

Rapport projet PFA 2018-2019

Lam NGUYEN THIET Kenyu KOBAYASHI

13 mai 2019

Table des matières

1	Introduction	3
2	Le jeu	3
2.1	Commandes	3
2.2	But	3
2.3	Factions	3
2.4	Unités	3
2.5	Items	4
2.6	Plateau du jeu	4
2.6.1	Type de cases	4
2.6.2	Caractéristiques du terrain	4
2.7	Tour par tour	4
2.7.1	Mouvement	4
2.7.2	Attaque	4
2.8	Intelligence Artificielle	5
3	Répartition du travail	5
3.1	NGUYEN THIET	5
3.2	KOBAYASHI	6
4	Développement du jeu	6
5	Quelques explications	8
5.1	Contexte & Boucle principale	8
5.2	Formalisation et généralisation	9
5.3	Et le reste ?	10
6	Problèmes et autre remarques	10
6.1	Build circulaire	10
6.2	Conception confuse	10
6.3	De bien trop grandes ambitions...	10
6.3.1	Quelques dessins non utilisés dans le rendu	11

7	Annexe & Source	11
7.1	Contact	11

1 Introduction

Ce projet implémente un jeu dans le langage OCAML, en utilisant les paradigmes fonctionnelles (en majorité), impératives, orientée objet.

2 Le jeu

2.1 Commandes

Nous n'avons pas eu le temps d'implémenter l'interaction avec la souris. Les commandes se font intégralement au clavier.

- Z S Q D, pour bouger la caméra, resp., en haut, bas, gauche droite.
- W C, pour zoomer, resp. dézoomer
- Entrée pour sélectionner un allié
- Y U I O P pour sélectionner un action, resp. utiliser la bombe nucléaire, utiliser un pack de soin, ramasser un item, attaquer, se déplacer.
- Les flèches permettent de déplacer le curseur.

Pour effectuer une action, il faut faire :

1. Sélectionner un personnage
2. Sélectionner une action
3. Sélectionner une destination
4. Appuyer sur entrée pour confirmer

2.2 But

Le but du jeu est de détruire les factions ennemies. Pour ce faire, il suffit de tuer toutes leur unités.

Vous commencez avec 5 unités, les ennemis commencent avec 2 et une ville chacun. Ils peuvent apparaître à l'infini. Le seul moyen de détruire une ville est d'envoyer la bombe nucléaire. La stratégie est donc de saisir le plus rapidement les bombes nucléaires pour détruire les villes aussitôt que possible.

2.3 Factions

Il y a 3 factions dans le jeu. Les cagoulés, les verts et les bleus. Vous contrôlez les bleus.

2.4 Unités

Il existe deux types d'unités.

- Le soldat est l'unité de base. Elle peut se déplacer, attaquer et récupérer des items.
- La ville permet de faire apparaître des soldats. Cette unité est très importante car si on la perd, on ne peut plus produire de soldats.

2.5 Items

Il existe deux items.

- Le pack de soin régénère la vie des unités. Il est possible d’avoir plus de points de vie que l’on avait au départ.
- La bombe nucléaire détruit tout dans un rayon de 3 cases.

2.6 Plateau du jeu

Le plateau est une grille d’hexagone. La carte est une île déserte dans l’océan avec des biomes variés.

2.6.1 Type de cases

Il y a 3 types de cases. A l’heure actuelle, elles ne sont que cosmétiques, par la suite elles peuvent avoir un impact sur l’efficacité de combat d’unités en fonction de leur faction, mais par manque de temps on n’a pas pu les faire.

Il y a la neige, le désert et l’herbe.

2.6.2 Caractéristiques du terrain

En plus du biome, il y a des caractéristiques sur le terrain pour chaque case.

- Les forêts et les collines coûtent plus cher pour le mouvement.
- Les montagnes et les lacs sont des obstacles ne peuvent pas être traversés.

2.7 Tour par tour

Le jeu se déroule en tour par tour, similaire au jeu *Civilization*. C’est d’ailleurs sur quoi je me suis inspiré pour le jeu.

2.7.1 Mouvement

Chaque unité a un nombre de mouvement. Lorsqu’elle se déplace d’une case, elle consomme n points selon la case sur laquelle elle atterrit. Dans le jeu, les soldats sont les seuls à pouvoir se déplacer. Les villes ne peuvent pas.

2.7.2 Attaque

Pour tuer les autres unités, il faut les attaquer. Encore une fois de manière analogue à *Civilization*, les unités disposent d’une force d’attaque et d’une force de défense.

Par la suite, *src* et *dst* représentent respectivement l’unité qui attaque et l’unité qui défend.

Lorsque deux unités s’attaquent, les nouveaux point de vie se calculent de cette manière :

$$healthpoints_{dst,new} = \max\{0, healthpoints_{dst,old} - strength_{attack,src}\} \quad (1)$$

$$healthpoints_{src,new} = \max\{0, healthpoints_{src,old} - strength_{defense,dst}\} \quad (2)$$

Et leurs nouvelles positions :

$$x_{dst,new}, y_{dst,new} = \begin{cases} \text{null, null} & \text{si } healthpoints_{dst,new} = 0, \\ x_{dst,old}, y_{dst,old} & \text{sinon.} \end{cases} \quad (3)$$

$$x_{src,new}, y_{src,new} = \begin{cases} \text{null, null} & \text{si } healthpoints_{src,new} = 0, \\ x_{dst,old}, y_{dst,old} & \text{si } healthpoints_{dst,new} = 0, \\ x_{src,old}, y_{src,old} & \text{sinon.} \end{cases} \quad (4)$$

2.8 Intelligence Artificielle

Les IA sont assez simple. De base, elles se déplacent au hasard. Si elles rencontrent un ennemi, elle va se diriger vers cet ennemi en priorité. Si il y a un autre ennemi elles s'attaquent, même si ce n'était pas l'ennemi en priorité.

En dessous d'un certain seuil, les IA vont chercher à fuir et chercher un pack de soin. Mais si il y a un ennemi tout près, elles vont se suicider et attaquer cet ennemi, car elles savent qu'elles vont mourir et vont préférer attaquer pour donner une chance à leur alliés.

Sinon, en mode patrouille, si elles voient une bombe nucléaire, elles la prennent et l'utilise sur un ennemi au hasard.

3 Répartition du travail

3.1 NGUYEN THIET

- Gestion des appels à la bibliothèque SDL (*e.g.*, gestion du render, chargement des textures, initialisation du windows, *etc.*)
- Dessins (tuiles, caractéristiques de terrain, soldats, ville, interfaces, fond d'écran, titre, items, effets spéciaux)
- Définition des boucles du menu et du jeu et le retour que chaque boucle passe entre elles.
- Boucle principale. Gestion d'un type **context**, comment le mettre à jour et comment récupérer informations contenus pour avancer le jeu dans le temps
- Définition du type des **unités** et les méthodes associées
- Implémentation du plateau de jeu, le système de couches et la grille d'héxagone
- Système d'animation (animation de personnages et effets spéciaux)
- Comportements des unités ennemies (Choix des actions selon le comportement, choix du comporter selon la situation)
- Implémentation de la caméra (déplacement et zoom)

- Gestion du curseur et un afficheur de portée de mouvement/attaque
- Affichage d'informations par rapport à une unité (points de mouvements et barre de vie)
- Système d'interface avec quelque chose similaire aux *event listeners* de Javascript
- Définition et interaction avec les items
- Interaction clavier souris avec le jeu (en se servant des appels à *SDL*)
- Implémentation des algorithmes de recherche dans le graphe (pour les chemins)
- Empaquetage des textures
- Rédaction du rapport

3.2 KOBAYASHI

- Calcul des PV et mise à jour du plateau pendant les attaques
 - Menu de réglages
 - Cache LRU
- Il y a également quelques travaux qui n'ont pas été retenus.
- Un système d'interface.

4 Développement du jeu

Chaque phase sera développée plus en profondeur dans la suite du rapport. Chaque implémentation est listé par ordre chronologique dans le développement.

Phase 1 : Fondation

1. Familiarisation avec *SDL* et factorisation de code qui était assez récurrent.
2. Implémentation de la boucle principale
3. Généralisation du type `context` et sa mise à jour

Phase 2 : Instances de jeu

1. Définition des instances du jeu (la boucle du menu, du jeu etc.)
2. Création de boutons temporaires pour lancer le jeu et le quitter si on appuie sur la croix

Phase 3 : Plateau de jeu

1. Définition du plateau de jeu (représenté par une matrice)
2. Implémentation des fonctionnalités de la grille d'hexagone
3. Définition des tuiles et des `enum` qui la définissent
4. Les dessins du plateau (tuiles, forêts, etc.)

Phase 4 : Unités

1. Définition des unités
2. Définition des constantes qui les définissent
3. Les dessins des unités

Phase 5 : Actions

1. Formalisation et généralisation des actions et leur retour pour qu'ils puissent tous être du même type
2. Implémentation des actions de déplacement et attaques
3. Interaction temporaire avec les unités avec le clavier
4. Interaction du retour des actions avec le système de contexte

Phase 6 : Animation

1. Définition d'un type stockant les informations nécessaire aux animations
2. Système de rendu pour les animations couplé avec les effets spéciaux

Phase 7 : IA

1. Formalisation d'un comportement d'une unité
2. Systèmes similaire aux automates d'états finis pour sélectionner le comportement de chaque unités selon son environnement
3. IA qui se déplacement au hasard,
4. et attaque une cible si il y en a une qui se trouve à proximité,
5. et qui va chercher des packs de soin si elles n'a pas beaucoup de points de vie,
6. et qui va chercher les bombes nucléaires si elle peut.

Phase 8 : Interfaces

1. Affichage des informations liés à chaque unités (ses points de vie et points de mouvement)
2. Système d'interface avec des *event listeners*
3. Formalisation et généralisation des interactions avec les interfaces dans le contexte

5 Quelques explications

5.1 Contexte & Boucle principale

Vous remarquerez que le fichier `context.ml` contient une (très) grande partie du code. Son rôle est de définir le type `context` et de mettre à jour la variable qui y est associé. Dans ce type on trouve tout ce dont nous avons besoin pour faire tourner le jeu :

```
type t = {
  over : bool;
  camera : MCamera.t;
  grid : MGrid.t;
  cursor_selector : MCursor.cursor;
  faction_list : MFaction.t list;
  faction_controlled_by_player : MFaction.t;
  action_src : MHex.axial_coord option;
  action_dst : MHex.axial_coord option;
  action_layer : MLayer_enum.t option;
  action_type : MAction_enum.enum option;
  movement_range_selector : MTile.t list;
  to_be_added : MEntity.t list;
  to_be_deleted : MEntity.t list;
  animation : MAnimation.t;
  new_turn : bool;
  frame : int;
  scale : float;
  interface : MInterface.structure;
  current_layer : MLayer_enum.t;
  window : Sdl.window;
}
```

NB : Ceci est une ancienne version du type `context`. Il se peut qu'elle ait évolué au fil du temps.

Et pour chacun de ces attributs, il existe une fonction qui fait un appel aux autres modules, prend la réponse et modifie l'attribut dans l'objet `context`. La boucle principale `run` se sert de `context` pour afficher ce qu'il y a à afficher.

La boucle de menu possède un type similaire, mais étant moins complexe, nous n'allons pas l'aborder en détail. Les noms des attributs parlent d'eux même.

Pour les attributs qui ne sont pas évidents :

- `cursor_selector` représente l'objet *Curseur*, qui contient sa position entre autres.
- `action_src`, lorsqu'une action est en train d'être sélectionnée (pour le joueur humain), représente les coordonnées de la source de l'action
- `action_dst`, de manière similaire, représente la destination de l'action
- `action_layer` représente la couche sur laquelle on veut effectuer l'action
- `action_type` nous dit quelle action effectuer

- `movement_range_selector` est l'ensemble des tuiles sur lesquelles la prochaine action va avoir un effet. Par exemple, si on veut faire un mouvement, on verra le tracé de la trajectoire.
- `to_be_added` et `to_be_deleted` représente une liste d'unité à ajouter, resp. à effacer. J'en discuterai plus tard dans le rapport.

5.2 Formalisation et généralisation

Dans les phases il y a beaucoup de *"formalisation et généralisation"*. Plus précisément, j'ai cherché à généraliser le retour de certains types pour pouvoir les mettre dans une liste pour qu'ensuite `context.ml` s'en serve. Nous allons prendre l'exemple des actions, mais cela s'applique aussi pour les interfaces.

Voici à quoi ressemble le produit de type que les actions retournent :

```
type res = {
  added : MEntity.t list;
  deleted : MEntity.t list;
  animation : MAnimation.t;
}
```

Chaque action retourne une liste des unités ajoutées, détruites et une animation. Généralement, les unités détruites sont les anciennes entités avant l'action et les ajoutées sont les mêmes entités sous un nouvel état.

L'intérêt est de pouvoir regrouper plusieurs actions qui se passent en même temps et en combiner les résultats, à l'aide de cette fonction :

```
let add r1 r2 = {
  added = r1.added @ r2.added;
  deleted = r1.deleted @ r2.deleted;
  animation = MAnimation.add r1.animation r2.animation;
}
```

`context.ml` dispose de toutes les unités effacées ainsi que toutes les animations à faire. Il les traite de cette façon :

```
let faction_list =
  List.fold_right (
    fun x acc-> (MFaction.update_entities x ctx.to_be_added
      → [] ) :: acc
  ) ctx.faction_list []
```

```
let faction_list =
  List.fold_right (
    fun x acc-> (MFaction.update_entities x []
      → (MAction.get_deleted res) ) :: acc
  ) ctx.faction_list []
```

La raison pour laquelle on ne fait pas les deux en même temps est qu'il faut attendre que l'animation s'arrête avant de pouvoir les remettre.

5.3 Et le reste ?

A l'heure où j'écris ce rapport (11/05/2019 23 :58), je pense qu'il est déjà bien chargé. J'ai jugé que les deux points précédemment cités étaient les plus importants à comprendre. Ces *designs* sont assez récurrents. Le projet n'étant pas encore complètement fini, j'ai préféré m'arrêter ici pour les explications, et continuer à travailler sur la finition du projet et les quelques tâches qui restent à faire.

Si il y a besoin de clarifications, je les ferai lors de la soutenance.

6 Problèmes et autre remarques

6.1 Build circulaire

Vous remarquerez qu'il y a beaucoup de fichier ayant le nom `x_enum.ml`. C'est ma solution pour contourner le problèmes des *circular builds*. Par exemple, `faction_enum.ml` contient des constantes pour chaque factions dans le jeu. Les factions contiennent des unités, et les unités ont besoin de savoir à quelle faction elles appartiennent. On voit clairement le problème de build circulaire.

On se rend compte qu'au final, les unités n'ont pas besoin de savoir tout sur la faction, mais seulement dans quel camp ils sont (représenté par un `enum`), et éventuellement un identifiant unique si il y a plusieurs camps ayant la même faction. Ainsi, les factions importent ce type, et les unités importent également ce type. Les unités ont besoin de tout juste ce qu'il faut et les factions contiennent les informations nécessaires pour leur bon fonctionnement.

Pour le reste des `x_enum.ml` qu'on rencontre, on peut en dire la même chose.

6.2 Conception confuse

Assez tôt dans le projet nous avons cru avoir besoin du paradigme programmation orientée objet. Finalement, ça ne servait à pas grand chose à part la factorisation de code. Par manque de temps je n'ai pas reconverti en type produit comme il l'aurait fallu le faire.

6.3 De bien trop grandes ambitions...

Au début du projet nous étions bien trop ambitieux. C'était du au fait que c'était le seul projet qu'on avait à ce moment là. Puis d'autres projets se sont greffés à nos emplois du temps, puis les examens, puis les concours externes à passer. Nous sommes quand même contents de ce que nous avons fait. J'ai fait attention à rendre le code assez générique, si on le souhaitait, on peut facilement rajouter des unités, des actions etc.

6.3.1 Quelques dessins non utilisés dans le rendu



FIGURE 1 – Des icônes pour l'arbre de technologie



FIGURE 2 – Des icônes pour l'arbre de compétence



FIGURE 3 – Une autre unité qui attaque à distance

7 Annexe & Source

- Le pseudocode de la grille par hexagone : <https://www.redblobgames.com/grids/hexagons/>
- La musique du jeu : <https://soundcloud.com/leagueoflegends/omega-squad-teemo>
- Les dessins ont été faits avec Illustrator <https://www.adobe.com/products/illustrator.html>

7.1 Contact

- nguyenthiet.lam@gmail.com
- lam.nguyen-thiet@u-psud.fr
- kenyu.kobayashi@u-psud.fr