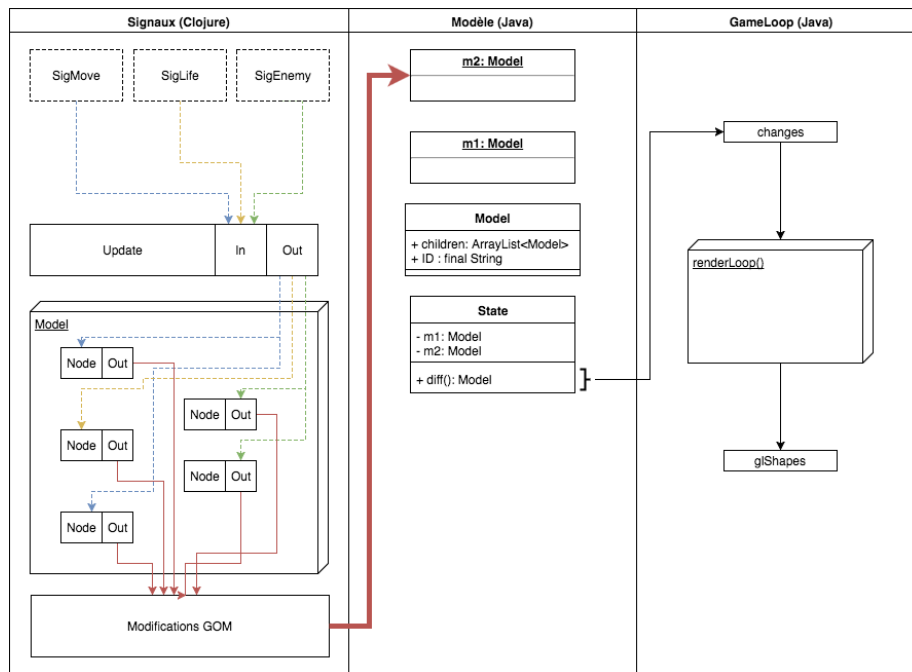


Figure 2: Schéma d'exécution

La réception de signaux (ici **SigMove**, **SigEnemy**, **SigLife**) entraîne, après passage dans la fonction de mise à jour, la modification de noeuds du modèle. On accumule les modifications du modèle pour les passer à la partie de gestion du modèle d'Embla, qui met à jour le modèle courant et le modèle précédent. L'action de réaliser une diff du modèle a un effet de bord sur la "Vue", où la variable **changes** prévue à cet effet est mise à jour pour refléter les modifications du modèle. Ces modifications sont mises en oeuvre au tour de boucle de rendu suivant, en mettant à jour les formes OpenGL connues par le moteur de rendu dans la liste d'associations identifiant/forme **glShapes**.

5 Extensions

A partir de l'itération actuelle d'Embla, aidés par l'architecture existante, nous pourrions imaginer certaines extensions que nous n'avons pas réalisé par soucis de temps et de complexité du programme.

5.1 Acyclisme du graphe de signaux

Dans l'état actuel du projet, la charge de s'assurer que les signaux sont correctement formés revient à l'utilisateur. Une extension immédiate du moteur de rendu est de gérer, en plus de l'arbre de formes, un graphe représentant les signaux, de manière à vérifier pour l'utilisateur la validité de ses définitions de signaux vis-à-vis des signaux déjà existants.

5.2 Hiérarchie des zones de dessin

L'aspect structuré de notre représentation du modèle nous donne un outil puissant pour propager des modifications de manière générale. Par exemple, lorsqu'on réduit la taille d'un élément, on pourrait vouloir que le comportement par défaut des éléments fils de celui-ci subissent une réduction de taille proportionnelle au changement subi. Avec l'introduction de métadonnées supplémentaires dans les noeuds du modèle, on imagine facilement l'introduction de telles relations entre éléments.

Quant au mode de définition de ces relations, on envisagerait plutôt une approche laissant l'utilisateur définir lui-même ce comportement, plutôt que de restreindre ceci à un comportement prédéfini.

5.3 Voyage dans le temps

Ayant décidé de fournir comme exemple d'utilisation d'Embla le déplacement d'un personnage dans une scène, nous n'avons pas réalisé le "voyage dans le temps". Malgré cela, il serait aisé pour un utilisateur de le réaliser, étant donné qu'il suffit d'être capable de calculer l'inverse d'un signal à partir d'un signal pour réaliser ceci, modulo les opérations de création et de destruction, qui nous posent problème. Dans notre optique de réduction du moteur à l'essentiel, nous avons choisi de ne pas garder en mémoire les noeuds de modèle détruits pour des raisons de performance si une scène devient suffisamment compliquée.

6 Bibliographie

1. Un grand merci à tous les anonymes de StackOverflow - <http://stackoverflow.com/>
2. Brave Clojure - <http://www.braveclojure.com/clojure-for-the-brave-and-true/>
3. ClojureDoc - <https://clojuredocs.org/>
4. Modern OpenGL, Anton Gerdelan - <http://antongerdelan.net/opengl/>
5. LWJGL wiki - http://wiki.lwjgl.org/wiki/Main_Page
6. Documentation LWJGL - <http://javadoc.lwjgl.org/>
7. Documentation GLFW - <http://www.glfw.org/docs/latest/>
8. Code source de React - <https://github.com/facebook/react>
9. Elm: Concurrent FRP for Functional GUIs, Evan Czaplicki - <http://elm-lang.org/papers/concurrent-frp.pdf>