Sorbonne Université

Rapport final

21 mai 2023

Dune II

Projet PAF

Auteurs :
Daniel Sima
Yukai Luo

Encadrants :
Frédéric Peschanski
Romain Demangeon

Table des matières

1 Introduction du projet

Dans le cadre de l'UE de Programmation Avancée en style Fonctionnel, ce projet consiste à développer un jeu sûr dans le langage fonctionnel Haskell. Ce projet se propose de reprendre la spécification faite en examen et de poursuivre avec l'implémentation, d'une part avec la programmation sûre avec des invariants, pre/post conditions et des tests unitaires, et d'autre part avec la programmation en style fonctionnel.

1.1 Présentation du projet

Dune II étant un jeu de stratégie en temps réel (RTS) sur une grille, utilisant de la 2D, ou plusieurs joueurs s'affrontent en construisant des bâtiments et créant des unités. Le but étant de vaincre l'ennemi en détruisant sa base.
Une bibliothèque graphique s'est avéré nécessaire, nous avons donc repris le squelette du TME 6 comme base, avec la bibliothèque SDL, pour implémenter la partie graphique.

1.2 Manuel d'utilisation du jeu

Le jeu utilise la bibliothèque graphique `SDL` pour afficher le contenu graphique, comme les différentes sprites et textures utilisées. Cependant nous avons eu rapidement besoin d'afficher du texte a l'écran pour représenter des données comme les crédits, l'énergie, le prix, les points de vies, etc ... Pour cela nous avons fait appel au module `SDL.Font` present dans la biblitheque `sdl2-ttf` et qui peut être installé avec `stack` en suivant cette commande :

```
1 stack install sdl2-ttf
```

`sdl2-ttf` doit ensuit être ajouté dans le fichier `package.yaml` dans dependencies.
Comme tout projet `stack`, nous utilisons `stack run` directement pour build et exécuter notre projet. Ceci doit être fait dans le répertoire du projet, ainsi nous tombons sur la version STL du jeu Dune II :

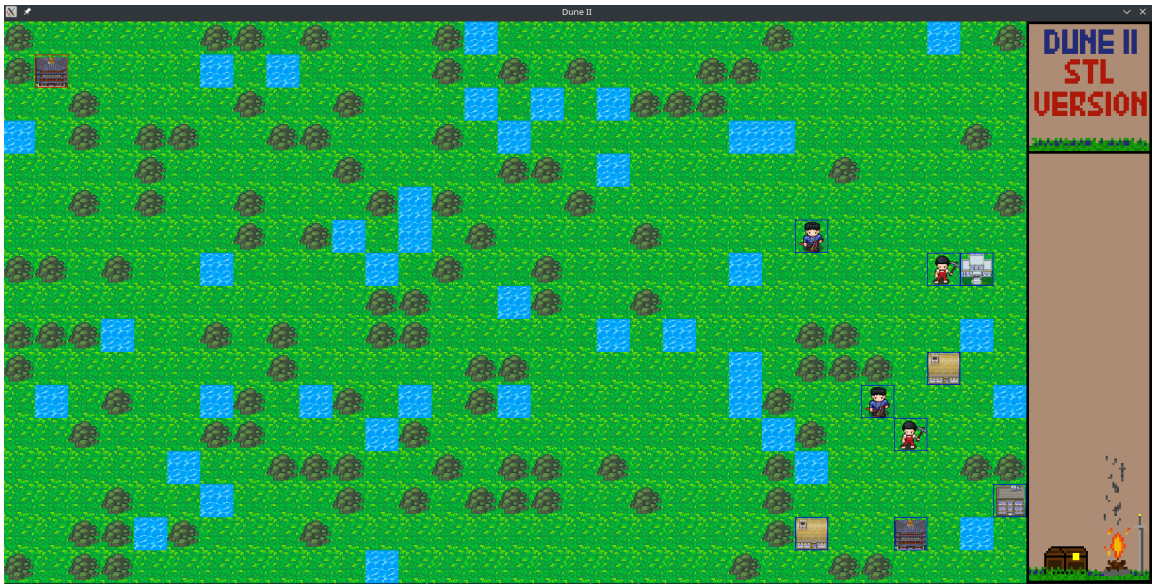

FIGURE 1 – Aperçu du jeu Dune II - STL version

Il n'y a donc pas "d'écran d'accueil" et le jeu commence directement après le lancement de la commande, avec 2 `QG` présents sur la carte comportant les 3 types de `Terrain` : `Eau`, `Ressources`, `Herbe`, facilement distinguables. Ces 2 `QG` opposés en diagonale représentent les 2 joueurs de la partie, l'un distingué avec un rectangle bleu creux sur ses `Batiments` et `Unites` et l'autre avec un rectangle de couleur rouge. Ceci nous permet de différencier les deux joueurs au cours de la partie. Le jeu peut être joué à deux, ce qui se justifie par le fait que toutes les `Unites` et `Batiments` peuvent être contrôlés en s'assurant que chaque joueur joue à tour de rôle, ou en "mode solo" ce qui explique la présence de `Batiments`, de `Combattants` patrouillant et de `Collecteur` collectant des ressources dès le début de la partie pour les entités avec un rectangle bleu. Ce dernier mode de jeu présente une "intelligence artificielle" assez limité, mais laisse l'occasion au joueur de tester l'ensemble des fonctionnalités du jeu tout en se confrontant à un adversaire.
En clinquant sur les différentes entités nous pouvons accéder à leur menu et interagir avec, comme nous pouvons le voir ci-dessus avec les différentes fenêtres des `Batiments` et `Unites` :

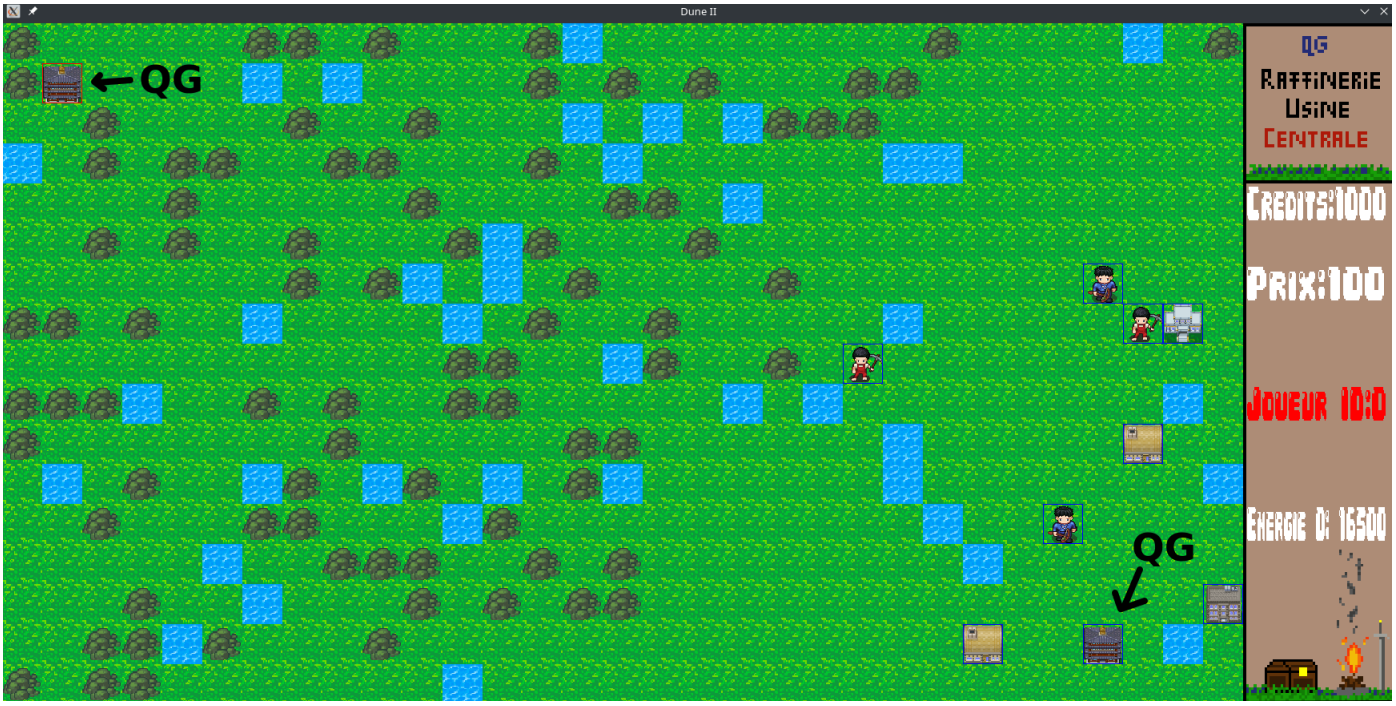

FIGURE 2 – Carte et menu du `QG`

Le menu du `QG` étant le plus important car s'est par lui que passent tous les achats de `Batiments`, comme nous pouvons l'observer plus haut, il est possible d'acheter 3 `Batiments` à partir de ce menu à condition d'avoir les crédits nécessaires. Ces trois `Batiments` sont : `Raffinerie`, `Usine` et `Centrale`. Le `QG` possede la particularité de fournir de l'énergie qui pourra être emmagasinée si les autres bâtiments ne la consomment pas en entier, à chaque frame (étape) le `QG` produit 50 unités d'énergie, visibles en dessous du menu.
Afin d'acheter un `Batiment`, il suffit de cliquer sur son nom dans le menu (l'écriture passe en rouge) et de cliquer à un endroit de la carte pour le placer. Pour placer `Centrale`, il suffit donc de la sélectionner et de la placer sur la carte comme ci-dessous :

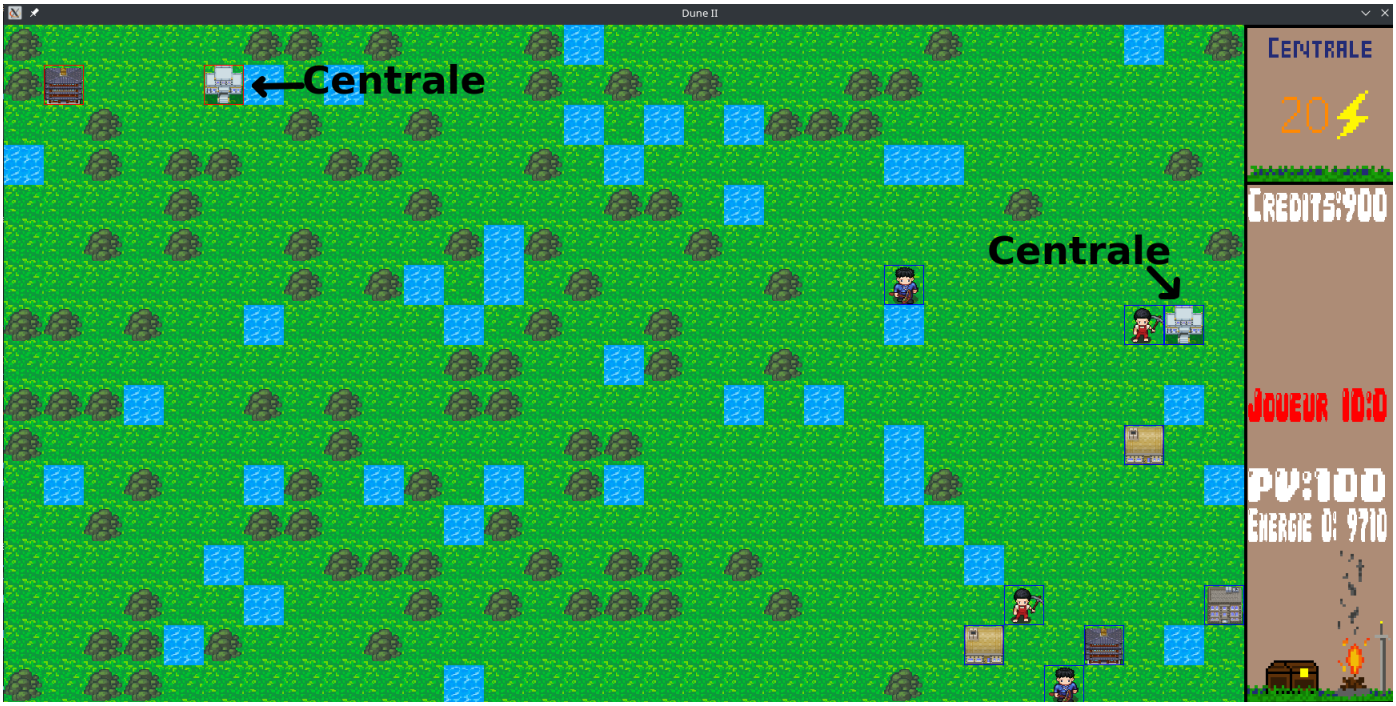

FIGURE 3 – Carte et menu de `Centrale`

Ce `Batiment` comme le `QG` fournit de l'énergie, plus exactement, 20 unités à chaque frame (étape). L'énergie étant essentielle pour préserver le fonctionnement des autres `Batiments`, la `Raffinerie` ne pourra plus convertir les `Ressources` en `Credits` et l'`Usine` ne pourra plus produire d'`Unites` sans cette dernière. La `Centrale` ne possede plus d'autres fonctionnalités dans cette version du jeu, d'où son menu sans option, comme pour la `Raffinerie` ci-dessous :

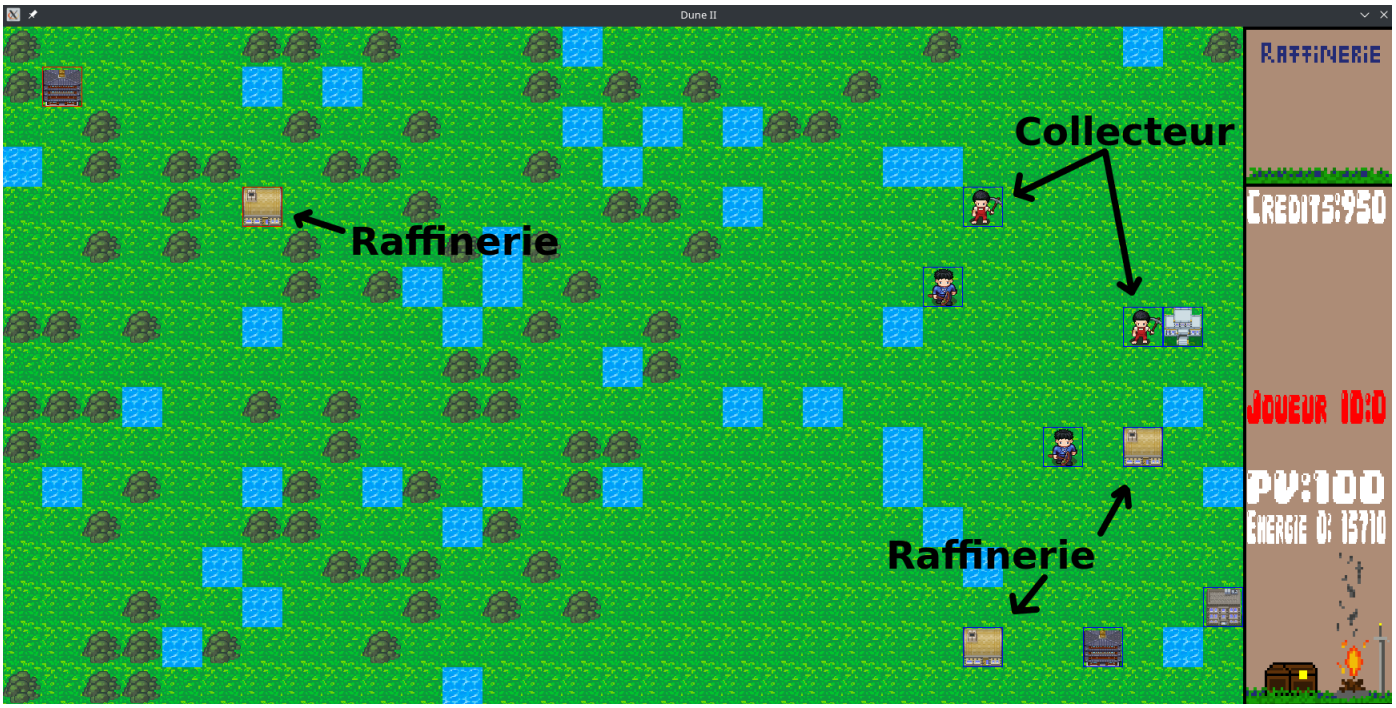

FIGURE 4 – Carte et menu de `Raffinerie`

La `Raffinerie`, comme la `Centrale` ne possede pas d'option dans son menu, cependant elle a une fonctionnalité très importante, celle de convertir les `Ressources` ramenés par les `Collecteurs` en `Crédits`, plus exactement une `Ressource` vaut 10 `Crédits`. Sans ce `Batiment` les `Collecteurs` remplissent leur `Cuve` (taille maximale de 1) en collectant et n'ont pas d'endroit ou ramener ce qu'ils ont collecté et ne peuvent donc plus collecter.
Cette `Unite` qui permet de collecter est crée par l'`Usine` :

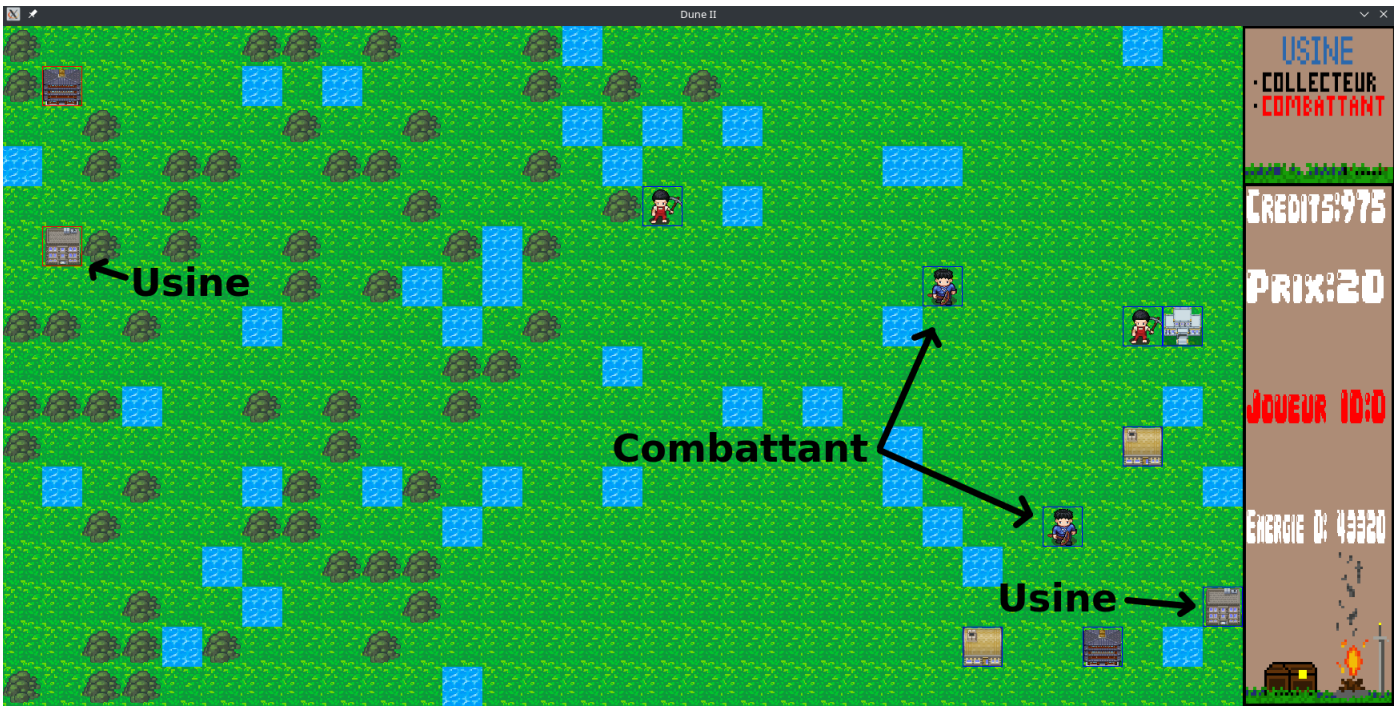

FIGURE 5 – Carte et menu de `Usine`

Ce `Batiment` a un rôle crucial puisqu'il permet de créer les `Unites` collectrices et combattantes. Ces dernières peuvent être crées comme les `Batiments` dans le menu du `QG` avec la particularité que après avoir sélectionné l'`Unite` souhaité, il faut choisir son propre `Batiment Usine`, cela veut dire que nous pouvons créer sur toutes nos `Usines` mais pas sur celles de notre adversaire. Une autre particularité est que l'`Unite` ainsi crée sera produite "devant" l'`Usine`, plus exactement en `y+1` d'où est situé le `Batiment`. Ainsi une `Usine` construite "derrière" l'eau n'a aucun bénéfice (n'étant pas vérifié par la pré-condition).
Ces `Unites` crées possèdent elles aussi un menu avec les `Ordres` possibles qu'elles peuvent effectuer :

FIGURE 6 – Carte et menus de `Combattant` et `Collecteur`

Les deux `Unites`, `Combattant` et `Collecteur`, possèdent la faculté de se déplacer (`Move` et `Deplacer`), ceci se fait en sélectionnant l'option puis en cliquant sur un endroit de la carte. Il existe toutefois des chemins non réalisables par les `Unites`, par exemple à cause de l'`Eau` qui les entoure, obligeant ainsi l'utilisateur de redonner un ordre de déplacement afin d'atteindre le point souhaité.
L'`Ordre` caractérisant l'`Unite Collecteur` est celui de collecter des `Ressources` et de les ramener à une `Raffinerie`, sous condition d'existence, pour conversion en `Crédits`. Il suffit donc de choisir un `Collecteur`, de sélectionner l'option `Collect` et cliquer sur une `Ressource` pour que la collecte se fasse (si pas de `Ressource` a cet endroit, l'`Ordre` est équivalent à un déplacement).
De l'autre côté, le `Combattant` se caractérise par la possibilité de `Patrouiller` et ainsi de combattre les autres `Unites` ou `Batiments` ennemis. Cette fonctionnalité se fait en sélectionnant le `Combattant`, puis l'option `Patrouiller` et enfin, comme pour tous les autres cas, cliquer sur un endroit de la carte. L'`Unite` se déplacera entre la case à laquelle elle était et la case sélectionné par l'utilisateur en permanence, si une `Unite` ou `Batiment` ennemi se trouve à une case près du `Combattant`, il attaquera et enlèvera à chaque frame (étape) un `pv` de l'ennemi.
Finalement, pour gagner il faut impérativement détruire le `QG` de l'adversaire, ceci se fait donc avec un `Combattant` qui va `Pattrouiller` à côté et le détruire en enlevant ses 100 `pv` (tous les `Batiments` on 100 de `pv` contre `30` pour les `Unites`). La destruction du `QG` ennemi produira l'écran de fin de partie affichant le gagnant :

FIGURE 7 – Écran de fin de partie

2 Programmation sûre

2.1 Liste de propositions

2.1.1 Proposition pour la carte

- **Invariant du terrain :** L'invariant du terrain se présente dans la fonction *prop_inv_terrain* dans notre projet. Dans cet invariant, on vérifie que la valeur de ressource soit supérieur à 0. Ce invariant est utilisé dans l'invariant de la carte.
- **Invariant de la carte :** L'invariant de la carte de présente dans la fonction *prop_inv_carte* dans notre projet. Dans cet invariant, d'abord on vérifie que chque valeurs dans le map respecte l'invariant du terrain, et puis on vérifie qu'Il existe au moins un chemin entre tous les deux points de la carte (ça veut dire que les chemin ne soit pas bloqué par l'eau).

On a aussi écrit les proposition pour la fonction *collect_case* pour que les collecteurs collectent la ressource.

- **Pré-condition pour *collect_case* :** Avant qu'on appelle cette fonction, on vérifie que la coordonné donnée dans l'argument est bien dans la carte.
- **Post-condition pour *collect_case* :** Après on appelle la fonction *collect_case*, notre vérification passe en trois étapes.
  — Si le nombre de ressource récupéré est 0. On vérifie que la carte n'a pas changé.
  — Si le nombre de ressource récupéré est supérieur à 0 et égale à le nombre de ressource demandé. On vérifie que le cas après la récupération est Herbe ou Ressource. Et le collecteur a bien récuperé ce nombre de ressource. Et les autres cas n'a pas changé.
  — Si le nombre de ressource récupéré est supérieur à 0 et inférieur à le nombre de ressource demandé. On vérifie que le cas après la récupération est Herbe. Et les autres cas n'a pas

changé.

2.1.2 Proposition pour l'environnement

Dans cette partie, on va introduire successivement les invariant on a créé pour chaque composition d'un environnement, y compris les joueurs, les bâtiments et les unité (l'invariant de la carte a été présenté dans la section avant). A la fin, on va présenter l'invariant de environnement.

- **Invariant pour joueur :** On vérifie que le nombre de n'est pas vide et le crédit de joueur est supérieur ou égale à 0.
- **Invariant pour bâtiment :** On vérifie que le nombre de bâtiment n'est pas vide et son point de vie est supérieur ou égale à 0.
- **Invariant pour unité :** On vérifie que le nombre de unité n'est pas vide.
- **Invariant pour environnement :**
  — Pour les joueurs, on vérifie que tous les joueurs respectent l'invariant de joueur et ont un id et nom unique.
  — Pour les bâtiment, on vérifie que tous les bâtiments respectent l'invariant de bâtiment, et que tous les bâtiments ont un proprio dans le jeu et que tous les coordonnés d'un bâtiment est dans la carte mais n'est pas l'eau ou ressource.
  — Pour les unités, on vérifie que tous les unités respectent l'invariant de unité, et que tous les unités ont un proprio dans le jeu et que tous les coordonnés d'un bâtiment est dans la carte mais n'est pas l'eau.
  — Pour la carte, on vérifie qu'elle respecte l'invariant de la carte.

2.1.3 Proposition pour les bâtiments

Dans cette partie, on va introduit les propositions on a fait pour les bâtiments. En tout d'abord on va introduire les propositions pour les quartier centrale. Dans le reste de la section, on va introduire successivement les propositions d'usine, de raffinerie et de centrale, y compris la pré/post-condition de leur construction et destruction.

- **Quartier Général**
  - **Pré-condition pour la destruction d'un quartier général :** On vérifie que le proprio de quartier général est dans le jeu, et l'environnement respecte son invariant.
  - **Post-condition pour la destruction d'un quartier général :** On vérifie que le proprio de quartier général et son bâtiment et unités est supprimé dans le jeu, et l'autre joueurs ainsi que son bâtiments et unités ne sont pas changé. On vérifie aussi que la carte reste la même.

- **Usine**
  - **Pré-condition pour la construction d'une usine :** On vérifie que le joueur est dans le jeu et son crédit est supérieur à le prix d'usine, la coordonné dans la carte est bien herbe et l'environnement respecte son invariant
  - **Post-condition pour la construction d'une usine :** On vérifie que le décrémentation du crédit de joueur est le prix d'usine, le bâtiment qui vient d'être construit est une usine, les autres joueurs et autres bâtiments sont pas changé, la carte (qui compris juste l'eau, herbe et ressource) et les unité ne sont pas changé.

- **Pré-condition pour la destruction d'une usine :** On vérifie que le joueur est dans le jeu et il est bien le proprio de ce bâtiment, et l'environnement respecte son invariant, et le bâtiment qui va être détruire est bien une usine,
- **Post-condition pour la destruction d'une usine :** On vérifie que ce bâtiment n'est plus dans l'environnement. On vérifie aussi que la carte, les joueurs, les unité et les autres bâtiment reste la même.

- **Raffinerie**
  - **Pré-condition pour la construction d'une raffinerie :** On vérifie que le joueur est dans le jeu et son crédit est supérieur à le prix de raffinerie, la coordonné dans la carte est bien herbe et l'environnement respecte son invariant
  - **Post-condition pour la construction d'une raffinerie :** On vérifie que le décrémentation du crédit de joueur est le prix d'usine, le bâtiment qui vient d'être construit est une raffinerie, les autres joueurs et autres bâtiments sont pas changé, la carte (qui compris juste l'eau, herbe et ressource) et les unité ne sont pas changé.
  - **Pré-condition pour la destruction d'une raffinerie :** On vérifie que le joueur est dans le jeu et il est bien le proprio de ce bâtiment, et l'environnement respecte son invariant, et le bâtiment qui va être détruire est bien une usine,
  - **Post-condition pour la destruction d'une raffinerie :** On vérifie que ce bâtiment n'est plus dans l'environnement. On vérifie aussi que la carte, les joueurs, les unité et les autres bâtiment reste la même.

- **Centrale**
  - **Pré-condition pour la construction d'une centrale :** On vérifie que le joueur est dans le jeu et son crédit est supérieur à le prix de centrale, la coordonné dans la carte est bien herbe et l'environnement respecte son invariant
  - **Post-condition pour la construction d'une centrale :** On vérifie que le décrémentation du crédit de joueur est le prix d'usine, le bâtiment qui vient d'être construit est une centrale, les autres joueurs et autres bâtiments sont pas changé, la carte (qui compris juste l'eau, herbe et ressource) et les unité ne sont pas changé.
  - **Pré-condition pour la destruction d'une centrale :** On vérifie que le joueur est dans le jeu et il est bien le proprio de ce bâtiment, et l'environnement respecte son invariant, et le bâtiment qui va être détruire est bien une usine,
  - **Post-condition pour la destruction d'une centrale :** On vérifie que ce bâtiment n'est plus dans l'environnement. On vérifie aussi que la carte, les joueurs, les unité et les autres bâtiment reste la même.

2.1.4 Proposition pour les unités

Dans cette section, notre présentation des proposition est divisé en deux partie. La première partie est les invariant des combattants et des collecteurs, ainsi que des nouveaux types qui apparaît dans cette partie, par exemple le cuve (comme ce lui dans TD1) qui est une partie de collecteur et les ordres. Dans la deuxième partie on va présenter les pré/post-conditions pour construire un collecteur ou un combattant. On avait aussi fait les proposition pour détruire une unité. Cependant, nous avons constaté que les fonctions de destruction écrites n'étaient pas applicables aux partie suivant. Donc nous avons réécrit les fonctions, mais à ce moment-là, il était presque le deadline du projet, nous n'avons pas eu le temps pour finir les propositions de destruction.

- **Invariants**
  - **Invariant de cuve :**
    — Si le cuve est vide, sa capacité est supérieur à 0.
    — Si le cuve est plein, sa capacité est supérieur à 0.
    — Si le cuve est non vide non plein, sa capacité et sa quantité sont supérieur à 0.
  - **Invariant de Ordre :** Si cet ordre est *Patrouiller*, les deux coordonné donné ne sont pas même.
  - **Invariant de Collecteur :** On vérifie que la unité et le cuve de ce collecteur respecte bien son invariant. On vérifie aussi tous les ordres dans la liste des ordres ainsi que le but respecte bien l'invariant d'ordre. A la fin, on vérifie que le point de vie est supérieur à 0.
  - **Invariant de Combattant :** On vérifie que la unité et le cuve de ce combattant respecte bien son invariant. On vérifie aussi tous les ordres dans la liste des ordres ainsi que le but respecte bien l'invariant d'ordre. A la fin, on vérifie que le point de vie est supérieur à 0.

- **Construction**
  - **Pré-condition pour la construction d'une collecteur :** On vérifie que la coordonné donné est bien dans la carte, et sur cette cordonné il y a bien une usine. On vérifie que le joueur donné est bien dans le jeu, et son crédit est supérieur à le prix de collecteur. On vérifie aussi que l'environnement donné respecte bien son invariant. A la fin, on vérifie que dans la liste de collecteur donné qui contient les collecteur qui est déjà existe, tous les éléments respectent bien l'invariant de collecteur.
  - **Post-condition pour la construction d'une collecteur :** On vérifie que les joueurs, la carte, les bâtiment ne sont pas changé après la construction. On vérifie aussi l'environnement après la construction respecte l'invariant d'environnement. En plus, on vérifie que ce qui a été construire est bien un collecteur. A la fin, On vérifie que le décrémentation du crédit de joueur est le prix de collecteur.
  - **Pré-condition pour la construction d'une combattant :** On vérifie que la coordonné donné est bien dans la carte, et sur cette cordonné il y a bien une usine. On vérifie que le joueur donné est bien dans le jeu, et son crédit est supérieur à le prix de combattant. On vérifie aussi que l'environnement donné respecte bien son invariant. A la fin, on vérifie que dans la liste de collecteur donné qui contient les combattant qui est déjà existe, tous les éléments respectent bien l'invariant de combattant.
  - **Post-condition pour la construction d'une combattant :** On vérifie que les joueurs, la carte, les bâtiment ne sont pas changé après la construction. On vérifie aussi l'environnement après la construction respecte l'invariant d'environnement. En plus, on vérifie que ce qui a été construire est bien un combattant. A la fin, On vérifie que le décrémentation du crédit de joueur est le prix de combattant.

2.2 Description des tests

2.2.1 Tests pour la carte

Pour faire les tests de la carte, on a crée une carte valide de 5×5.

```
carteEx :: Carte
carteEx = Carte (M.fromList [ (C 0 0, Herbe)
                    , (C 0 1, Eau)
                    , (C 0 2, Herbe)
                    , (C 0 3, Herbe)
                    , (C 0 4, Herbe)
                    , (C 1 0, Eau)
                    , (C 1 1, Herbe)
                    , (C 1 2, Ressource 9)
                    , (C 1 3, Herbe)
                    , (C 1 4, Ressource 8)
                    , (C 2 0, Herbe)
                    , (C 2 1, Herbe)
                    , (C 2 2, Ressource (4))
                    , (C 2 3, Herbe)
                    , (C 2 4, Eau)
                    , (C 3 0, Herbe)
                    , (C 3 1, Herbe)
                    , (C 3 2, Herbe)
                    , (C 3 3, Eau)
                    , (C 3 4, Herbe)
                    , (C 4 0, Eau)
                    , (C 4 1, Herbe)
                    , (C 4 2, Ressource 2)
                    , (C 4 3, Herbe)
                    , (C 4 4, Herbe)])
```

Listing 1 – "Carte valide"

- **Tests pour terrains**

  Pour les test des terrain, on a fait 3 tests. Pour le premier, on fait sur le cordonné (C 0 0) de la carte valide, qui est Herbe. Pour le deuxième, on le fait sur le cordonné (C 4 2) de la carte valide, qui est Ressource 2. Ces deux test doivent renvoyer True. Pour le troisième, on crée une carte non valide qui contient qu'un seul cas avec le nombre de ressource est négative. Ce test va renvoie False.

- **Tests pour l'invariant de la carte**

```
carteFaux :: Carte
carteFaux = Carte (M.fromList [ (C 0 0, Herbe)
                    , (C 0 1, Eau)
                    ...
                    , (C 5 5, Herbe)])
```

Listing 2 – "Test sur terrain"

Pour les test de l'invariant de la carte, on a créé une carte qui est invalide, qui contient (C 5 5) qui est inaccessible. Donc le première test on fait sur la carte valide, et deuxième sur ce lui qui est invalide.

- **Tests pour la pré-condition de *collect_ case***
  On a fait 4 tests pour pré-condition de fonction *collect_ case*. Les deux première doivent True. Pour le troisième test, on essaye de collecter des ressource sur une coordonné qui n'existe pas dans la carte. Pour le dernier, on essaye de collecter un nombre négative de ressource. Donc ces deux tests divent renvoyer False.

- **Tests pour la fonction *collect_ case***
  On a fait 4 tests pour la fonction *collect_ case*. Les trois premiers correspondant à la situation où le nombre de ressource demandé est supérieur, égale, inférieur à le nombre de ressource dans le cas. Et le dernier on a essayer de collecter le ressource dan l'herbe.

- **Tests pour la fonction *collect_ case***
  On a fait 4 tests pour la post-condition de la fonction *collect_ case*, qui prends les situation correspondante dans la dernier test.

2.2.2 Tests pour l'environnement

- **Tests pour l'invariant des joueurs**
  On a fait 2 tests pour l'environnement, une test valide et un test que le nom de joueur est vide.

- **Tests pour l'invariant des bâtiments**
  Pour effectuer le test de bâtiment, on a fait 3 test, avec deux tests qui est valide et un test que le nom de bâtiment qui est vide donc invalide.

- **Tests pour l'invariant des unités**
  On a fait deux tests pour tester l'invariant des unités. Le premier test est valide et le deuxième test va obtenir un résultat False parse que cet unité a un nom vide.

- **Tests pour l'invariant de l'environnement**
  Pour effectuer le test pour l'environnement, douze tests ont été effectués pour simuler le plus fidèlement possible tous les scénarios. Par exemple, on a des environnement dont les joueurs qui ont le même id, qui a les bâtiments sur le ressource, ou des unité sur l'eau.

2.2.3 Tests pour les bâtiments

- **Tests pour *smartEnv***
  On prends le carte valide (voir Listing 1) et crée une liste de coordonné pour faire le test de fonction *smartEnv*.

- **Tests pour quartier général**
  Pour le quartier général, on a fait 3 tests successivement pour la pré-condition de destruction, la post-condition de destruction et la fonction destruction.

- **Tests pour raffinerie**
  - **Pré-condition de construction** Pour tester la pré-condition de construction d'un raf-

finerie, on a pris un cas valide, et un cas que le joueur essaye de construire la raffinerie sur l'eau.

- **Construction** On a pris un exemple valide pour tester la fonction de construction.
- **Post-condition de construction** On a pris un exemple valide pour tester la post-condition de constuction d'une raffinerie.
- **Pré-condition de destruction** Pour tester la pré-condition de destruction d'un raffinerie, on a pris 3 exemples en total, avec un exemple valide, un exemple que le joueur donné en paramètre ne correspond pas le proprio de la raffinerie et le dernier exemple que le bâtiment qui va être détruit est un quartier général.
- **Destruction** On a pris un exemple valide pour tester la fonction de destruction.
- **Post-condition de destruction** Comme toujours, on a pris un exemple valide pour tester la post-condition de destruction d'une raffinerie.

- **Tests pour usine**
  - **Pré-condition de construction** Pour tester la pré-condition de construction d'une usine, on a pris 2 exemple. Le premier exemple est valide. Et dans le deuxième exemple, le joueur essaye de construire une usine sur l'eau
  - **Construction** On a pris un exemple valide pour tester la fonction de construction.
  - **Post-condition de construction** On a pris un exemple valide pour tester la post-condition de constuction d'une usine.
  - **Pré-condition de destruction** Pour tester la pré-condition de destruction d'un raffinerie, on a pris 3 exemples en total. Le premier exemple est valide, cependante, dans l'autre exemple, le bâtiment passé en paramètre est un quartier général.
  - **Destruction** On a pris un exemple valide pour tester la fonction de destruction.
  - **Post-condition de destruction** On a pris un exemple valide pour tester la post-condition de destruction d'une usine.

- **Tests pour centrale**
  - **Pré-condition de construction** On a pris deux cas pour tester la pré-condition de la construction, un exemple valide, et un exemple que le joueur essaye de construire la centrale sur l'eau.
  - **Construction** Pour tester la fonction de construction d'une centrale, on a pris un example valide.
  - **Post-condition de construction** On a pris un exemple valide pour tester la post-condition de constuction d'une usine.

- **Pré-condition de destruction** Pour tester la pré-condition de destruction d'une centrale, on a pris 3 exemples en total. Le premier exemple est valide, cependante, dans l'autre exemple, le bâtiment passé en paramètre est un quartier général.

- **Destruction** On a pris un exemple valide pour tester la fonction de destruction.

- **Post-condition de destruction** On a pris un exemple valide pour tester la post-condition de destruction d'une usine.

2.2.4 Tests pour les unités

- **Tests pour l'invariant de cuve**
  On a fait trois tests en total pour le cuve, successivement pour le cuve vide, le cuve plein et le cuve normal.

- **Tests pour l'invariant de l'ordre**
  Pour le test de l'invariant de l'ordre, on a passé deux tests, un test valide, et un test pour un ordre de *Patrouiller* où les deux coordonné est la même.

- **Tests pour l'invariant du collecteur**
  On a passé trois tests pour vérifier l'invariant du collecteur. Le premier test est valide. Dans le deuxième test, le point de vie du collecteur est négative. Dans le troisième test, le collecteur a un ordre *Patrouiller*. Cependant, le collecteur ne peut pas avoir ce genre d'ordre comme il est un ordre pour combattant.

- **Tests pour l'invariant du combattant**
  Pour vérifier l'invariant du combattant, on a passé 2 tests, un test valide et dans l'autre test le combattant a un ordre *Collecter*.

- **Tests pour la construction d'un collecteur**
  - **pré-condition**
    On a utilisé trois exemple pour faire ce test de pré-condition de construction de collecteur. Le premier exemple est valide et les deux autres sont invalides. Dans le deuxième exemple, il n'y a pas de usine dans l'environnement. Dans le dernier exemple, il y a des usines dans l'environnement mais il n'appartient au joueur qui essaye de construire un collecteur.

  - **Construction**
    Pour éffectuer ce test, on a passé un test valide. On a passé des paramètre dans la fonction et on a crée le résultat manuellement pour finir ce test.

  - **post-condition**
    On a pris un cas valide pour finir ce test de post-condition d'un collecteur.

- **Tests pour la construction d'un combattant**

- **pré-condition**
  Pour effectuer ce test de pré-condition de construction de combattant, on est passés trois test comme ce lui de collecteur. Pour le premier test, on est passés un test valide, qui respecte tous les sous-condition dans la pré-condition. Pour le deuxieme, on a créé un environnement pour passer le test, donc ce test doit obtenir un résultat False. Et pour le troisième test, on est passés un environnement dans lequel le joueur qui essaye de construire le combattant n'a pas d'usine. Doc évidement ce test obtient un résultat False ;

- **Construction**
  Pour éffectuer ce test, on a passé un test valide. On a passé des paramètre dans la fonction et on a crée le résultat manuellement pour finir ce test.

- **post-condition**
  On a pris un cas valide pour finir ce test de post-condition d'un combattant.

- **Tests pour la destruction des unité** La destruction des unités est un peu différent à construction. Pour construction, on construit l'unité un par un. Mais pour la destruction, on parcours la liste qui contient tous les collecteur et celui qui contient tous les combattant, et supprime ce lui avec un point de vie qui est 0, et aussi supprime dans l'environnement.
  Pour effectuer la destruction des unités, on a fait trois test en total. Le premier cas, les unité sont toutes avoir un point de vie supérieur à 0, donc on va avoir des même liste. Pour le deuxième exemple, on supprime tous les unité qui est morte dans deux listes. Dans le troisième exemple, il y a que des unité mortes dans la liste de combattant, donc la liste de collecteur reste la même.

2.2.5 Test pour le Deplacement des unité

On est passés 4 tests pour le deplacement. Le premier test, tous les unité a deplacé un cas et il n'y a pas d'autre changement. Dans la deuxième test, il y a des unité qui arrive son destinataire donc il y a des changement dans sa liste des ordres. Dans la troisième test, il y a un collecteur qui est bloqué, donc il deplace pas et il y a des changement dans sa liste des ordres. Dans le dernier cas, il y a une unité qui est devant de l'eau, donc il va changer sa direction.

3 Extensions

Après l'implémention du cahier des charges, nous nous sommes focalisé sur l'implémention d'éventuelles extensions, comme une génération de `Carte` de la manière la plus aléatoire possible ou une combinaison de modes de jeu (jeu solo et jeu à deux).

3.1 Génération de carte

Pour générer une carte "aléatoire" utilisable par les joueurs, nous avons utilisé le pseudo-aléatoire vu dans le TME1, pour nous éviter d'entrer dans l'IO et d'utiliser des générateurs.

```
1  -- | "Graine pseudo-aleatoire"
2  randomNb :: Integer -> [Integer]
3  randomNb seed = iterate (\x -> (25210345917 * x + 11) `mod` (2^48)) seed
4
```

```haskell
-- | Generation d'une carte pseudo-aleatoire en prenant un entier en argument
genCarte :: Integer -> Carte
genCarte nb = Carte $ M.fromList [((C x y), getTerrain x y) | x <- [0..30], y <- [0..16]]
  where
    getTerrain x y
      | r < 0.2 = Ressource 1 -- 1 Ressource par case pour aller plus vite
      | r < 0.3 = Eau
      | otherwise = Herbe
      where
        r = fromIntegral (randomNb nb !! (x * 17 + y)) / fromIntegral (2^48)
```

Listing 3 – "Code de la génération pseudo-aléatoire de la `Carte` à partir d'un numéro"

Ceci nous a permis d'avoir une carte valide (voir captures d'écrans précédentes) et respectant l'invariant de carte en fournissant l'entier 2 en argument.

3.2 Jeu solo

Malgré le fait que l'utilisateur puisse accéder aux commandes de l'autre joueur (entités avec des rectangles bleus), le mode solo peut être pratiqué. L'initialisation au début de 2 `Raffineries`, une `Centrale`, une `Usine`, deux `Combattants` qui "protégeant" en patrouillant le `QG` et 2 `Collecteur` dont l'un collecte les `Ressources` à proximité, permet une partie courte en mode solo.

3.2.1 "Intelligence artificielle simple"

Pour que le `Collecteur` puisse collecter les `Ressources` et les ramener à la `Raffinerie` nous avons implémenté un algorithme qui parcours tous les `Ressources` disponibles et donne comme ordre à l'`Unite` de se diriger vers la plus proche, puis de se diriger vers la `Raffinerie` la plus proche.

```haskell
-- | Calcul la distance euclidienne en gardant les carres pour plus d'efficacite
distance :: Coord -> Coord -> Int
distance (C x1 y1) (C x2 y2) = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2)

-- | Donne au premier collecteur du JoueurId 1 (PC) l'ordre de collecter la Ressource la plus proche
get_set_Ressource_closest :: [Collecteur] -> Environnement -> [Collecteur]
get_set_Ressource_closest listeCollecteurs (Environnement joueurs (Carte mapp) unites bats) =
  if (length listeCollecteurs > 0) then
    let listeCollectsProprio = (List.filter (\(Collecteur _ (Unite _ uCoord uProprio) _ _ _ _) -> uProprio == (JoueurId 1)) listeCollecteurs)
    in
      if (length listeCollectsProprio > 0) && (not (has_but (head listeCollectsProprio))) then
        let permierCollect@(Collecteur _ (Unite _ uCoord _) _ _ _ _) = head listeCollectsProprio
        in
          let coordsRessources = M.keys $ M.filterWithKey (\_ terrain ->
              case terrain of
                  Ressource _ -> True
                  otherwise -> False) mapp
          in
```

```
19              if (coordsRessources /= []) then
20                let coordClosest = foldr (\coords acc -> if (Lib.distance coords
                      uCoord) < (Lib.distance acc uCoord) then coords else acc) (
                      head coordsRessources) coordsRessources
21                in
22                  (set_ordre_collect_collecteur (get_Collecteur_from_list uCoord
                        listeCollecteurs) listeCollecteurs coordClosest)
23              else listeCollecteurs
24        -- partie gerant certains blocages de deplacement
25        else if (length listeCollectsProprio > 0) && (is_blocked mapp (head
              listeCollectsProprio)) then
26        -- a voir directement dans le code
27        else listeCollecteurs
28    else listeCollecteurs
```

Listing 4 – "Code de l'algorithme de collecte de `Ressources` par l'ordinateur en "mode solo""

Nous avons donc utilisé la formule de distance euclidienne pour déterminer les `Ressources` et `Raffineries` les plus proches. Comme il y a des chemins inaccessibles à cause des cases d'`Eau`, des cas particuliers ont été prévus pour donner un `Ordre` de déplacement afin de "débloquer" le `Collecteur` pour qu'il puisse continuer la collecte.

3.2.2 Étape de jeu

La fonction étape est appelé a chaque frame et correspond a la fonction qui gère les `Ordres` de déplacement, de patrouille, de collecte des différentes `Unites`. Cette dernière vérifie également si des `Unites` ou `Batiments` n'ont plus de `pv` pour les éliminer de l'`Environnement` et des listes de `Collecteurs` et `Combattants`.

```
1  -- | Fonction qui gere le bon fonctionnement des Ordres des Unites et applique
       les suppressions des Unites/Batiments "morts"
2  etape :: Environnement -> [Collecteur] -> [Combattant] ->  M.Map JoueurId Int
       -> (Environnement, [Collecteur], [Combattant])
3  etape env listCollecteurs listeCombattants  ener =
4      let (env_resDep, list_collecteurDep, list_combattantDep) =
           deplacer_Unite_Cood env listCollecteurs listeCombattants
5      in
6          let (list_collecteurRes, list_combattantRes, envRes) =
              collecter_Collecteur_patrouiller_Combattant_coord env_resDep
              list_collecteurDep list_combattantDep ener
7          in
8              let (envRes2, listColl2, listCom2) = verifie_unites envRes
                  list_collecteurRes list_combattantRes
9              in
10                 let (envRes3, listCom3, listCol3) = eliminer_bats_pv_null
                       envRes2 listCom2 listColl2
11                 in
12                     (envRes3, listCol3, listCom3)
```

Listing 5 – "Code de la fonction `etape` gérant le jeu toutes les frames"

Ces listes font le lien avec la `M.Map UniteId Unite` à travers l'`UniteId` pour permettre l'accès à d'autres champs comme les `Ordres`, `pv`, `But` demandés dans le cahiers des charges après l'ajout de `Unite`. Ces dernières sont mises à jour au même moment que les `Unites` correspondantes dans l'`Environnement` et inversement.

Un bug persiste toutefois dans ce mode de jeu, à savoir, le combat entre deux `Combattants` peut impliquer une mort des deux à la même étape ce qui cause un probleme de mise à jour de `Sprites` et `Textures` bloquant ainsi l'achat futur d'autres `Unites`. Dans ce cas précis, un redémarrage du jeu est nécessaire.

4 Discussion

La réalisation d'une refonte du jeu Dune II à été mené a bien en utilisant le style fonctionnel proposé par Haskell et la programmation sûre avec les différents invariants, pres/post conditions et en testant ces derniers dans `Spec.hs`. Les tests nous ont permis d'identifier plus facilement et assez tôt dans le développement du jeu les potentielles bugs ainsi que problèmes de conception. La bibliothèque SDL nous a également permis de fournir une interface graphique pouvant montrer le bon fonctionnement des fonctions implémentés ainsi que deux scénarios de jeux qui sont le "mode solo" (avec une difficulté très facile contre l'ordinateur) et le "mode 1 vs 1" ou chaque utilisateur joue chacun son tour.
Nous avons également rencontré des difficultés, notamment en termes de temps, ce qui nous a contraint a repousser la date de soumission après les examens et de ne pas implémenter les tests basés sur les propriétés avec le `QuickCheck` et d'ajouter d'autres extensions. Également la complexité du développement d'un jeu de cette taille dans un langage comme Haskell nous a poussé a nous documenter d'avantage pendant l'implémentation, ce qui à causé un ralentissement.