Sorbonne Université
Faculté des Sciences et Ingénierie

PSTL

4I508

Programmation Fonctionnelle Réactive en Clojure

Étudiants
Clément Busschaert
Antoine Meunier

Superviseur
Frédéric Peschanski

27 mai 2018

Table des matières

Chapitre 1

Le projet

Ce projet s'inscrit dans la continuité de précédents PSTLs. L'objectif principal est d'étendre les fonctionnalités actuelles vers une API réactive.

YAW (Yet Another World) est une bibliothèque Java, développée au fil de plusieurs PSTL, de manipulation et affichage d'objets 3D avec une surcouche Clojure. L'objectif de ce projet est de trouver un moyen d'offrir une interface de programmation interactive permettant de facilement développer des applications 3D de façon déclarative.

Chapitre 2

Remerciements

Nous tenons à remercier notre encadrant Frédéric Peschanski pour ses conseils et sa patience.

Chapitre 3

YAW : code existant

YAW est une bibliothèque Java/Clojure de rendu et manipulation 3D faisant principalement interface avec OpenGL.

OpenGL est une bibliothèque de rendu très bas-niveau, manipulant des points et des triangles, sans aucune notion d'objets ou de transformations telles que des rotations ou translations. YAW offre ces fonctionnalités de manipulation de plus haut niveau, tout en gardant le contrôle de l'utilisateur sur la géométrie des objets manipulés.

3.1 Interface Java

3.1.1 La classe `World`

L'interface publique Java de YAW est principalement accédée par la classe `World`.

Cette classe représente un fenétre sur un espace 3D et sa population. Le listing 3.1 liste quelques méthodes proposées par cette classe.

Un processus notable est celui d'ajout d'un objet 3D dans la scène. L'objet (dans le code *item*) est inséré dans le monde avec un nom, une position et une échelle, et un *mesh*. Ce *mesh* est aussi créé via une méthode de `World` avec les données bas-niveau de géométrie requises par OpenGL.

Cette classe propose un nombre de méthodes d'altération de la scène directement, à l'exception des lumières, qui nécessitent de récupèrer la `SceneLight` et d'interagir avec.

```
public class World implements Runnable {
  // ... implementation de Runnable omise, il s'agit de la gestion de la
  //     fenetre.

  public World(int pInitX, int pInitY, int pInitWidth, int pInitHeight);

  public Item createItem(String id, float[] pPosition, float pScale, Mesh
  pMesh);

  public Mesh createMesh(float[] pVertices, float[] pTextCoords, float[]
  pNormals, int[] pIndices, int pWeight, float[] rgb, String
  pTextureName);

  public Item createBoundingBox(String id, float[] pPosition, float pScale,
  float[] pLength);

  public boolean isInCollision(Item item1, Item item2);

  public void setSkybox(float pWidth, float pLength, float pHeight, float pR,
  float pG, float pB);
  public void removeSkybox();

  public SceneLight getSceneLight();

  public void addCamera(int pIndex, Camera pCamera);

  //... reste omis
}
```

Listing 3.1 – Interface incomplète de la classe World

3.1.2 Les lumières

```
public class SceneLight {
  public static final int MAX_POINTLIGHT = 5;
  public static final int MAX_SPOTLIGHT = 5;

  public SceneLight();

  public void removeAmbient();

  public void removeSun();

  public void setPointLight(PointLight pl, int pos);

  public void setSpotLight(SpotLight sl, int pos);

  public DirectionalLight getSun();
  public void setSun(DirectionalLight sun);

  public AmbientLight getAmbientLight();
  public void setAmbient(AmbientLight ambient);

  //... reste omis
}
```

Listing 3.2 – Interface incomplète de la classe SceneLight

La gestion des lumières se fait à travers une instance de la classe `SceneLight`, présentée partiellement en listing 3.2.

Il est possible d'avoir jusqu'à `MAX_POINTLIGHT` sources de lumières ponctuelles, et jusqu'à `MAX_SPOTLIGHT` projecteurs.

Contrairement à la création des objets dans `World`, la création des lumières se fait directement par constructeur.

3.2 Interface Clojure

L'interface Clojure de YAW est une interface quasiment traduite. Un fichier `world.clj` comporte un ensemble de fonctions appelant directement les méthodes de la classe `World`.

```
(defn start-universe! [& {:keys [width height x y]}] (...))

(defn create-mesh! [world & {:keys [vertices text-coord normals faces weight
  rgb texture-name]}] (...))

(defn create-item! [world & {:keys [id position scale mesh]}] (...))

(defn create-block! [world & {:keys [id position scale color texture]}] (...))

(defn create-bouding-box! [world & {:keys [id position length scale]}] (...))
(defn add-bounding-box! [item bounding-box] (...))
(defn check-collision! [world item1 item2] (...))

(defn rotate! [item & {:keys [x y z]} (...))
(defn translate! [item & {:keys [x y z]} (...))
```

Listing 3.3 – Interface initiale simplifiée de `world.clj`

Les fonctions clojure présentée en listing 3.3 ont été simplifiées par souci de brièveté : les arguments suivant les esperluettes sont optionnels et ont des valeurs par défaut. Ces fonctions ont un point d'exclamation à la fin de leur nom, ce qui dans la communauté Clojure a été accpeté comme convention pour signifier qu'une fonction a des effets de bords sur la mémoire et n'est pas « thread-safe. »

L'interface est incomplète. Il manque la gestion des lumières, des caméras, et d'autres fonctionnalités diverses offertes par la bibliothèque Java. Le premier travail a été de compléter cette interface.

3.3 Refactoring

3.3.1 Simplification

L'interface Clojure impérative est assez simple, mais le code à l'intérieur n'était pas suffisamment cohérent. Par exemple, la fonction `create-mesh!`, qui prend des données géométriques, crée par défaut un cube blanc, avec le code présenté en listing 3.4.

```
(defn create-mesh!
  [world & {:keys [vertices text-coord normals faces weight rgb texture-name]
            :or   {texture-name ""
                   rgb          [1 1 1]
                   weight       1
                   vertices     {:v0 [-1 1 1] :v1 [-1 -1 1] :v2 [1 -1 1] :v3 [1 1 1]
                                 :v4 [-1 1 -1] :v5 [1 1 -1] :v6 [-1 -1 -1] :v7 [1 -1 -1]}
                   normals      {:front  [0 0 1 0 0 1 0 0 1 0 0 1]
                                 :top    [0 1 0 0 1 0 0 1 0 0 1 0]
                                 :back   [0 0 -1 0 0 -1 0 0 -1 0 0 -1]
                                 :bottom [0 -1 0 0 -1 0 0 -1 0 0 -1 0]
                                 :left   [-1 0 0 -1 0 0 -1 0 0 -1 0 0]
                                 :right  [1 0 0 1 0 0 1 0 0 1 0 0]}
                   faces        {:front  [0 1 3 3 1 2]
                                 :top    [4 0 3 5 4 3]
                                 :back   [7 6 4 7 4 5]
                                 :bottom [2 1 6 2 6 7]
                                 :left   [6 1 0 6 0 4]
                                 :right  [3 2 7 5 3 7]}
                   text-coord   {:front  [0 0 0 0.5 0.5 0.5 0.5 0]
                                 :back   [0 0 0.5 0 0 0.5 0.5 0.5]
                                 :top    [0 0.5 0.5 0.5 0 1 0.5 1]
                                 :right  [0 0 0 0.5]
                                 :left   [0.5 0 0.5 0.5]
                                 :bottom [0.5 0 1 0 0.5 0.5 1 0.5]}}}]

  (.createMesh world
               (float-array (flat-map vertices))
               (float-array (flat-map text-coord))
               (float-array (flat-map normals))
               (int-array (flat-map faces))
               (int weight) (float-array rgb) texture-name))
```

Listing 3.4 – Code complet de `create-mesh!`

Le problème principal de cette fonction est qu'elle demande en paramètres des données structurées par des *maps*, puis aggrège chaque structure avec une fonction `flat-map` qui « applatit » les structures à plusieurs niveaux (ici des vecteurs dans des maps) avec le *one-liner* `(flatten (conj (vals m)))` (où `m` est la map en paramètre). Le problème majeur vient du manque de garantie du maintien de l'ordre des valeurs. Les tableaux que le code Java de YAW attend sont très sensibles à l'ordre des valeurs, car ils dèterminent la géométrie exacte de l'objet 3D à envoyer à OpenGL.

Le second problème est le « gâchis » des structures. La map `vertices` est décorée de *keywords* pour chaque vertex (`:v1`,`:v2`, etc), mais ces keywords ne servent en rien à réferer aux vertex dans les autres structures telles que `faces`.

Il était important de décider de spécifier une structure plus simple pour représenter les meshs.

L'idée de nommer les vertex et de les réutiliser facilite énormément la création de géométries à la main.

La décision a été faite d'abandonner l'idée générale de « faces » et de la remplacer par des triangles, parce qu'OpenGL fonctionne quasiment exclusivement avec des triangles. Chaque triangle a trois vertex, et un vecteur normal pour aider au calcul des couleurs.

Les textures ont aussi étés abandonnées parce que leur fonctionnement était non-vérifié et incertain.

Il a donc été spécifiée une structure de données utilisant des vertex nommés et des triangles.

Un cube aligné avec les axes, centré en `(0,0)`, peut être simplifié par la structure présentée en listing 3.5.

```
          {:vertices {:a [1 1 1]
                      :b [1 -1 1]
                      :c [-1 -1 1]
                      :d [-1 1 1]
                      :e [1 1 -1]
```

```
                        :f [1 -1 -1]
                        :g [-1 -1 -1]
                        :h [-1 1 -1]}
           :tris [{:n [0 0 1]
                   :v [:a :c :b]}
                  {:n [0 0 1]
                   :v [:a :d :c]}
                  {:n [0 0 -1]
                   :v [:e :f :h]}
                  {:n [0 0 -1]
                   :v [:f :g :h]}
                  {:n [0 1 0]
                   :v [:a :e :h]}
                  {:n [0 1 0]
                   :v [:a :h :d]}
                  {:n [0 -1 0]
                   :v [:b :c :f]}
                  {:n [0 -1 0]
                   :v [:f :c :g]}
                  {:n [1 0 0]
                   :v [:a :f :e]}
                  {:n [1 0 0]
                   :v [:a :b :f]}
                  {:n [-1 0 0]
                   :v [:d :h :c]}
                  {:n [-1 0 0]
                   :v [:c :h :g]}]}
```

Listing 3.5 – Géométrie d'un cube

Aprés isolation de cette structure dans une fonction `box-geometry` dans un nouveau fichier `mesh.clj` et *namespace* `yaw.mesh` dédié à la géométrie des meshs, le code de `create-mesh!` peut être remplacé par celui de `create-simple-mesh!` proposé en listing 3.6.

```
(defn create-simple-mesh!
  [world & {:keys [geometry rgb]
            :or {geometry (yaw.mesh/box-geometry)
                 rgb [0 0 1]}}]
  (let [{:keys [vertices tris]} geometry
        vidx (zipmap (keys vertices) (range (count vertices)))
        coords (float-array (mapcat second vertices))
        normals (float-array (mapcat (fn [{n :n v :v}] (concat n n n)) tris))
        indices (int-array (mapcat (fn [{n :n v :v}] (map #(% vidx) v)) tris))]
    (.createMesh world coords normals indices (float-array rgb))))
```

Listing 3.6 – Code complet de `create-simple-mesh!`

Chapitre 4

Recherche

4.1 Inspirations initiales

4.1.1 Reagent

Reagent est une bibliothèque Clojurescript se définissant comme un « React minimal pour Clojurescript. »

Reagent est une interface par dessus React, donc elle ne fait pas elle-même le travail de maintenance de la *page*, mais elle propose une interface en Clojurescript, ce qui en a fait une première référence.

Reagent permet de définir des composants à assembler pour créer une page entière, grace à une interface très intuitive, c'est à dire de directement lister, sous forme de structure clojure, le HTML escompté au rendu, comme dans l'exemple 4.1.

```
(defn simple-component []
  [:div
   [:p "I␣am␣a␣component!"]
   [:p.someclass
    "I␣have␣" [:strong "bold"]
    [:span {:style {:color "red"}} "␣and␣red␣"] "text."]])
```

Listing 4.1 – Composant Reagent simple

L'interface est plus complexe si on veut « retenir un état. » Le listing 4.2 montre comment Reagent propose de faire un compteur. Le code fait utilisation d'*atomes*, qui sont un moyen en Clojure d'avoir des « variables. » Les modifier fait appel au même type d'interface impérative utilisée dans `world.clj` (listing 3.3).

```
(ns example
  (:require [reagent.core :as r]))

(def click-count (r/atom 0))

(defn counting-component []
  [:div
   "The␣atom␣" [:code "click-count"] "␣has␣value:␣"
   @click-count ".␣"
   [:input {:type "button" :value "Click␣me!"
            :on-click #(swap! click-count inc)}]])
```

Listing 4.2 – Composant Reagent avec état

Insérer un composant dans un autre est aussi simple que de citer son nom à la place d'un élément HTML. La structure renvoyée par un composant n'est donc pas un flot continu de *pseudo-HTML*, mais peut contenir des fonctions qui pourront être appelées pour récupérer le contenu total du composant parent.

L'idée finale est de créer un « méga-composant » à partir d'autres plus petits, qui correspond à l'application entière et servira de référence racine à Reagent pour tout calculer.

C'est une interface adaptée au développement d'applications web. Les principes de composants sont adaptables à notre contexte de scène 3D à condition d'avoir une notion de « groupe d'objets. »

4.1.2 Unity

Un autre objectif du projet est de fournir un moyen de développer des applications 3D, et donc de contrôler une fréquence d'affichage et de mise à jour.

Unity est, en plus de l'éditeur qui va avec, un moteur de jeu qui propose une API en C#, et qui fonctionne aussi à base de *composants*. L'API propose des classes à hériter et des méthodes à implémenter pour spécifier le comportement de chaque composant.

Il est ainsi possible de spécifier un composant qui nécessite d'être attaché à un objet avec une boite de collision (`Collider` dans Unity) et qui va changer la couleur de l'objet à chaque fois que le joueur clique dessus, mais pas plus souvent qu'une fois par seconde, avec une simple implémentation de la méthode `Update()` qui a accès à un *delta-time* représentant le temps passé depuis le dernier appel à cette méthode `Update()`. Cette durée peut être comparée pour vérifier qu'elle dépasse une certaine valeur, ou utilisée en facteur pour les rendre les déplacements et les interactions physiques plus correctes par rapport au rendu final. Sans cette durée *delta-time*, si on déplace un objet dans l'espace de une unité à chaque appel à `Update()`, si l'appel se fait à des fréquences variables dues à des fluctuations d'occupation du CPU, alors la vitesse de l'objet dans le jeu changera aussi.

L'API de Unity propose un moyen de vérifier l'état du système à tout moment. Dans la méthode `Update()`, on peut vérifier quelles touches du clavier sont enfoncées, où est la souris, si un bouton a été cliqué, et d'autres informations. Au moment où ces informations sont lues et traitées, elles peuvent être devenues fausses à cause du temps écoulé : Unity fragmente le temps selon les itérations du cycle d'affichage.

Le système va commencer son cycle, calculer une *frame* initiale, normalement sans événement matériel, puis appeler `Update()` pour tous les composants dans la scène, dans un ordre indeterminé, collectant les informations matériels pendant ce temps, puis terminera son cycle en ouvrant au « public » ses événements matériels, avant de recommencer.

Cette idée de boucle est intéressante pour une raison : en collectant les événements indépendamment et en les offrant dans l'API, on permet à chaque composant de se mettre à jour comme il veut, par exemple de réagir seulement si deux événements indépendants ont lieu en même temps, ce qui ne serait pas possible si on utilisait une approche « *callback* » qui associerait à chaque événement une réaction spécifique.

Chapitre 5

Conclusion

Ce projet a mis plus de temps qu'il n'aurait dû à démarrer, et la phase de recherche s'est terminée trop tard pour que des avancées notables soient faites. Trop de temps a été alloué à l'étude de l'existant, et pas assez à l'expérimentation.

Ce projet nous a apporté de nouvelles connaissances théoriques et nous a approchés à de nouveaux concepts de programmation.

D'un point de vue technique, nous avons réalisé la difficulté de travailler avec des interactions en temps-réel et de la translation entre paradigmes.

D'un point de vue humain, ce projet nous a permis de mieux comprendre quel rôle nous pouvions jouer dans un projet de groupe, et nous a montré un bon nombre d'erreurs à ne pas répéter.

YAW n'a au final pas progressé dans ses fonctionnalités, nous espérons que ce travail de recherche pourra aider ceux qui voudrons reprendre le projet.