Projet Logiciel Transversal

Paul VIELLET – Lucas EHLINGER

Table des matières

Objectif **3**

Présentation générale 3

Règles du jeu 4

Conception Logiciel 10

Description et conception des états **11**

Description des états 11

Conception logiciel 11

Conception logiciel : extension pour le rendu 11

Conception logiciel : extension pour le moteur de jeu 11

Ressources 11

Rendu : Stratégie et Conception **13**

Stratégie de rendu d'un état 13

Conception logiciel 13

Conception logiciel : extension pour les animations 13

Ressources 13

Exemple de rendu 13

Règles de changement d'états et moteur de jeu **15**

Horloge globale 15

Changements extérieurs 15

Changements autonomes 15

Conception logiciel 15

Conception logiciel : extension pour l'IA 15

Conception logiciel : extension pour la parallélisation 15

Intelligence Artificielle **17**

Stratégies 17

Intelligence minimale 17

Intelligence basée sur des heuristiques 17
Intelligence basée sur les arbres de recherche 17
Conception logiciel 17
Conception logiciel : extension pour l'IA composée 17
Conception logiciel : extension pour IA avancée 17
Conception logiciel : extension pour la parallélisation 17
Modularisation **18**
Organisation des modules 18
Répartition sur différents threads 18
Répartition sur différentes machines 18
Conception logiciel 18
Conception logiciel : extension réseau 18
Conception logiciel : client Android 18

1 Objectif

1.1 Présentation générale

Figure 1 - Exemple d'un écran de jeu

Le but du projet est de réaliser un jeu basé sur Armello (figure 1), un jeu vidéo inspiré des jeux de plateau. Bien sûr il sera modifié (vue 2D sans déplacement de caméra, règles simplifiées).

Note : Toutes les règles et mécaniques qui seront énoncées dans la suite sont celles du jeu de base, elles sont donc susceptibles d'être changées ou abandonnées par soucis de simplification.

Légende :

1. Santé du personnage du joueur
2. Portrait du personnage en train de jouer
3. Nombre de points d'actions(1) restant au personnage jouant (joueur seulement)
4. Cartes en possession du joueur
5. Caractéristiques du joueur. De gauche à droite :
    - Combativité(2)
    - Vitalité(3)
    - Présence d'esprit(4)

    - Spiritualité[5]
6. Ressources du joueur. De gauche à droite :
    - Or[6]
    - Magie[7]
    - Prestige[8]
    - Déclin[9]

Glossaire

[1] Points d'actions : détermine le nombre de déplacement par tour.

[2] Combativité : détermine le nombre de dés disponible en combat.

[3] Vitalité : détermine la santé maximale du joueur.

[4] Présence d'esprit : détermine le maximum de cartes dans la main du joueur.

[5] Spiritualité : détermine le maximum de magie du joueur.

[6] Or : sert à jouer des cartes. Une colonie[10] sous le contrôle du joueur rapporte 1 or à l'aube.

[7] Magie : sert à jouer des cartes. La magie est restaurée au crépuscule.

[8] Prestige : à l'aube, le joueur avec le plus de prestige choisi une loi à appliquer. Un joueur perd 1 point de prestige à chaque mort.

[9] Déclin : maladie pouvant affecter les personnages. Le Roi commence à un Déclin de 1 et gagne 1 point de Déclin à chaque crépuscule. Un personnage touché par le Déclin perd 1 point de santé à chaque aube. Entre 1 et 4 points de Déclin, un joueur est considéré comme Infecté[11]. À partir de 5 points, et au-delà, un joueur est considéré comme Corrompu[12].

[10] Colonie : les cases spécifiques villages deviennent des colonies appartenant à un joueur quand ce joueur passe dessus.

[11] Infecté : un joueur infecté perd un 1 point de santé à chaque aube

[12] Corrompu : un joueur corrompu perd 1 point de santé à chaque aube. Une classe de case spécifique (cercle de pierres) tue les joueurs corrompus quand ils passent dessus. Effets supplémentaires en phase de combat.

1.2 Règles du jeu

Généralités

Dans Armello, les joueurs (au nombre de quatre -les places vacantes sont jouées par l'IA-) ont pour but de venir à bout d'un Roi touché par le Déclin (situé au centre du plateau). Le Déclin fait perdre 1 point de santé au Roi chaque jour (un jour est composé de deux tours de table, un diurne et un nocturne) ce qui limite donc le temps de jeu. En plus des joueurs, des Gardes Royaux contrôlés par l'IA sont en jeu, des Fléaux également contrôlés par l'IA peuvent apparaître, les spécificités de ces unités ne seront pas détaillées ici. Lors d'une interaction, un joueur peut perdre tous ses points de santé. Le cas échéant, il revient à sa case départ. Si un joueur perd toute sa santé durant son tour, celui-ci s'achève.

Plateau

Le jeu se déroule sur un plateau rectangulaire composé de cases hexagonales de taille 6x7. Les joueurs commencent aux coins du plateau.

Il existe 8 types de cases différentes :

- Plaine
- Forêt
- Montagne
- Village
- Donjon
- Marais
- Cercle de pierres
- Jardin du palais

Les cases ont des spécificités différentes. Il faut noter que les cases de départ des joueurs sont nécessairement et uniquement mitoyennes de 2 plaines. Il y a seulement 4 jardins du palais en jeu, positionné au centre du plateau. Pour accéder au Roi, un joueur doit se trouver sur une case jardin du palais.

Condition de victoire

Il existe quatre conditions de victoire :

- Tuer le Roi.
- Confronter le Roi en ayant un niveau de Déclin plus élevé que lui.
- Avoir le niveau de prestige le plus élevé quand le Roi meurt à cause du Déclin.
- Confronter le Roi après avoir réuni 4 pierres de d'esprit[1].

Cartes

Figure 1 - Exemple de cartes

Il existe 3 catégories de cartes dans Armello :

- Équipement
- Sort
- Ruse

Les cartes sont séparées en plusieurs types, le type est inscrit dans le coin supérieur droit de la carte, voir figure 2. Au début de chaque nouveau tour (aube ou crépuscule), le joueur pioche des cartes dans une des trois piles jusqu'à avoir rempli sa main.

Il existe 6 types de cartes :

- Bouclier
- Épées
- Lune
- Soleil
- Déclin
- Wyld[2]

Combat

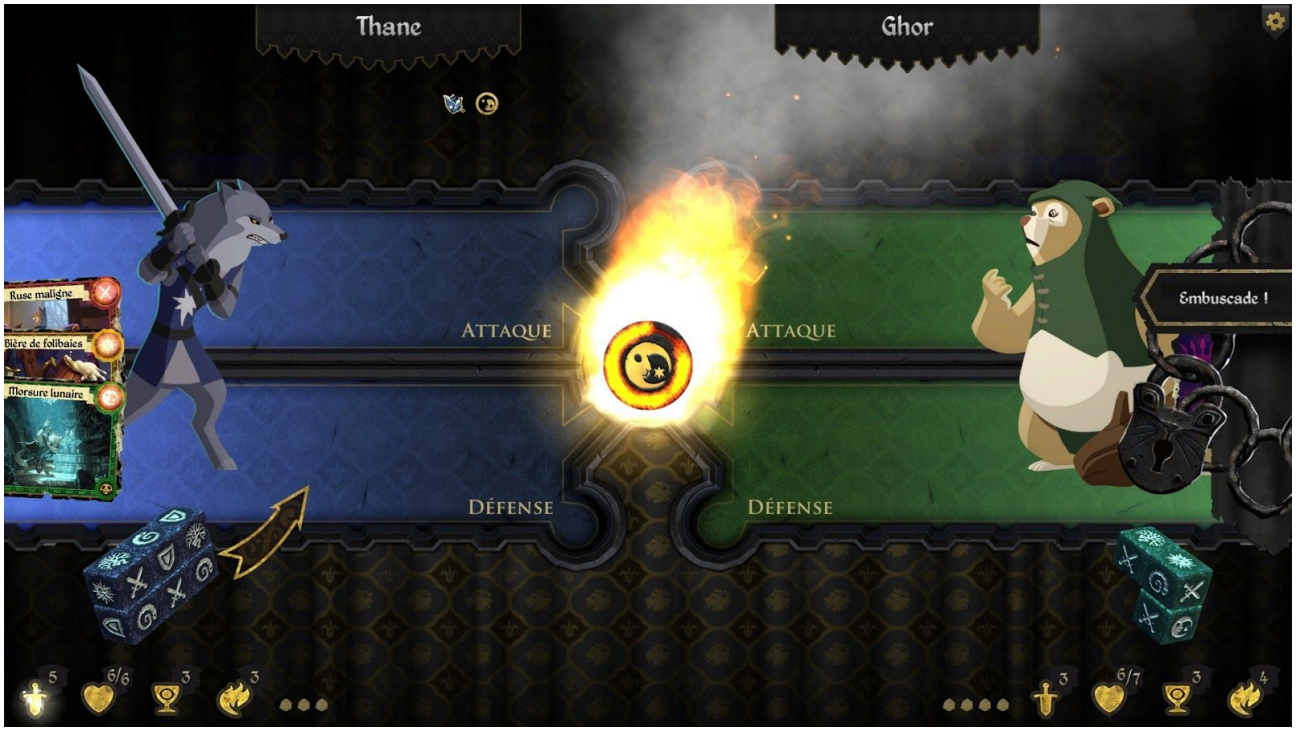


Figure 2 - Début d'un combat

Figure 3 - Résolution d'un combat

Lors du jeu, des combats peuvent se déclencher (joueur contre joueur, joueur contre IA, IA contre IA). Le déroulement est décidé aux dés, voir figure 2, le nombre de dés à lancer est déterminé par la caractéristique de combativité. Les participants peuvent brûler[3] leurs cartes au début du combat afin de fixer le résultat d'un nombre de dés équivalent, en fonction des cartes brûlées.

Dés

Figure 4 - Exemple de dé

Les dés possèdent 6 faces, une pour chaque type de carte, voir figure 4, le détail des effets en combat est précisé :

- Bouclier
  - Le participant gagne une défense[4]
- Épées
  - Le participant gagne une attaque[5]
- Lune
  - La nuit, le participant gagne une attaque
  - Le jour, le dé est défaussé
- Soleil
  - La nuit, le dé est défaussé
  - Le jour, le participant gagne une attaque
- Déclin
  - Participant Corrompu, le participant gagne une explosion[6]
  - Participant Non-Corrompu, le dé est défaussé
- Wyld
  - Participant Corrompu, le dé est défaussé
  - Participant Non-Corrompu, le participant gagne une explosion

Péril

Figure 5 - Exemple de péril

Un péril est une épreuve au cours de laquelle le joueur doit tirer des symboles spécifiques avec ses dés sous peine de subir une pénalité. Comme en combat, le joueur peut brûler ses cartes pour fixer le résultat d'un nombre de dés équivalent, voir figure 5.

Glossaire

[1] Pierre de d'esprit : apparaît aléatoirement sur des cercles de pierres, trouvable lors de quêtes ou lors de l'exploration de donjons.

[2] Wyld : élément d'histoire dans le jeu, inutil dans le projet, c'est l'arbre sur les dés et cartes.

[3] Brûler une carte : défausser une carte au début d'un péril ou combat afin de fixer le résultat d'un dé.

[4] Défense : empêche la perte de 1 point de santé lors d'un combat.

[5] Attaque : retire 1 point de santé à l'adversaire lors d'un combat.

[6] Explosion : le dé confère une attaque et un nouveau dé est lancé, disponible seulement en combat.

1.3 Conception Logiciel

Figure 1 - Cases utilisées

Figure 2 - Exemple de sprite de personnage

2 Description et conception des états

2.1 Description des états

Le jeu, même simplifié, possède différents éléments de jeu (des pions, des cases, des ressources,...) ces éléments possèdent des propriétés et des actions. Ici en programmation orientée objets nous définissons un élément par un objet, les propriétés des éléments sont des attributs et les actions des méthodes.

Le jeu étant un jeu de plateau ce dernier est l'élément central du logiciel, ce plateau est de forme hexagonale et est constitué de cases de différents types. Sur ce plateau se situe aussi des pions, les pions sont de différents types et tous les types n'ont pas les même possibilités d'actions.

2.1.1 Les cases

Le plateau est composé de 127 cases, aucune n'est une simple case, elles ont toutes des particularités. Cependant certaines particularités sont communes.

- Départ : la case départ définit un type de case sur lequel un joueur débute, une fois la partie lancée le joueur n'y retourne plus, sauf en cas de mort.

- Champ : la case champ est la case la plus banale, elle n'applique aucun effet au pion se déplaçant dessus, mais nécessite l'utilisation par le pion d'un point de déplacement pour s'y rendre.

- Ruine : la case ruine comme le champ nécessite un point de déplacement, mais permet au joueur d'effectuer une téléportation sur une autre case du même type.

- Cercle de pierre : comme la case ruine et champ, la case cercle de pierre nécessite un point de déplacement, cette case permet au joueur l'occupant la récupération d'un point de vie, dans la limite de son maximum de points de vie.

- Marais : comme la case cercle de pierre, ruine et champ, la case marais nécessite un point de déplacement, mais elle inflige au joueur s'y rendant la perte d'un point de vie.

- Montage : à l'inverse de la plupart des cases précédente, cette case nécessite non pas un mais deux points de déplacement pour y accéder. Cette case permet aussi au pion qui s'y trouve et qui s'y fait attaquer de disposer d'un dispositif de défense (bouclier) contre les attaques, il n'est pas à l'abris mais est renforcé.

- Colonie : la colonie nécessite un point de déplacement et devient la propriété du pion qui la traverse (cette propriété aura une utilité plus tard dans le jeu, mais il ne s'agit pas d'un état ici).

- Forêt : la forêt nécessite un point de déplacement et permet aux joueurs, de devenir invisible la nuit tant qu'ils restent sur une case du même type.

- Roi : le roi est en fait un personnage, mais il n'existe que sur la case centrale du tableau, il est donc plus assimilable à une case particulière qu'à un pion. Le roi pourrait être aussi assimilé comme un pion et une case. Il s'agit du boss du jeu qui peut être mis en défaut de différentes manières.

- Château : Le château est la demeure du roi, il s'étend tout autours de la case Roi. sa propriété principale est de faire passer un tour aux joueur atteignant ce type de case.

2.1.2 Les pions

Il existe plusieurs types de pions avec des actions différentes.

- Le joueur, il est réel ou piloté par une IA. C'est un personnage qui est présent à plusieurs reprises sur la carte et qui démarre la partie sur une case départ.

- Les gardes sont ceux du château, ils se baladent sur le plateau et peuvent dans certains cas attaquer les joueurs.

- Fléau, présent sur certaines cases les fléaux peuvent attaquer les joueurs et les gardes.

Tous les pions disposent de certaines ressources et statistiques.

2.2 Conception logiciel

Les éléments du jeu sont définis comme des objets, nous utilisons la hiérarchie de classe pour diminuer le nombre de méthodes via l'utilisation du polymorphisme.

Le tableau (Board) est la classe de base du jeu, c'est sur cette classe que les propriétés de la partie sont définie.

- Liste des joueurs, un vecteur regroupant les joueurs de la partie. Ce vecteur pourra être modifié en longueur au cours de la partie.

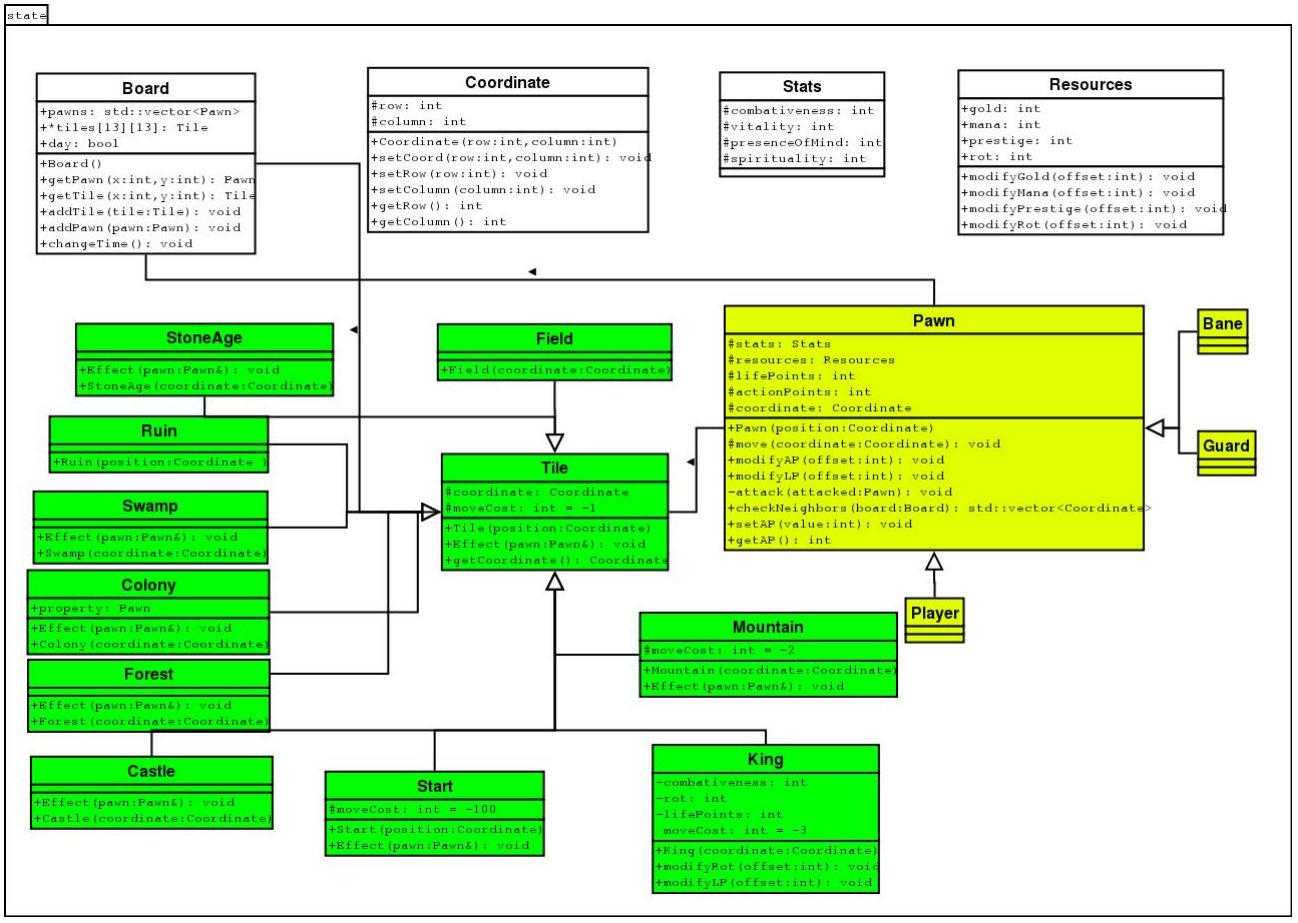
- Tableau des cases, les cases sont générées au début de la partie, c'est à dire dans le constructeur de la classe Board. Ce tableau ne doit pas changer en nombre d'éléments.

- Jour, une simple valeur booléenne définissant si les actions se déroulent de nuit ou de jour.

Toutes les cases évoquées précédemments sont définies dans une classe case (Tile). Les particularités des différentes cases sont définies dans des classes enfant. En effet, une montagne, un cercle de pierre, une forêt,... sont des types de cases cela a donc du sens de les définir comme enfants de cases et de développer les éléments communs aux différentes cases dans la classe mère (Tile).

Les différents types de joueurs suivent le même principe.

On ajoute des classes coordonnées, statistiques et ressources qui pourraient être définies comme des structures mais dans notre contexte l'utilisation de classe est plus adéquate.



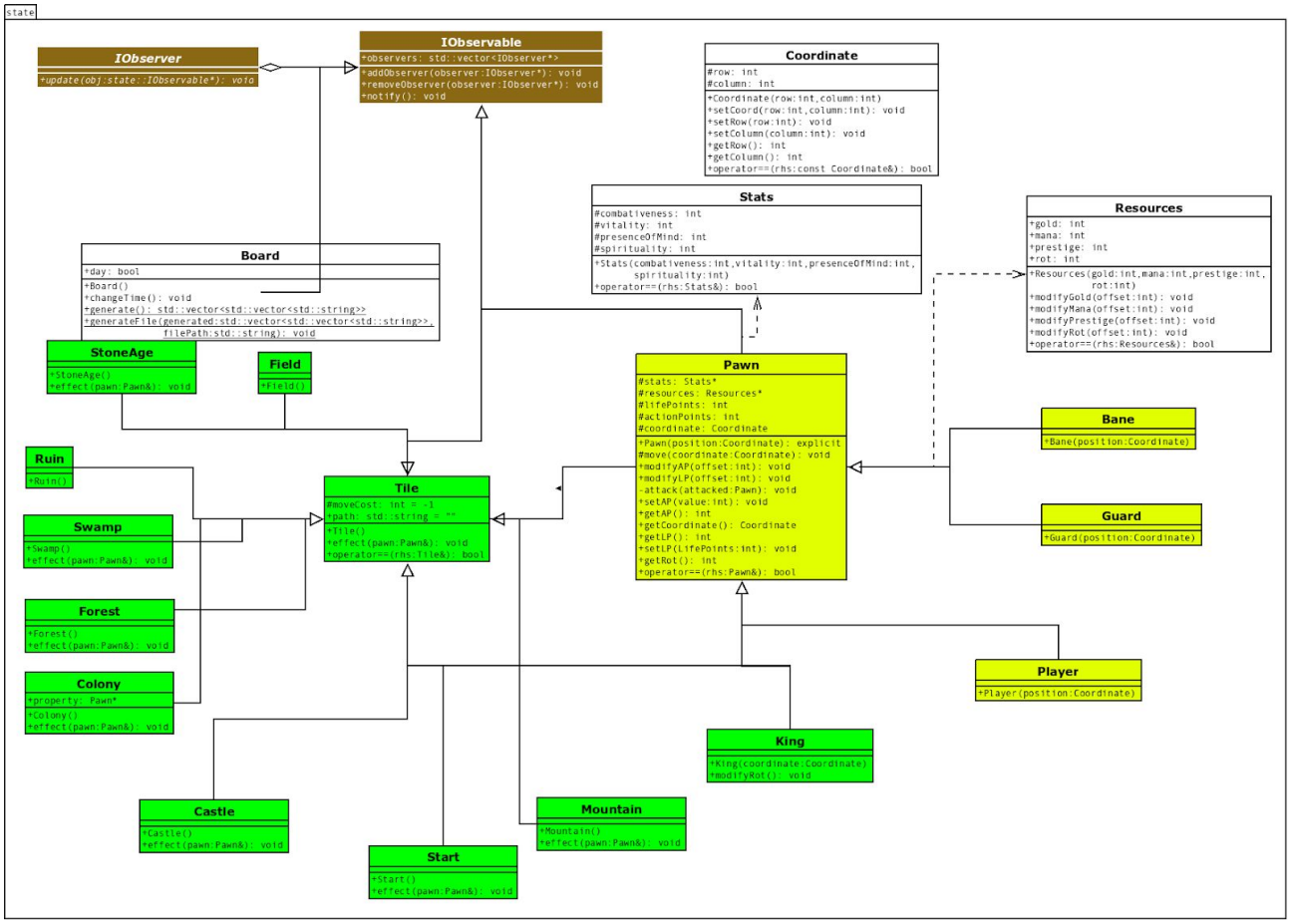
2.3 Conception logiciel : extension pour le rendu

En prenant un peu de recul sur le projet on observe une erreur de conception. En effet nous somme dans le cas d'un schéma classique : Model Vue Controller (MVC) or nous mélangeons un peu les choses dans notre schéma précédent.

Le namespace state représente le Modèle du schéma MVC et est stocké sur un serveur de jeux. Le Modèle doit être singleton sur une partie et tous les clients doivent s'y référer pour l'affichage de données. Il n'est aussi pas du ressort du modèle d'implémenter des éléments de la logique de jeux (voir Controller plutôt), nous avons donc revu la conception de cette partie pour que les éléments communiquent mieux entre eux et pour préserver une logique simple de fonctionnement.

2.4 Conception logiciel : extension pour le moteur de jeu

2.5 Ressources


state
IObserver
+update(obj:state::IObservable*): void
IObservable
+observers: std::vector<IObserver*>
+addObserver(observer:IObserver*): void
+removeObserver(observer:IObserver*): void
+notify(): void
Coordinate
#row: int
#column: int
+Coordinate(row:int,column:int)
+setCoord(row:int,column:int): void
+setRow(row:int): void
+setColumn(column:int): void
+getRow(): int
+getColumn(): int
+operator==(rhs:const Coordinate&): bool
Stats
#combativeness: int
#vitality: int
#presenceOfMind: int
#spirituality: int
+Stats(combativeness:int,vitality:int,presenceOfMind:int,
spirituality:int)
+operator==(rhs:Stats&): bool
Resources
+gold: int
+mana: int
+prestige: int
+rot: int
+Resources(gold:int,mana:int,prestige:int,
rot:int)
+modifyGold(offset:int): void
+modifyMana(offset:int): void
+modifyPrestige(offset:int): void
+modifyRot(offset:int): void
+operator==(rhs:Resources&): bool
Board
+day: bool
+Board()
+changeTime(): void
+generate(): std::vector<std::vector<std::string>>
+generateFile(generated:std::vector<std::vector<std::string>>,
filePath:std::string): void
StoneAge
+StoneAge()
+effect(pawn:Pawn&): void
Field
+Field()
Pawn
#stats: Stats*
#resources: Resources*
#lifePoints: int
#actionPoints: int
#coordinate: Coordinate
+Pawn(position:Coordinate): explicit
#move(coordinate:Coordinate): void
+modifyAP(offset:int): void
+modifyLP(offset:int): void
-attack(attacked:Pawn): void
+setAP(value:int): void
+getAP(): int
+getCoordinate(): Coordinate
+getLP(): int
+setLP(lifePoints:int): void
+getRot(): int
+operator==(rhs:Pawn&): bool
Bane
+Bane(position:Coordinate)
Guard
+Guard(position:Coordinate)
Ruin
+Ruin()
Tile
#moveCost: int = -1
+path: std::string = ""
+Tile()
+effect(pawn:Pawn&): void
+operator==(rhs:Tile&): bool
Swamp
+Swamp()
+effect(pawn:Pawn&): void
Forest
+Forest()
+effect(pawn:Pawn&): void
Colony
+property: Pawn*
+Colony()
+effect(pawn:Pawn&): void
Player
+Player(position:Coordinate)
King
+King(coordinate:Coordinate)
+modifyRot(): void
Castle
+Castle()
+effect(pawn:Pawn&): void
Mountain
+Mountain()
+effect(pawn:Pawn&): void
Start
+Start()
+effect(pawn:Pawn&): void

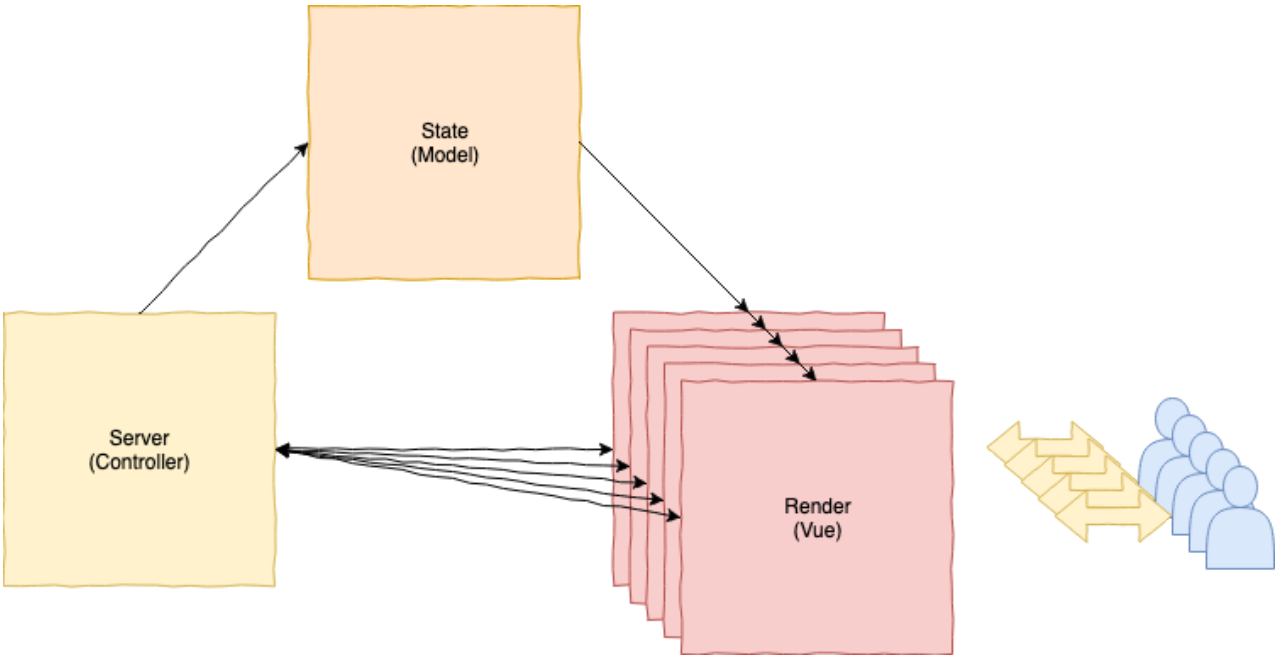
3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Un humain jouant doit comprendre très rapidement, en observant sa fenêtre de jeu, le plateau, sa position et la position des ennemis. Nous avons donc choisi de nous orienter vers un modèle de rendu d'état qui puisse être scalable à une infrastructure client/serveur avec un nombre de client inférieur ou égal à 6.

Le rendu est donc conçu pour nécessiter le moins de requête réseau à effectuer. Ainsi au lieu de surveiller l'état du plateau en entier et de le recharger à chaque modification, nous considérons chaque case de jeu comme un élément de la vue, s'initialisant et suivant les modifications effectuées sur l'état. On peut illustrer par un objet tuile dans le modèle mettant à jour un objet case dans le rendu des différents clients. Le plateau sera généré par le serveur (controller) au début de chaque partie.

Ce que nous appliquons pour les cases suit la même logique pour les pions et pour l'état général du plateau.



Pour le moment le seul problème non résolu est le suivant : comment passer aux éléments du Render la nouvelle version de l'objet modifié dans le State ? En d'autres mots : que devons nous passer en argument de la fonction *"notif"* implémentée dans les objets *observer* ? Plusieurs solutions (peut-être) possibles :

- Envoie en argument d'un objet de type "Observable" la fonction serait donc : *update(this)*
- ~~Envoie en argument la totalité des attributs des objets du même type. La fonction s'appellerait donc de cette manière : update(arg1, arg2, arg3, ….) mais cela risque de ne pas être possible sans beaucoup d'aménagement et une implémentation peu propre.~~
- Envoie en argument d'une sérialisation de l'objet. c'est à dire à chaque modification, sérialiser l'objet qui a été modifié (en json ou xml) puis envoyer cette sérialisation (sous forme de chaîne de caractère) en argument. La fonction se définirait comme telle : update(std::string serialized). Nous pensons que c'est ici la façon la plus propre de faire les choses mais aussi la plus difficile à implémenter.

Pour le moment nous décidons de retenir la solution d'un passage d'argument d'un objet IObservable, mais cette solution sera portable sans trop de difficultés à un système de sérialisation et de trames IP, indispensables pour une utilisation en réseau.

3.2 Conception logiciel

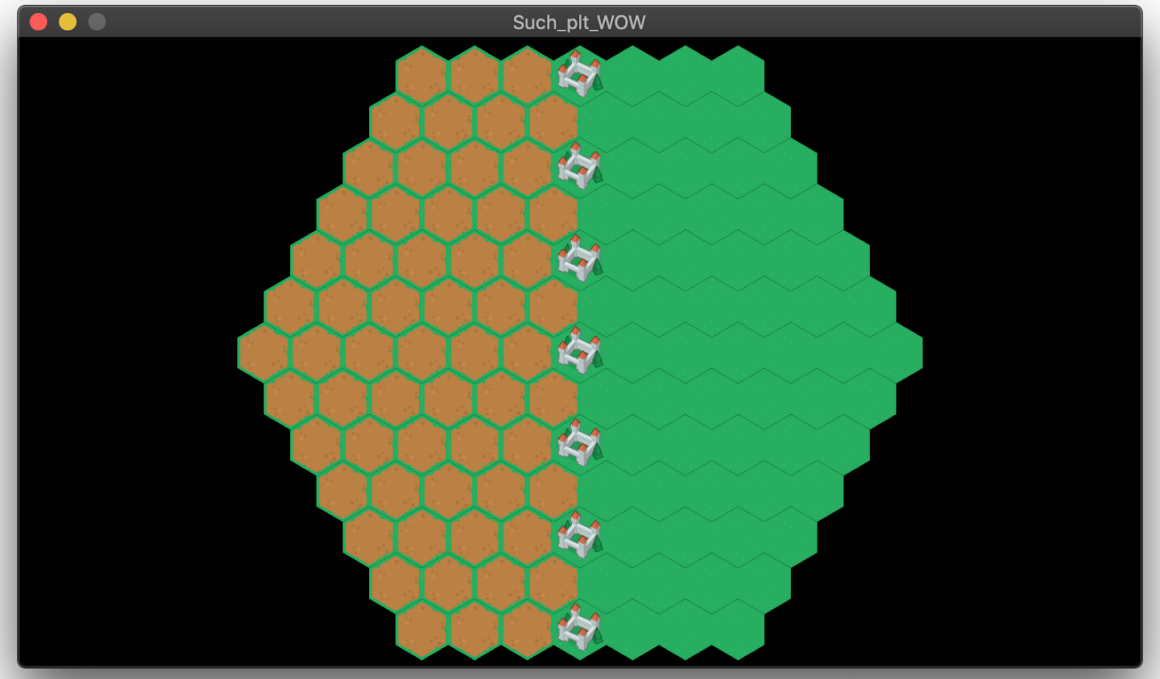
Nous aurions pu aussi voir le plateau comme le seul objet à observer et à mettre à jour, cependant ceci aurait impliqué de grosses requêtes ethernet pour la mise à jour des différents clients. Ici les requêtes sont plus nombreuses mais nettement plus légères.

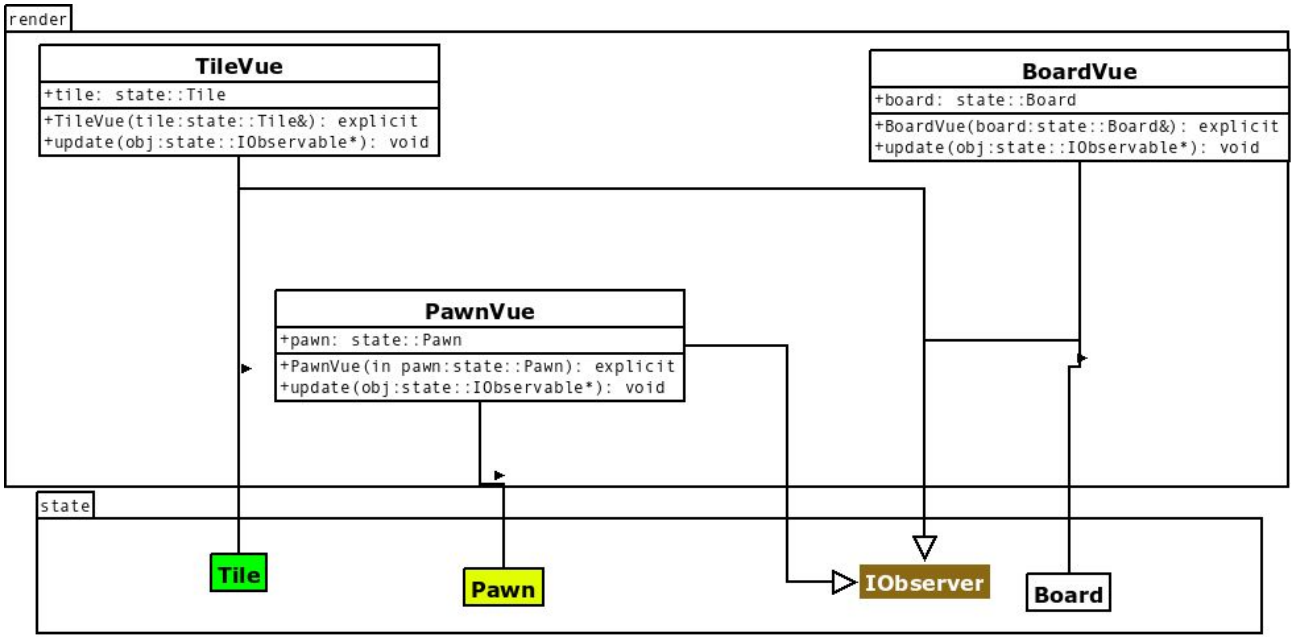
3.3 Conception logiciel : extension pour les animations

3.4 Ressources

3.5 Exemple de rendu

Pour le moment nous générons par plusieurs boucles un ensemble de tuiles qui sera une idée de la base de notre programme. Pour générer un plateau nous allons implémenter dans la classe state::Board une méthode statique qui lorsqu'elle sera appelée générera une carte de jeu aléatoire mais en respectant les contraintes inhérentes au jeu (case départ, case arrivée). Cette méthode statique rendra un fichier json qu'il faudra dé-sérialiser dans le moteur de rendu pour l'afficher.




render
TileVue
+tile: state::Tile
+TileVue(tile:state::Tile&): explicit
+update(obj:state::IObservable*): void
BoardVue
+board: state::Board
+BoardVue(board:state::Board&): explicit
+update(obj:state::IObservable*): void
PawnVue
+pawn: state::Pawn
+PawnVue(in pawn:state::Pawn): explicit
+update(obj:state::IObservable*): void
state
Tile
Pawn
IObserver
Board

4 Règles de changement d'états et moteur de jeu

4.1 Horloge globale

Un changement d'état est provoqué par le changement et la notification d'un elements de la classe état. Mais ce changement ne peut se déclencher que par l'action d'un joueur. C'est un jeu tour par tour, chaque joueur pourra donc jouer à son tour suivant un ordre précis et défini au début de la partie. Seul l'action d'un joueur (ou le fait de passer un tour) déclenche un changement d'état. Le changement d'état est un simple déplacement d'un joueur (pouvant en résulter une attaque).

4.2 Changements extérieurs

Les changement

4.3 Changements autonomes

4.4 Conception logiciel

4.5 Conception logiciel : extension pour l'IA

4.6 Conception logiciel : extension pour la parallélisation

*Illustration SEQ Illustration * ARABIC 3: Diagrammes des classes pour le moteur de jeu*

5 Intelligence Artificielle

Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.

5.1 Stratégies

5.1.1 Intelligence minimale

5.1.2 Intelligence basée sur des heuristiques

5.1.3 Intelligence basée sur les arbres de recherche

5.2 Conception logiciel

5.3 Conception logiciel : extension pour l'IA composée

5.4 Conception logiciel : extension pour IA avancée

5.5 Conception logiciel : extension pour la parallélisation

6 Modularisation

Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est un parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau

6.4 Conception logiciel : client Android

*Illustration SEQ Illustration * ARABIC 4: Diagramme de classes pour la modularisation*