

Projet Logiciel Transversal

JAMET Romain DORRA Benjamin

Table des matières

1	Présentation Générale	1
1.1	Archétype	1
1.2	Règles du jeu	1
1.3	Ressources	2
2	Description et conception des états	3
3	Rendu : Stratégie et Conception	7
3.1	Stratégie de rendu d'un état	7
3.2	Conception logiciel	7
4	Règles de changement d'états et moteur de jeu	8
4.1	Règles	8
4.2	Conception logiciel	9
5	Intelligence Artificielle	9
5.1	Stratégies	9
5.2	Conception logiciel	10
6	Modularisation	10
6.1	Organisation des modules	10
6.2	Conception logiciel	11

1 Présentation Générale

1.1 Archétype

Le jeu d'origine est le jeu de stratégie/gestion Crusader Kings II, développé et édité par le studio Paradox Interactive.

Il s'agit d'un jeu se déroulant dans l'Europe et le Moyen-Orient entre le VIII^e et le XV^e siècle, où les joueurs jouent le rôle d'une dynastie de nobles régnant sur un domaine, qu'ils peuvent faire croître par une mécanique de titres de noblesse, de guerre et de diplomatie inspirés du Moyen-Âge. Les règles en seront bien évidemment simplifiées dans le cadre du projet.

Le descriptif des règles ci-dessous présente le projet tel qu'il devrait être au final.

1.2 Règles du jeu

Les joueurs sont des nobles qui règnent sur ces provinces, en possédant un ou plusieurs titres de noblesse. Les trois titres sont, dans l'ordre du plus puissant au moins puissant : les Rois, les Ducs, et les Comtes.

À chaque province est lié un titre de Comte. Posséder un titre de Comte, c'est posséder la province associée.

Une province possède plusieurs caractéristiques décrivant son niveau de développement, son état de dévastation ou de prospérité, qui influent sur les impôts qu'elle génère, et les levées militaires qu'elle peut supporter. Un Comte peut posséder plusieurs Comtés, cependant, le nombre en est limité. Si cette limite est dépassée, un malus d'impôts et de levées est appliqué.

Un Duc est plus puissant qu'un Comte. Pour pouvoir contrôler plus de territoire malgré la limite mentionnée précédemment, il doit confier la gestion de certaines provinces à des nobles qui seront ses vassaux.

Un Duc ne peut avoir que des Comtes pour vassaux.

Un Roi est plus puissant qu'un Duc. Il peut avoir comme vassaux des Ducs et des Comtes. Lorsqu'un seigneur est vassal, il paie un impôt en or et en hommes à son suzerain.

Pour faire la guerre à son voisin, il faut un Casus Belli. Il y aura la possibilité de dépenser de l'or pour fabriquer une revendication à un titre de Comte, pour essayer de s'en emparer à la guerre. Une fois la guerre déclarée, on peut mobiliser les troupes dans les provinces que l'on possède, ainsi que les troupes de ses vassaux si on en a, et leur donner des ordres de déplacement sur la carte.

Lorsque deux armées hostiles se rencontrent, un combat a lieu. L'armée défaite est mise en déroute et cherchera automatiquement à revenir à sa province d'origine.

Chaque Comté fait partie d'un Duché «de Jure», c'est-à-dire un Duché dont le titre n'existe pas, mais qui peut être créé si quelqu'un en contrôle plus de la moitié et dépense une certaine somme. Le fait de posséder un titre de Duc offre un Casus Belli permettant de vassaliser les Comtes dont les terres font partie «de Jure» de votre Duché.

De même, chaque Duché fait partie d'un Royaume de Jure. Posséder ou contrôler par le biais de vassaux une certaine proportion d'un Royaume de Jure vous permet de créer ce titre en échange d'argent. Posséder un tel titre offre un Casus Belli contre tous les Ducs et Comtes faisant partie de Jure de votre Royaume. On peut bien évidemment posséder ou contrôler par vassaux des provinces n'appartenant pas de Jure à votre domaine.

Il est possible d'interagir avec d'autres personnages, forger des alliances, des pactes de non-agression, proposer à un seigneur plus faible de devenir votre vassal.

Chaque personnage possède un score décrivant son opinion envers les autres, influencé par l'historique de leurs actions. Ce score permettra de définir le comportement de l'IA. Il sera également influencé par des traits de personnalité des personnages. L'IA prendra ses décisions en utilisant une mécanique de MMTH (mean time to happen, ou temps moyen d'attente). Pour chaque action, il y a un MTTH en nombre de tours, avec des modificateurs liés aux traits de personnalité aléatoires des personnages.

Au cours de son règne, un personnage accumule des points de prestige selon la quantité de titres qu'il possède, ainsi que les titres de ses vassaux. Lorsqu'il meurt, un héritier apparaît, et le score de prestige s'ajoute au score du joueur. Le score final d'un joueur correspond donc à la somme de points de prestige accumulés par tous les seigneurs successifs qu'il a joué.

1.3 Ressources

Pour réaliser ce projet, nous aurons besoin en termes de ressources d'une carte du monde du jeu, avec les différentes provinces découpées.

Nous aurons besoin d'une version pour l'affichage, et d'une autre qui ne sera pas affichée pour identifier la province sur laquelle le joueur clique, par un code couleur unique par province.

Nous aurons besoin de sprites d'armée immobile, en mouvement, et à la bataille.

Nous devons pouvoir coloriser facilement les bannières pour identifier immédiatement le contrôleur de l'armée.

Nous pouvons également avoir besoin d'images pour les personnages, même si ce n'est pas obligatoire il s'agirait d'un plus. Dans le même ordre d'idées, nous pourrions employer diverses images sur le thème médiéval pour illustrer les différents menus du jeu. Nous pouvons aussi utiliser différentes musiques d'ambiance, des bruitages de bataille, et quelques sons génériques pour l'appui sur les boutons de l'interface.

2 Description et conception des états

L'état de jeu est contenu dans une classe principale : GameState, qui contient tous les éléments du jeu propres à l'état d'une partie de jeu. Certains éléments de l'état de jeu ont besoin de références à d'autres éléments. Afin d'être compatible avec une exécution rapide en jeu et un système de sauvegarde sûr, les éléments auxquels il peut être fait référence sont dotés d'un champ id unique (par classe d'éléments), qui servent à les identifier. Ainsi, pour référencer le personnage propriétaire d'un titre dans la classe Title on doit mettre l'adresse de l'objet personnage correspondant ET l'id unique correspondant à ce personnage. Pour ce faire, on emploie des std::pair dont le premier élément est le std::string de l'id, et le second élément un pointeur vers l'objet. Pour les listes d'éléments, on utilisera un std::map selon le même principe.

- GameState

Elle contient :

- un objet ressources de la classe Ressources, chargé de contenir toutes les ressources statiques servant au fonctionnement du jeu.
- un objet gameMap de la classe GameMap, décrivant l'état des éléments dynamiques de la carte,
- un objet politics de la classe Politics, décrivant l'organisation géopolitique de la partie.
- une liste de joueurs (players) de la classe std::vector<Player> pour suivre l'évolution des différents joueurs, humains ou IA.

- Politics

La classe Politics décrit l'ensemble des entités politiques existantes.

- On a d'abord la liste des titres (titles), de la classe std::map<std::string, Title>. Il s'agit de tous les titres qui existent ou peuvent être créés.
- On a ensuite la liste de tous les personnages (characters), de la classe std::map<std::string, Character>, qui contient tous les personnages en vie ou morts.
- Enfin, on a la liste de toutes les relations (relations) qui peuvent exister entre deux personnages (alliances, etc.), de la classe std::vector<Relation>.

- Title

La classe Title décrit un titre de noblesse, que celui-ci soit possédé par quelqu'un ou non. Il contient

- un champ id de type std::string,
- un champ nom(name) de type std::string,
- un champ propriétaire (holder) de type std::string

De la classe Title, héritent les classes County, Duchy, Kingdom (Comté, Duché, Royaume).

- County

- un champ province de type std::string qui désigne la province associée au titre
- un champ liege de type std::string qui désigne le suzerain
- un champ deJureLiege de type std::string qui désigne le suzerain de jure

- County

- un champ province de type std::string qui désigne la province associée au titre
- un champ liege de type std::string qui désigne le suzerain
- un champ deJureLiege de type std::string qui désigne le suzerain de jure

- Duchy

- un champ liege de type std::string qui désigne le suzerain
- un champ deJureLiege de type std::string qui désigne le suzerain de jure

- Kingdom

- Character

La classe Character décrit les personnages du jeu. Les personnages peuvent posséder des titres, ont des statistiques, une opinion, des traits de caractère et peuvent mener des guerres ou comploter. Elle possède de nombreux attributs :

- un champ nom de type std::string
- un champ dynastyName de type std::string qui décrit la dynastie
- un champ age de type int
- un champ opinions de type std::vector<Opinion> qui contient les opinions d'un personnage sur les autres
- un champ diplomacy de type int qui décrit son talent diplomatique
- un champ stewardship de type int qui décrit sa capacité à gérer les fiscalités de provinces
- un champ martial de type int qui décrit son talent martial
-
- un champ intrigue de type int qui décrit son talent pour les complots
- un champ claims de type std::vector<std::string>
- un champ alive de type bool
- un champ prestige de type int qui décrit le score du personnage
- un champ gold de type int
- un champ currentPlot de type Plot qui décrit le complot (au plus un) mené actuellement par le personnage.

— Opinion

Cette classe montre l'opinion d'un personnage sur un autre

- un champ target de type std::string
- un champ score de type std::int

— Plot

Un complot mené par un personnage. Il présente un champ endTurn de type int qui détermine le tour où le but du complot est atteint. Plot a deux sous-classes : ClaimPlot et MurderPlot.

— MurderPlot

Un complot visant à tuer un personnage. Présente le champ targetCharacter de type std::string.

— ClaimPlot

Un complot visant à s'emparer d'un titre. Présente le champ targetTitle de type std::string.

— Relation

La classe Relation décrit un lien particulier entre deux personnages actifs (vivants). Elle contient

- un champ characterA et
- un champ characterB, tous deux de type std::string,
- une énumération : enum type non_aggression, alliance, friendship, rivalry, war,
- une date (numéro de tour) de fin (endTurn) qui vaut 0 dans le cas d'une durée indéterminée.

— Player

La classe Player décrit un joueur (humain ou IA). Les champs sont :

- id de type std::string
- currentCharacter de type std::string qui décrit le personnage actuellement contrôlé
- score de type int qui décrit le score total du joueur

— AIPlayer

Une sous-classe de Player.

— HumanPlayer

Une sous-classe de Player.

— Ressources

La classe Ressources décrit les ressources statiques utilisées par le jeu. Elle contient les champs suivants :

- un champ traits de type Traits qui contient tous les traits de caractère que peuvent avoir les

- personnages
 - un champ adjacencies de type Adjacencies qui contient le nombre de tours nécessaires à une armée pour passer d'une province à une autre.
- Adjacencies

La classe Adjacencies regroupe le nombre de tours nécessaires pour passer d'une province à une autre dans le champ val de type `std::map<std::string, std::map<std::string, int>`. Si les provinces ne sont pas adjacentes, la distance est 0. Entre une province et elle-même, la distance est -1.
- Traits

Cette classe regroupe tous les traits de caractère que peut avoir un personnage dans le champ traits de type `std::vector<Trait>`.
- Trait

Cette classe décrit un trait de caractère. Elle comprend :

 - un champ id de type `std::string`
 - un champ opposedId de type `std::string` qui donne l'id du caractère opposé
 - diplomacyModifier de type `int` qui décrit l'effet qu'a le trait sur la statistique diplomacy
 - stewardshipModifier de type `int` qui décrit l'effet qu'a le trait sur la statistique stewardship
 - martialModifier de type `int` qui décrit l'effet qu'a le trait sur la statistique martial
 - intrigueModifier de type `int` qui décrit l'effet qu'a le trait sur la statistique intrigue
- GameMap

La classe GameMap décrit la situation des différentes zones, les batailles et les armées. Elle présente les champs suivants :

 - un champ provinces de type `std::map<std::string, Province>` qui regroupe les différentes provinces.
 - un champ armies de type `std::map<std::string, Army>` qui regroupe les différentes armées.
 - un champ battles de type `std::map<std::string, Battle>` qui regroupe les différentes batailles.
- Battle

Les champs sont :

 - id de type `std::string`
 - province de type `std::string` la province où a lieu la bataille
 - whiteArmies de type `std::vector<std::string>`
 - blackArmies de type `std::vector<std::string>`
 - startTurn de type `int` le tour de début de la bataille
 - endTurn de type `int` le tour de fin de la bataille
- Army

Décrit une armée. Contient les champs :

 - id de type `std::string`
 - levies de type `std::vector<std::string>` les id des levées qui composent l'armée
 - currentProvince de type `std::string` la position actuelle de l'armée
 - currentBattle de type `std::string` la bataille actuellement menée par l'armée
 - orders de type `std::queue<TravelOrder>` les ordres que suit l'armée.
- Order

Ordre donné à une armée. contient les champs :

 - originProvinceId de type `std::string` id de la province de départ
 - destinationProvinceId de type `std::string` idée de la province d'arrivée
 - duration de type `int` qui contient le temps total (en nombre de tours) nécessaire pour exécuter l'ordre
 - elapsed de type `int`, le nombre de tours écoulés depuis que l'ordre a été donné.
- TravelOrder

Ordre de déplacement d'une armée. Hérite de Order. Contient un champ de type `std::vector<One-`

- ProvinceOrder>
- OneProvinceOrder
 - Hérite de Order.
- Province
 - Cette classe décrit une province. Elle comprend les champs suivants :
 - id de type std : :string
 - name de type std : :string
 - colorCode de type char[3], identifiant de couleur unique qui permet d'identifier la province à sa position sur la carte.
 - development de type unsigned char
 - prosperity de type unsigned char
 - baseLevy de type Levy
 - levy de type Levy
 - baseTaxIncome de type unsigned int
 - taxIncome de type unsignedInt
- Levy
 - Décrit une levée.
 - men de type int le nombre d'hommes dans la levée
 - isRaised de type bool qui décrit si la levée a été mobilisée
 - reinforcementRate de type float la vitesse de renouvellement de la levée

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

3.2 Conception logiciel

4 Règles de changement d'états et moteur de jeu

4.1 Règles

4.2 Conception logiciel

5 Intelligence Artificielle

5.1 Stratégies

5.2 Conception logiciel

6 Modularisation

6.1 Organisation des modules

6.2 Conception logiciel