Projet Orienté Objet - Rapport de modélisation

Labadie Lucie - Barbedor Franck

November 2016

Table des matières

Table des figures

Introduction

Le but du projet est de créer un jeu de stratégie à deux joueurs. Le projet se découpe en quatre parties : la modélisation, la génération aléatoire de la carte en *C++*, l'écriture des règles en *C#* et l'interface utilisateur en *WPF*.
Ce rapport présente les différents diagrammes associés à la partie *C#*, ainsi qu'une modélisation d'à quoi pourra ressembler notre logiciel final.

Diagramme de classes

Pour nous aider, un certain nombre de patrons de conceptions sont imposés, nous en avons aussi utilisé d'autres. Ici sont expliquées leurs fonctions et leur utilité.

1.1 Fabrique

Une fabrique est un objet dédié à la création d'autres objets. Dans notre projet elle s'appelle *UnitsFactory*. Ainsi la fabrique permet de créer toutes les unités du joueur en fonction de la race choisie.

1.2 Monteur

Un monteur sert à créer des objets complexes. La carte est un objet composé de nombreux autres objets ; l'utilisation du monteur peut être intéressante pour la création de celle-ci.
Nous avons donc choisi d'implémenter la classe *BuilderMap* qui permet de facilement monter une *GameMap*, c'est à dire de la créer puis de la remplir.

1.3 Poids-Mouche

Un poids-mouche est un patron de conception qui permet de de créer une multitude de petits objets en utilisant moins d'espace mémoire. Dans le diagramme de classe ces objets sont appelés *Tile*.

1.4 Stratégie

Une stratégie permet de s'adapter au contexte, en changeant d'implémentation en fonction de celui-ci. Dans ce projet on remarque deux fonctions qui correspondent, celle qui déplace les unités et celle qui compte les points. En effet en fonction de leur race les unités ne se déplacent pas de la même façon et ne rapportent pas le même nombre de points, suivant la case sur laquelle ils se trouvent. Pour ces deux fonctions, l'utilisation d'un stratégie est requise, c'est l'interface *Race* implémentée par *Centaurs*, *Cerberus* et *Cyclops*.

1.5 Commande

Une commande sert à encapsuler des méthodes, nous avons choisi les méthodes de gestion de partie telles que *NewGame*, *Save*, *Load*, *Exit*... Les classes-commandes implémentent l'interface *CommandMenu*.

1.6 Memento

Le memento est utile lorsqu'il faut sauvegarder l'état d'un objet à un instant donné, sans occuper trop d'espace mémoire. Pour les fonctions *play* et *save* du jeu nous avons pensé à utiliser un memento. De cette manière *save* peut garder l'état de la partie à un instant t pour qu'elle soit reprise plus tard et *replay* sauve tous les états d'une partie pour la rediffuser intégralement plus tard. Notre memento est appliqué sur les commandes qui implémentent l'interface *ToCollect*. Les états sont collectés par la classe *ReplayCollector*.

1.7 Singleton

Le singleton permet d'instancier une classe un seule fois. Ce patron de conception est intéressant notamment pour les fabriques et les monteurs, dans notre cas : *UnitsFactory* et *BuilderMap*.

1.8 Diagramme

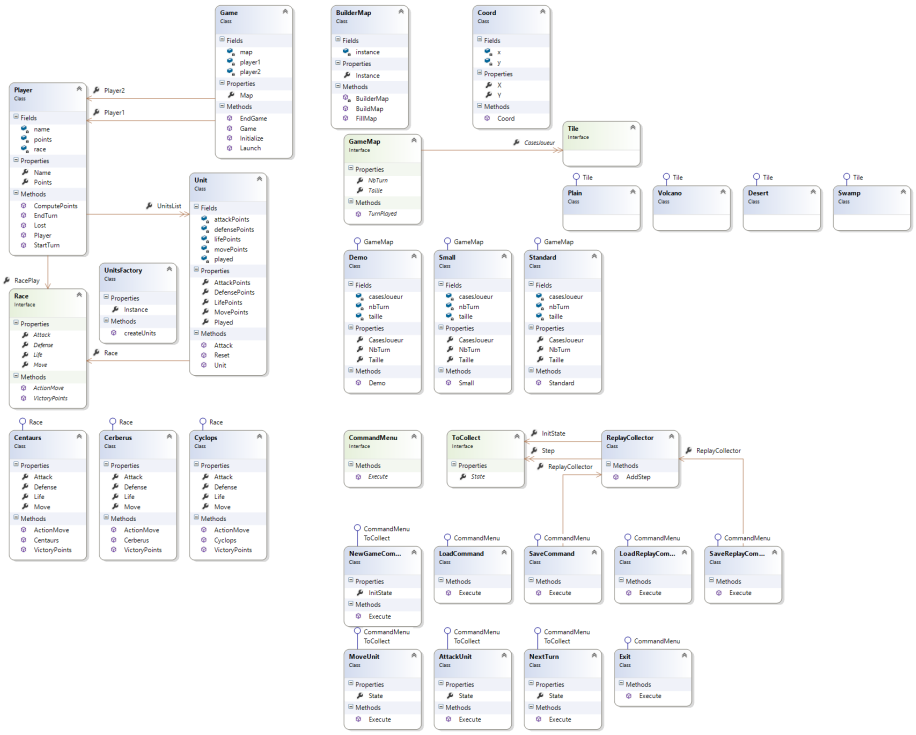

FIGURE 1.1 – Diagramme de classe

Diagrammes de séquence

Un diagramme de séquence est utilisable en parallèle du diagramme de classe, afin d'avoir une meilleure idée du fonctionnement concret du jeu et mieux définir les méthodes à utiliser. Nous avons choisi de travailler sur trois diagrammes de séquences.

2.1 Lancement d'une partie

Le premier diagramme porte sur la création d'un partie.

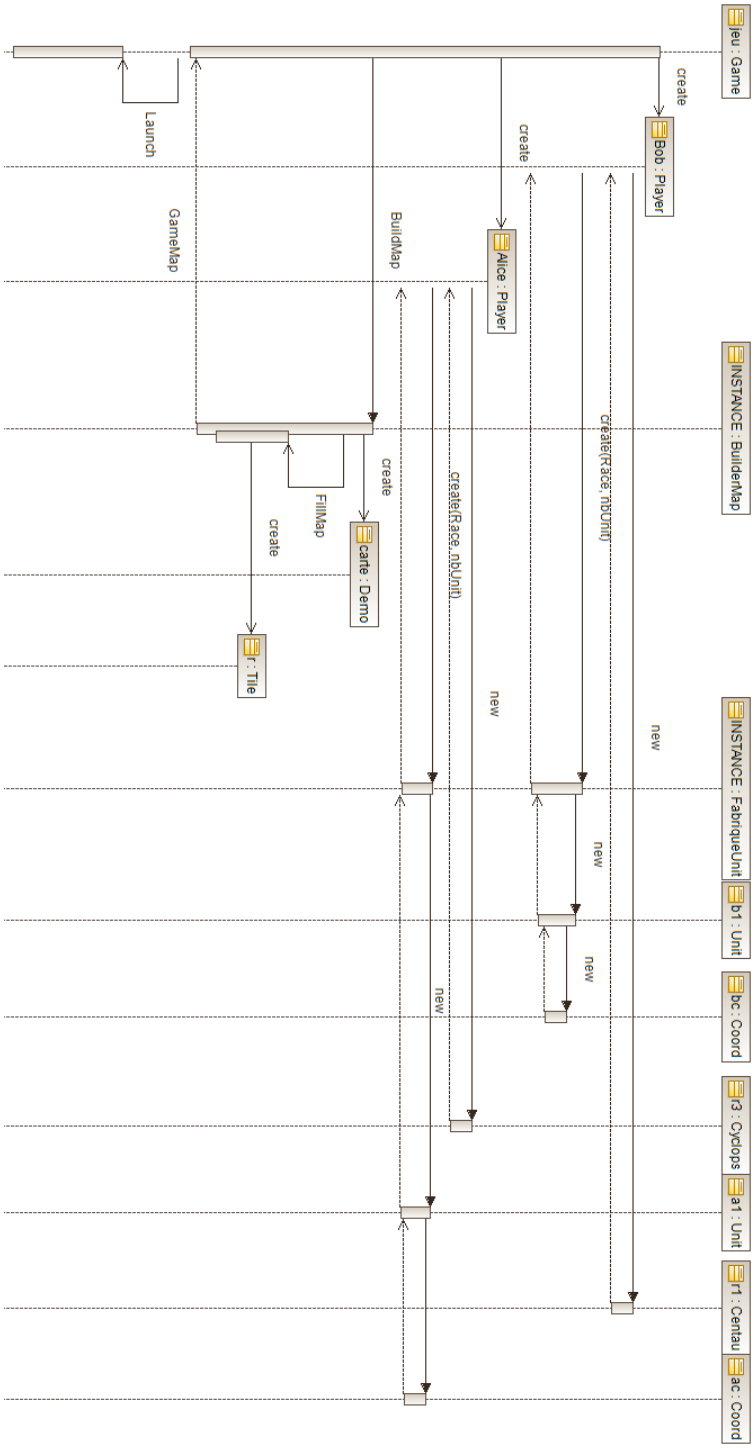

FIGURE 2.1 – Diagramme de séquence présentant le lancement d'une partie.

2.2 Déroulement d'une partie

Le second diagramme présente le déroulement d'un partie dans laquelle personne n'attaque, et sa fin. Par simplicité nous avons raccourci la partie à un seul tour.

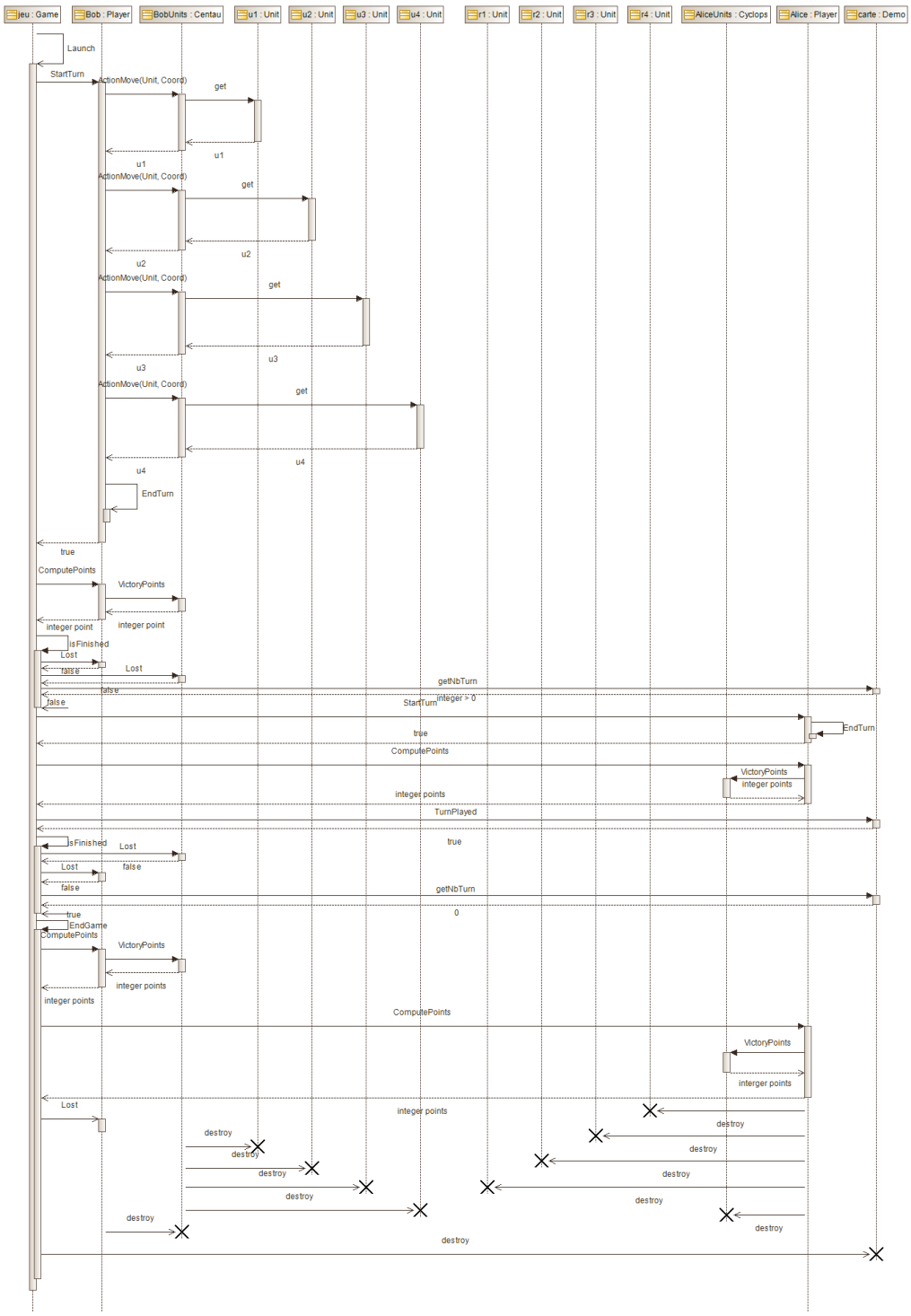

FIGURE 2.2 – Diagramme de séquence présentant le déroulement et la fin d'une partie par victoire d'un joueur. 9

2.3 Mécanisme d'attaque

Le troisième et dernier diagramme précise le mécanisme d'attaque d'une unité.

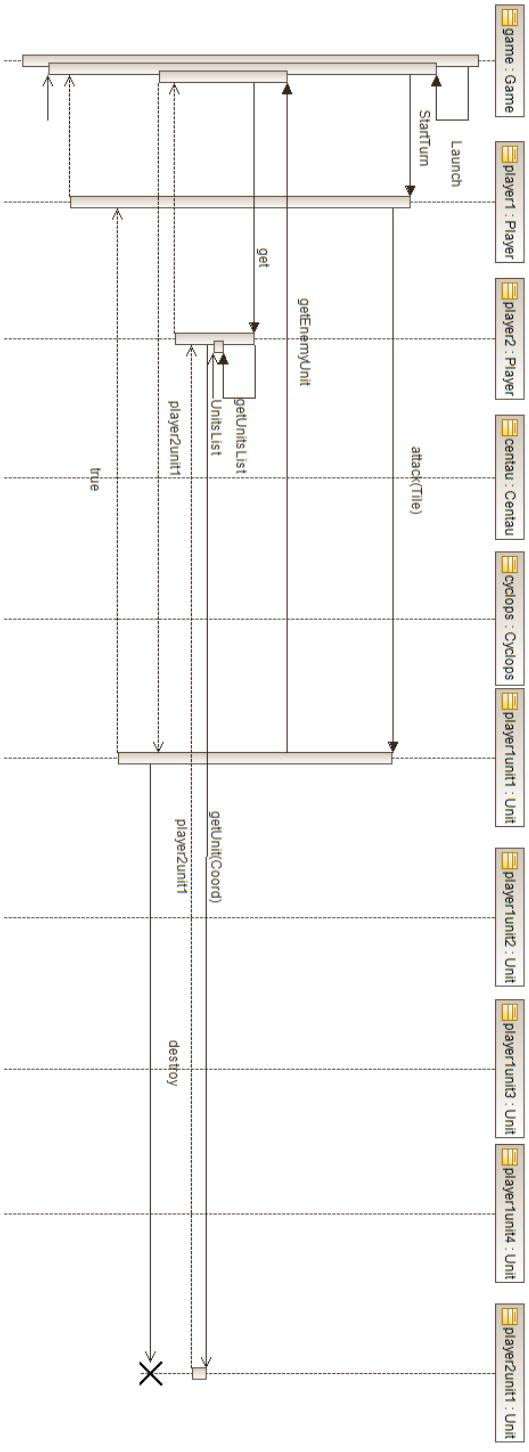

FIGURE 2.3 – Diagramme de séquence présentant le déroulement d'un combat entre 2 unités.

Diagramme d'état transition

Le diagramme d'état transition permet de visualiser les possibilités d'actions d'un objet ainsi que son cycle de vie. Nous avons choisi de représenter le diagramme d'état transition d'une partie en général.

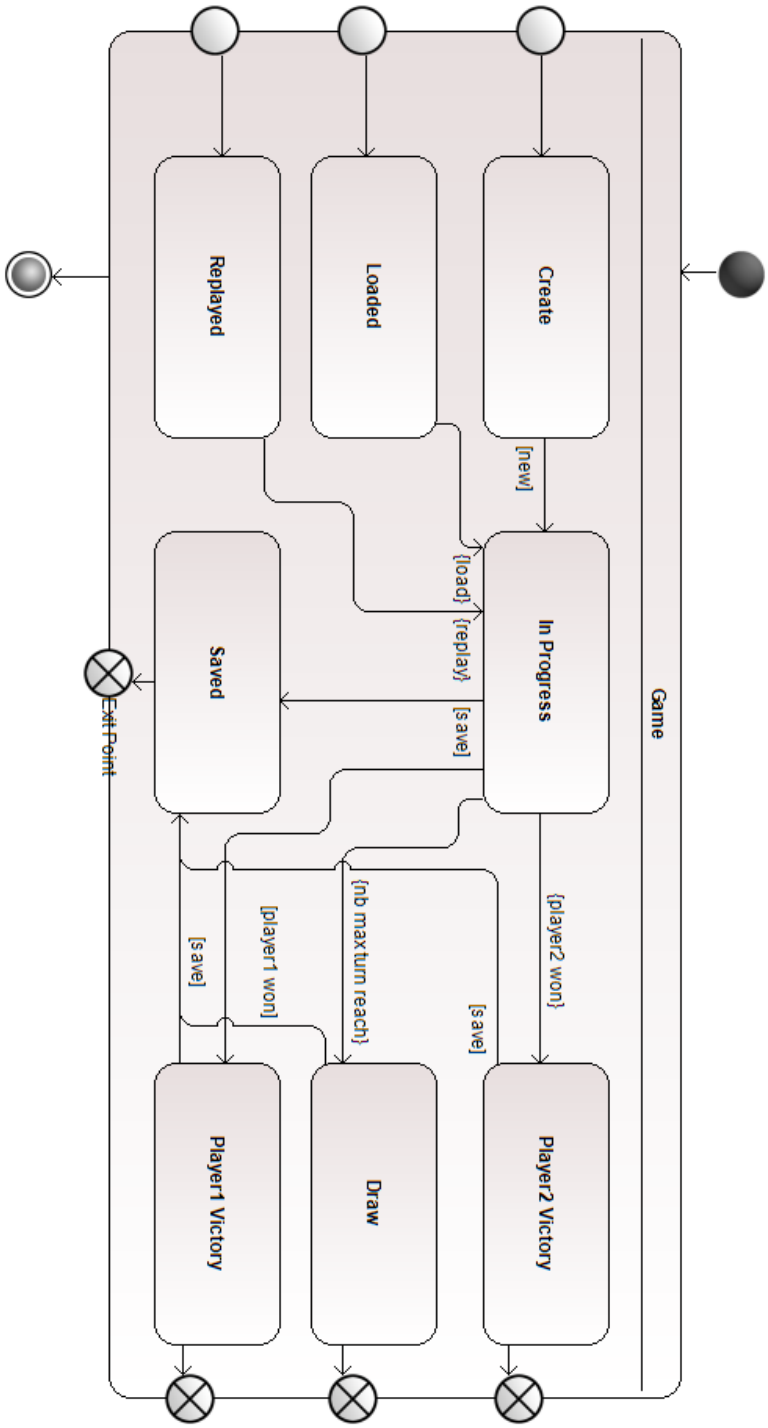

FIGURE 3.1 – Diagramme d'état transition d'une partie.

Mock-up

Le mock-up permet de mieux visualiser l'apparence du logiciel mais aussi vérifier qu'aucune des fonctions nécessaires au bon déroulement d'une partie pour des joueurs ne manque.

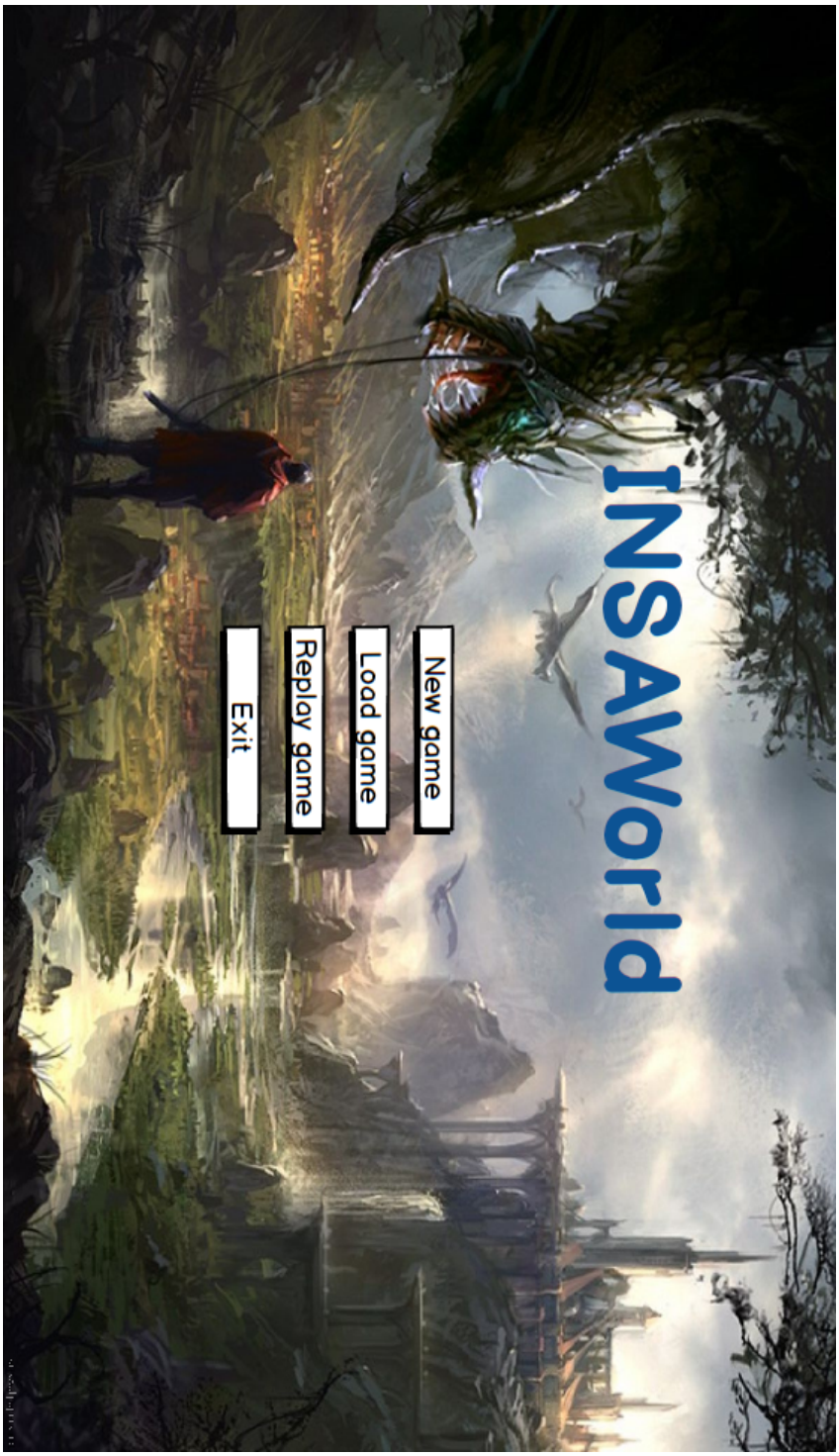

FIGURE 4.1 – Écran principal et menu principal

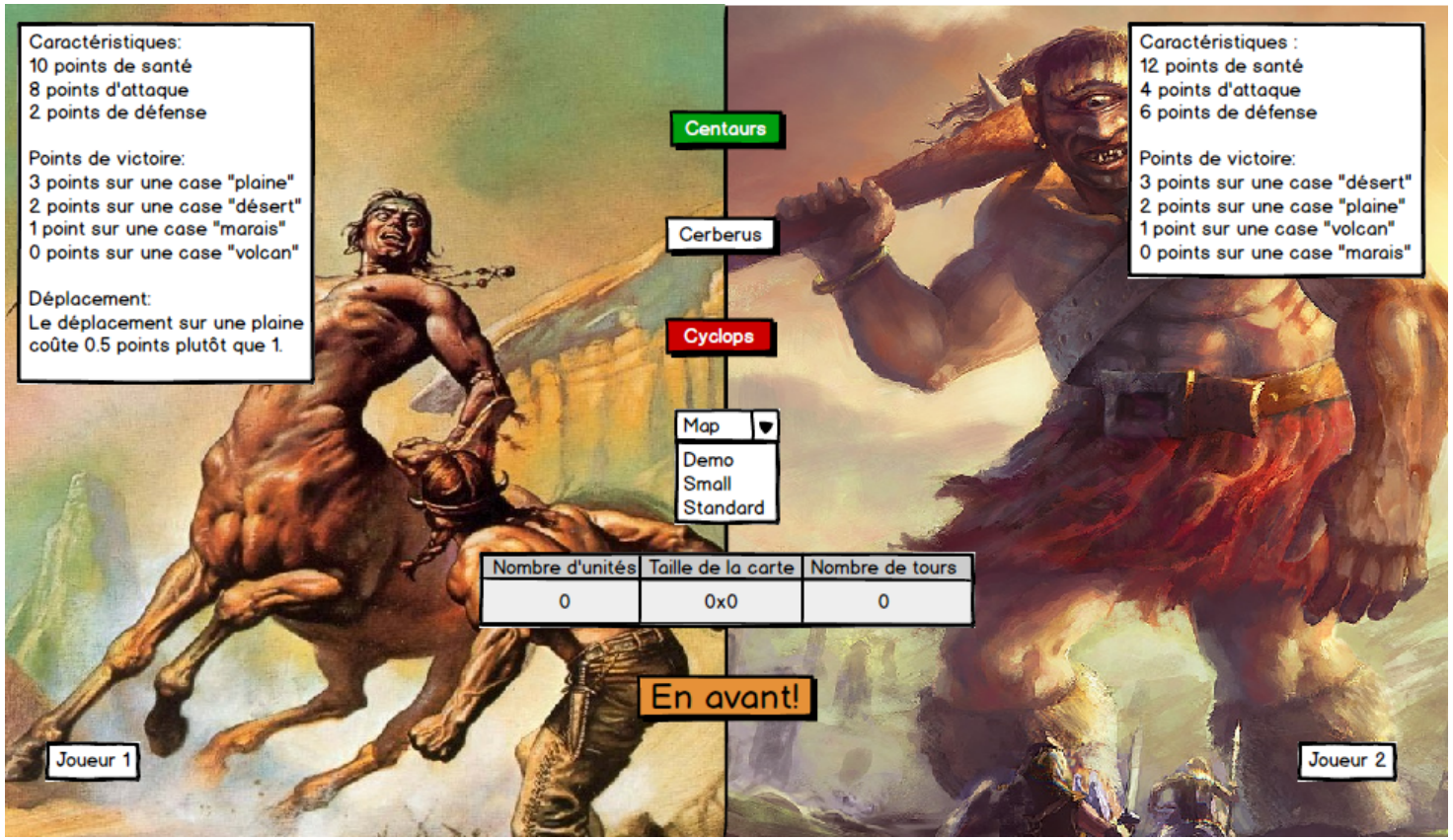

FIGURE 4.2 – Choix de la race et de la carte

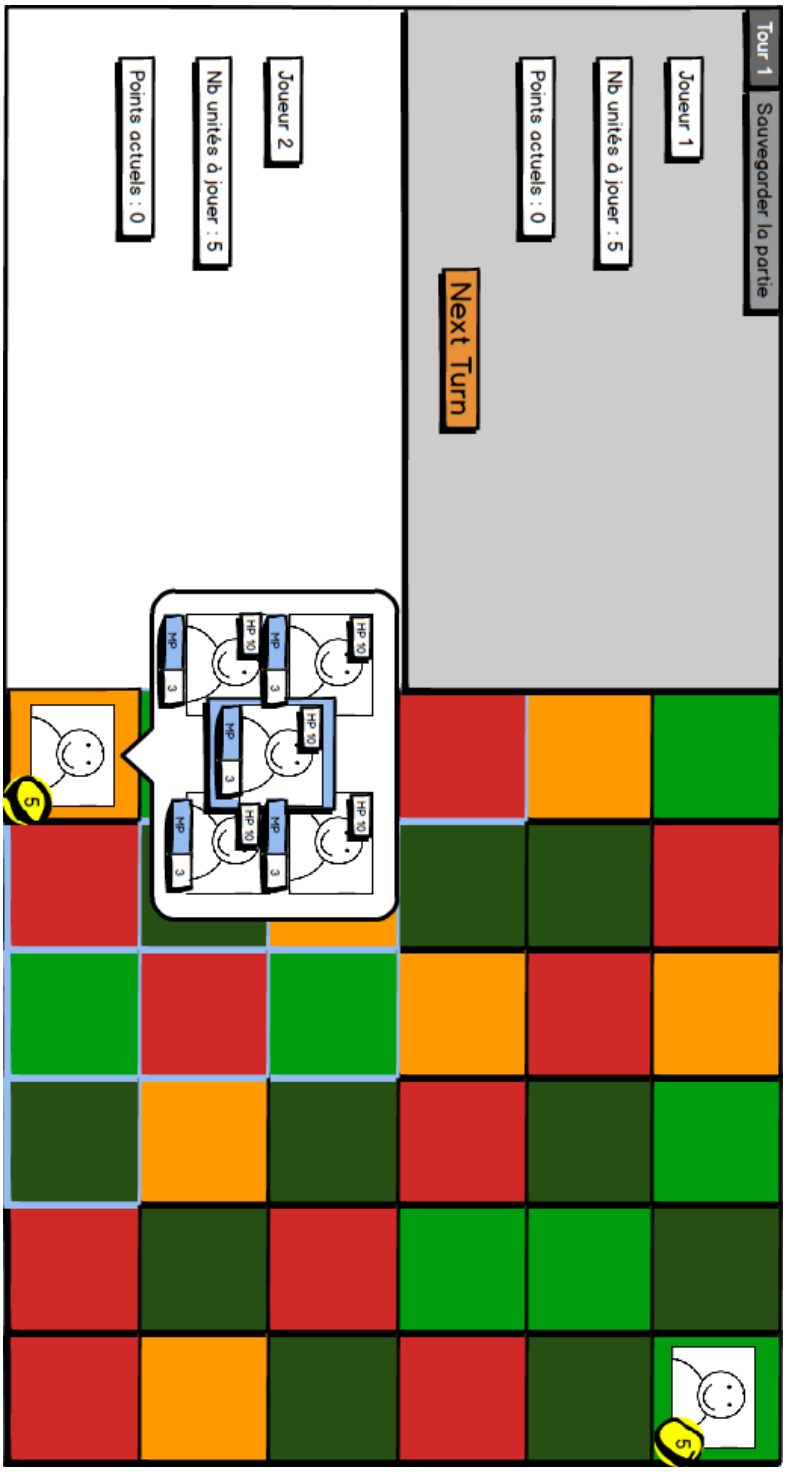

FIGURE 4.3 – Présentation du jeu pendant une partie

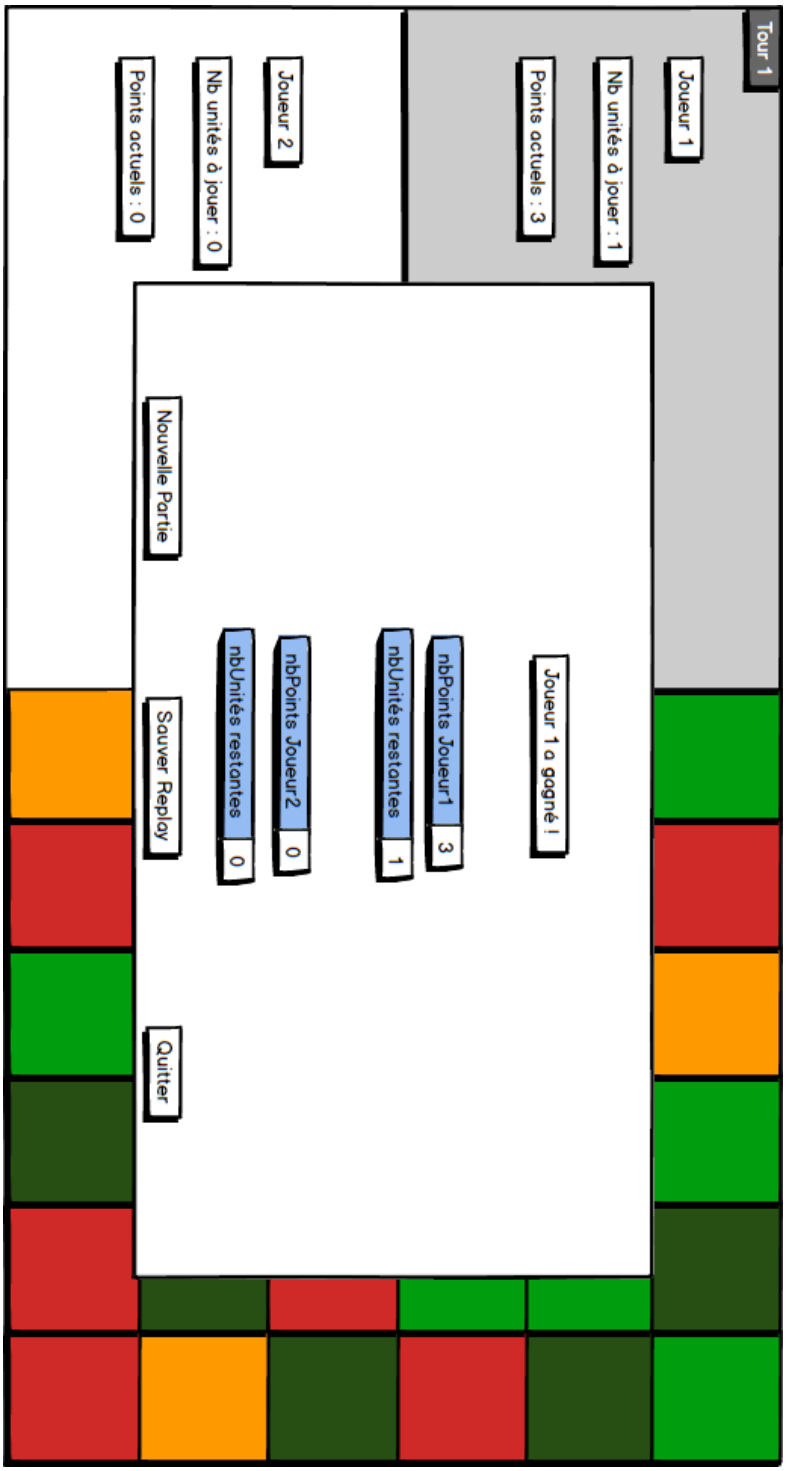

FIGURE 4.4 – Écran de statistiques et de victoire à la fin d'une partie

Conclusion

Après cette phase de modélisation, nous allons maintenant pouvoir passer à la réalisation concrète du projet. Certaines fonctionnalités et visuels pourront disparaître ou être modifiés lors de leur mise en oeuvre, mais le fonctionnement global de notre logiciel n'aura plus qu'à être implémenté tel quel.

Nous avons eu des difficultés sur les diagrammes présentés, car les contraintes de création de ceux-ci sont nombreuses. Cependant, à leur écriture nous nous sommes rendus compte que nous aurions oublié certaines fonctionnalités et que nous avons donc évité d'accumuler une dette technique trop importante.

Nous espérons avoir été assez exhaustifs dans le choix des diagrammes présentés dans ce rapport, nous les avons choisi car ils représentaient pour nous des étapes importantes ou qui pouvaient poser problème dans le développement du projet.