Projet d'Objet et Développement d'Application :
Untitled_Project : the Movie, the Game, Super Mega Neo Climax Ultimate Gold Turbo Anniversary X Remastered Edition[42] (ou UPMGSMNCUGTAXRE[42] pour faire plus court)

Loic BOUTIN Gr. 501B
Romain BERNARD Gr. 502C

Introduction :

Le choix du thème du projet fût très vite établi. Nous avions déjà prévu de coder un jeu ensemble pendant cette année scolaire et pouvoir le faire dans le cadre du travail

demandé par l'établissement nous a permis de nous motiver. Ce que nous voulions initialement était un jeu de plateau de stratégie à l'image d'un Fire Emblem, néanmoins nous étions motivé par le fait d'en faire un jeu plus dynamique faisant la part belle à la préparation hors combat et la rapidité de réaction pendant ces combats, à l'instar d'un Crypt of the Necrodancer entre autres.

Cette idée de "rythme" est restée et nous étions partis sur un mode de combat où les effets de nos actions changeraient en fonction du moment où l'action serait effectuée au cours d'un tour à durée limitée. L'idée était de découper chaque tour en plusieurs phases : le rush, le neutre et la préparation. Par exemple, une attaque de base lancée en rush aurait été plus puissante mais moins précise à l'inverse d'une attaque lancée en préparation. Chaque action aurait ainsi trois "formes" uniques en fonction du timing. (Spoiler, on a été naïfs)

1- Système de jeu :

Vous commencez sur un simple écran press start où vous pourrez, selon qu'on ait réussi à l'implémenter à temps potentiellement choisir une classe et un nom pour votre personnage. Après, vous voyez le plateau de jeu avec les ennemis et devez choisir à quelle position sera initialisé votre personnage.

La partie commence alors et se déroule sur un plateau de jeu vu de dessus, ce plateau est constitué de cases, pouvant avoir des propriétés comme leur type qui affecte l'entité la traversant de différentes façons comme en lui appliquant un buff ou un debuff. Pour l'instant, la condition de fin de partie est simplement la mort du joueur ou l'annihilation des ennemis (qui n'avaient rien demandés, espèce de monstre), à long terme, la récupération d'un objet/d'une unité pourrait affecter la victoire ou la défaite, mais ce n'est pas encore d'actualité.

En ce qui concerne les entités, chacune est composée de Caractéristiques et de Types qui définissent ses attributs et régissent ce qu'elles sont capables d'effectuer ou comment elles réagissent à certaines actions (un Zombie se faisant brûler par la lumière révélatrice d'un joyeux prêtre alors que cette même lumière soigne gaiement le reste du monde)

Ces mêmes entités possèdent une liste de Capacités qui représente le panel d'actions étant à leur disposition pour le combat, chaque liste étant dépendante de l'entité, une partie lancée avec J-X le broussailleux paladin ne ressemblera jamais à une lancée avec Bebito le moine boiteux. A long terme cette liste serait directement extensible par le joueur pour le laisser librement choisir s'il préfère vaincre ses ennemis à l'aide d'une tempête de lames ou s'il est un fervent utilisateur des arts de la pyromancie.

Et en parlant de pyromancie, le système de Decorator va nous permettre d'appliquer à toutes ces entités des buffs et des debuffs directement issus de ces Capacités, ainsi, brûlures, empoisonnements et galvanisations vont venir dynamiquement modifier certaines actions ou certaines caractéristiques de ces entités. Ces changements peuvent directement venir du terrain ou être issus des mêmes capacités dont on parlait un peu plus tôt.

Ces diverses actions sont gérées par le combo clavier souris, par le biais du clavier, le joueur sélectionnera la capacité (par défaut, c'est le mode de déplacement qui est sélectionné) à effectuer, on indiquera alors au programme d'un clique la case ciblée (que ça

soit un déplacement ou une capacité), et si la portée correspond à celle de la capacité/le nombre de Points de Mouvement du joueur et que le joueur a suffisamment de Points d'Action et/ou de Points de Corruption, (basiquement le mana) alors l'action sera effectuée.

2- Présentation des Classes :

Notre projet est constitué autour du pattern d'architecture MVC, nos classes sont donc divisibles en trois grande familles, les Modèles comme le board et les entités, les Contrôleurs, symbolisés par nos Game States, et enfin les Vues, qui sont les diverses classes Renderer.

2.1 Characteristics, Types, Capacity et Entity

Ces 3 premières classes permettent de modeler la dernière, une Entité représente le Personnage jouable ou alors un ennemi.

Les Caractéristiques sont un ensemble de valeurs chiffrées qui permettent d'évaluer les forces et faiblesses d'une Entité avec des valeurs tirées du RPG classique (Force, Dextérité, Constitution, Intelligence, Sagesse et Charisme) allant de -4 (extrêmement faible) à 20 (basiquement divin). De ces 5 valeurs (Le jeu ne comportant pas encore de dimension sociale, le Charisme n'a pas été implémenté) dépendent de nombreuses autres valeurs comme les Points de Vie, d'Action ou encore de Déplacements qui permettent de définir les actions que peut prendre cette entité.

Les Types sont un ensemble de valeurs booléennes permettant de traiter l'entité comme un individu d'une classe particulière (pour l'instant, bestial, élémentaire, humain, mythique et mort-vivant), certaines capacités ayant des effets différents en fonction de ces types.

Enfin, les Capacités représentent toutes les actions pouvant être effectuées par le joueur, de l'attaque de base au sort Météore, elles sont toutes représentées avec leur coût en Points d'Action et en éventuels Points de Corruption ainsi que leur portée et leur zone d'effet, enfin, une méthode onHit() permet d'appliquer cette action à toutes les Entités affectées par celle-ci.

2.2 Factory, NormalFactory et Entities

Entities regroupe un ensemble de classes héritant d'Entité ayant des valeurs prédéterminées et nous permet de créer une bibliothèque de différentes Entités, du zombie de base au personnage jouable, toute entité instanciée dans le jeu est déclarée ici.

Par la suite ces Entités sont utilisées dans une classe abstraite Factory et plus précisément dans son héritage NormalFactory afin d'être capable d'instancier toutes ces entités différentes avec un appel à une même fonction. En fonction des paramètres entrés elle nous renvoie une Entité définie dans la bibliothèque que forme Entities. Ce n'est

effectivement pas un pattern Factory Method, ni même Abstract Factory mais plutôt une sorte de Factory Simple. Comme c'était plus simple que de tout renommer, le nom est resté et… en soi, c'est bien de cette "usine" que sortent toutes nos entités donc bon…

Le choix d'une classe abstraite Factory et d'une NormalFactory a été fait dans le but de pouvoir par exemple modifier le niveau de difficulté du jeu simplement en changeant de Factory concrète et en n'ayant aucunement à modifier des lignes précédemment codées.

2.3 Le Board et ses Cases

Le Board est basiquement constitué d'une grille de Cases et des méthodes pour les gérer, il sert uniquement à stocker des données dedans pour pouvoir accéder aux données d'une case.

Une Case n'est pas très complexe non plus, pour l'instant c'est simplement une Entity et un type de case représenté par un entier.

2.4 Les vues/renderer

Les renderer sont des objets très simples, ils sont composés du/des modèles qui les intéressent, par exemple, pour le BoardRenderer, il contient (drumroll) le board en question afin de pouvoir en dessiner les cases de ce dernier, et donc les entités qui se trouvent dessus, un renderer contient uniquement une fonction render qui a en paramètre la fenêtre de jeu générale et dessine dessus.

2.5 Les Game States/controllers

Un petit diagramme d'état est en annexe, théoriquement tous les états sont terminaux, car on peut fermer la fenêtre n'importe quand, mais seuls les états qui gèrent via input la sortie du jeu comme la pause ou game over ont été notés comme terminaux par clarté.

2.5.1 Press Start, Pause et Game Over

Ces états sont les plus simples, ils ne servent que de transition, press start fait démarrer la partie avec la sélection de la position du personnage sur le board, pour l'instant rien d'ultra fancy de ce côté, il n'a donc même pas de méthode update complexe (c'est le cas de ces trois states d'ailleurs), plus ou moins pareil pour pause, cet état doit pouvoir se déclencher à la demande du joueur et sera appelé lorsque l'utilisateur perdra le focus sur la fenêtre, il permettra de reset la partie ou de quitter. Enfin, Game Over est appelé lorsque vous gagnez ou perdez la partie, il est donc accessible depuis PlayerTurn ou EnemyTurn, comme ses camarades, rien de passionnant ici.

2.5.2 SelectPos, PlayerTurn et EnemyTurn

Voici les trois états au coeur du système, et par conséquent les plus complexes. En terme de composition et de méthode, du classique, un Board qui est un pointeur vers un même Board dans les trois états qui a été créé dans l'Engine que l'on verra plus loin, c'est d'ailleurs aussi le cas du Renderer qui est commun aux trois. Encore une fois, un Renderer pour dessiner le tout, et dans le cas du Joueur, un entier pour définir son action courante, afin que l'updater sache quoi faire avec l'input selon le cas. Par exemple, à 0, le clic fera se déplacer, mais si on est sur une compétence, le clic l'activera sur la case ciblée.. Prenons les un par un :

SelectPos va initialiser le Board avec les ennemis, dessiner les joyeux lurons que sont vos ennemis, et va vous proposer des cases où mettre votre perso, parmi lesquelles vous devrez choisir. Une fois ceci fait, vous passerez dans l'état suivant, votre tour !

PlayerTurn, fini la rigolade, à vous de jouer ! L'input handler de cet état va changer l'action courante selon la touche du clavier pressée, et une fois que vous avez cliqué, le renderer va changer partiellement de comportement pour rendre visuellement votre action et l'update va actualiser les données du modèle pour que tout concorde avec ce que l'action doit faire, set les HP des entités, déplacer qui doit l'être, nettoyer les cadavres et autres joyeusetés.

EnemyTurn, là c'est votre tour d'en prendre pour votre grade, les ennemis arrivent vers vous manifestement peu content de votre intrusion. Basiquement un tour ennemi se déroule comme un tour de joueur, à la seule différence qu'une petite sous-méthode sera appelée sur chaque entité autre que le joueur pour la faire jouer automatiquement, cette dernière n'est pas obligatoire, mais simplifie grandement la lecture du code plutôt qu'avoir un gros morceau de code dans un for each.

2.6 L'engine et Game.cpp

L'Engine, c'est là que toute la magie des states se fait, rien de méchant dedans, l'instanciation de tous les états, dont un état courant initialisé sur PressStart, les méthodes pour changer d'état, qui sont appelées dans les différents états au besoin, et une petite redéfinition d'update et render pour les amener vers les versions des States.

Enfin, Game, c'est un petit Game Engine, une petite GameLoop, un peu de sel et de code SFML pour initialiser la fenêtre de jeu qui sera gribouillée par les Renderer tout au long de l'exécution de la GameLoop. Pour la GameLoop, afin de fixer le framerate, la solution finale n'est pas encore décidée, soit utiliser une fonction SFML pour ça, mais qui n'est pas très précise, ou faire soi-même un compteur pour que le framerate puisse être variable selon la puissance permise sans impact sur le gameplay en permettant à la game loop de zapper quelques appels à render pour update correctement le tout et rattraper son retard, je trouve cette solution plus plaisante, mais il faudra faire attention aux imprécisions.

Pour résumer le Game.cpp, on instancie un engine qui va lui-même initialiser le Board et tout ce dont il a besoin, on crée une fenêtre, et tant que cette dernière est ouverte,

on “poll” les évènements via l’Engine puis les states, puis on appelle les méthodes update et render de l’Engine tour à tour, éventuellement, on rappelle update jusqu’à avoir rattrapé son retard, et sauf erreur de notre part, on est prêt pour jouer !

3 - Patterns

Pour réaliser notre projet nous pensions utiliser 4 patterns, un pattern de création Factory, un pattern de structure Decorator et deux patterns de Comportement Observer et State. Finalement, comme vu précédemment, la Factory n’en est pas une, mais ce modèle correspondait plus à ce que l’on avait besoin de faire à l’instanciation d’un mob. (En annexe des petits UML des patterns utilisés)

3.1 Decorator

Très rapidement dans le développement du projet nous avons été amené à chercher un pattern satisfaisant pour représenter les altérations d’état appliquées aux entités. Dans un premier temps, le pattern State nous est apparu comme très intéressant mais c’est au final le pattern Decorator qui fût retenu pour sa capacité à gérer plusieurs altérations au même moment. Et riche de sa flexibilité nous avons décidé de l’utiliser aussi pour gérer buff et debuffs appliqués aux entités.

```
class Decorator : public Entity {
private:
	Entity *base_;

public:
	// Constructeur
	Decorator(Entity *up);

	virtual void applyDecorator();
	virtual string getAlt();

	// Nombreux setters et getters

	~Decorator();
};

Decorator::Decorator(Entity *up) : base_(up) {}
```

Le Décorateur se présente ainsi : à chaque fois qu’une entité a est décorée :

a = new Decorator(a); le constructeur de Decorator est appelé et stocke dans ce décorateur l'entité qu'il va décorer en tant que pointeur.

Chaque fonction d'Entity est redéfinie dans Decorator pour être appliquée sur ce pointeur. Après de nombreux essais, c'est (très tristement) la seule façon que nous avons trouvé pour faire fonctionner le Decorator exactement comme on l'entendait. A savoir :
-toujours conserver de manière intacte la toute première entité telle qu'elle
-être capable de gérer de potentiellement nombreux Decorator sans qu'ils entrent en conflit les uns avec les autres
-gérer les entités décorées et non décorées avec les mêmes fonctions.

3.2 Observer

Notre jeu requiert un affichage, le pattern Observer a donc semblé immédiatement une évidence, on l'utilise dans le cadre d'un pattern d'architecture MVC (qui n'est pas un design pattern). Son principe est des plus simples, l'observer étant unique, on a juste besoin d'une méthode update qui va actualiser les données dont on a besoin. Ensuite, la vue récupère les données après update via le contrôleur pour dessiner tout ça.

Dans un pattern Observer, on parle de sujet et d'observateur, le sujet, ce sont les données attendue par l'observer, c'est à dire notre vue. Le MVC découple tout proprement, mais basiquement, le pattern consiste simplement à l'arrivée d'un event pouvant potentiellement changer les données (ce qui ne fait pas partie du Decorator en soit, c'est juste histoire d'expliquer le fonctionnement), la vue va alors demander régulièrement à être tenue au courant des changements pour tout dessiner. L'Engine sert donc d'intermédiaire, une fois un Event reçu, il update les données, puis appelle le Renderer. Cet Observer a une signature légèrement différente du classique, car la notification est effectuée à chaque frame, et on a un seul observer, donc pas besoin d'add ni de remove.

Depuis la game loop, l'Engine récupère un event, qui est envoyé dans le state courant qui va le traiter pour voir si il en a besoin, et si oui, que faire. Le renderer quand a lui a la donnée qui l'intéresse en attribut, et c'est donc via l'usage des pointeurs et de l'instance utilisée pour plusieurs états et vues que l'actualisation des données pour la vue se fait

3.3 State

C'est là que s'est créée une lutte personnelle, au début, on avait aucune idée de comment rendre différents écrans de jeux via un MVC, c'est passé par des idées comme instancier une multitude de contrôleurs et vues pour chaque écran dont on a besoin, mais ce sont des objets trop lourds à instancier pour être une solution viable à long terme, c'est en s'aventurant sur les sujets parlant des Finite State Machines dans les jeux vidéos que tout est devenu plus clair. Dans le cadre d'un MVC, un state est son propre contrôleur, et les Renderer peuvent être réutilisés, comme dans notre cas, le BoardRenderer utilisé dans trois états, SelectPos, PlayerTurn et EnemyTurn. On évite d'instancier 2 objets par écrans, et le code reste ainsi sans trop de redite entre états, on a juste besoin d'une petite classe qui

contient les transitions pour servir de conteneur à tous ces états instanciés au lancement, c'est à dire la classe Engine.

Je n'ai pas encore eu le temps de l'expérimenter, mais les zones à risque dans ce pattern seront le partage du Board entre certains states, qui, si mal géré peut amener à des erreurs de pointeurs des plus déplaisantes et souvent laborieuses à régler.

4 - Problèmes rencontrés

Evidemment, un projet sans soucis, ça n'existe pas, voici donc une liste des problèmes les plus notables (i.e les plus infâmes/longs à régler) :

Tout d'abord, le temps, pas forcément un manque de temps, mais plus une difficulté à s'imaginer la difficulté, déjà on a mis un certain temps à poser une idée générale de gameplay cible, ensuite, il a fallu faire pas mal de recherches, pour apprendre à coder au moins à peu près correctement en c++ qu'on a pas appris correctement jusqu'ici, surtout en terme de programmation objet.

Pour l'affichage, il a fallu trouver quelle architecture de code utiliser, quelle bibliothèque utiliser... Là encore, du temps a été passé en recherches, tout d'abord sur Qt mais qui ne correspond pas au type de contrôle qui était voulu (donc à nouveau du temps perdu), puis après le changement sur SFML, qui là aussi a causé une perte de temps.

Effectivement, au début, j'utilisais le sous-système Ubuntu installable sur Windows depuis la mise à jour anniversaire, mais problème de "core dumped" à la compilation, des recherches ont donc dû être faites, pour au final, au bout d'environ une semaine (temps de recherche, puis attente d'une réponse sur un forum), effectivement, ce sous-système étant relativement nouveau, peu de questions à son propos et sur SFML se trouvaient sur internet, là encore, du temps perdu, si on y ajoute les divers problèmes que j'ai eu pour installer ce sous-système, j'ai dû passer presque 2 semaines rien que sur ça.

La compréhension du MVC a aussi pris longtemps, même très longtemps. On avait une mauvaise idée de ce que faisaient chacun de ses éléments en lisant un tuto openclassrooms dessus, mais après beaucoup de recherche, on est tombé sur un site avec un exemple de jeu se déroulant sur des cases, exactement ce qu'il nous fallait donc, là encore, pas mal de temps a été passé pour comprendre entièrement le code, mais une fois ceci fait tout était devenu beaucoup plus clair en terme d'affichage, même si bien sûr, du temps a dû être passé à nouveau pour adapter tout ça avec SFML et le c++ car l'exemple de base utilisait un applet Java.

Le fonctionnement de SFML a été un peu laborieux aussi, oscillant entre les tutos officiels, la doc auto généré, les recherches sur la notion de Game Loop, tout ça a pris pas mal de soirées de lecture, encore une fois.

Donc au final, plus une accumulation de choses faisant perdre à chaque fois quelque jours et retardant toujours un peu plus notre avancée, et aussi le côté nouveau de l'exercice des patterns et de la conception de jeux qui est un challenge excitant, mais nouveau pour nous et dont on a donc eu du mal à imaginer la difficulté et la longueur.

Cette mauvaise imagination de la difficulté a fait abandonner beaucoup de fonctionnalités prévues à la base, comme le fameux tour par tour avec un temps par tour, l'expérience aussi, une mappemonde, un inventaire (peut-être fait à temps pour la démo ?), on a aussi laissé à plus tard l'implémentation du pattern ECS qui a l'air intéressant pour la spécialisation d'ennemis.

Autre variable non négligeable quand on code, les bugs, particulièrement avec les pointeurs, même si des erreurs triviales ont pu aussi prendre un temps inversement proportionnel à la complexité du bug. Mais revenons aux pointeurs, ont pensera par exemple aux skills qui reçoivent un lanceur et un receveur, mais quand on appliquait le skill sur le pointeur du receveur, l'entité externe ne subissait pas les modifications, on a donc mis, au lieu de "Entity* receiver" "Entity*& receiver" dans les paramètres de la fonction, et là, les modifications s'appliquent enfin.

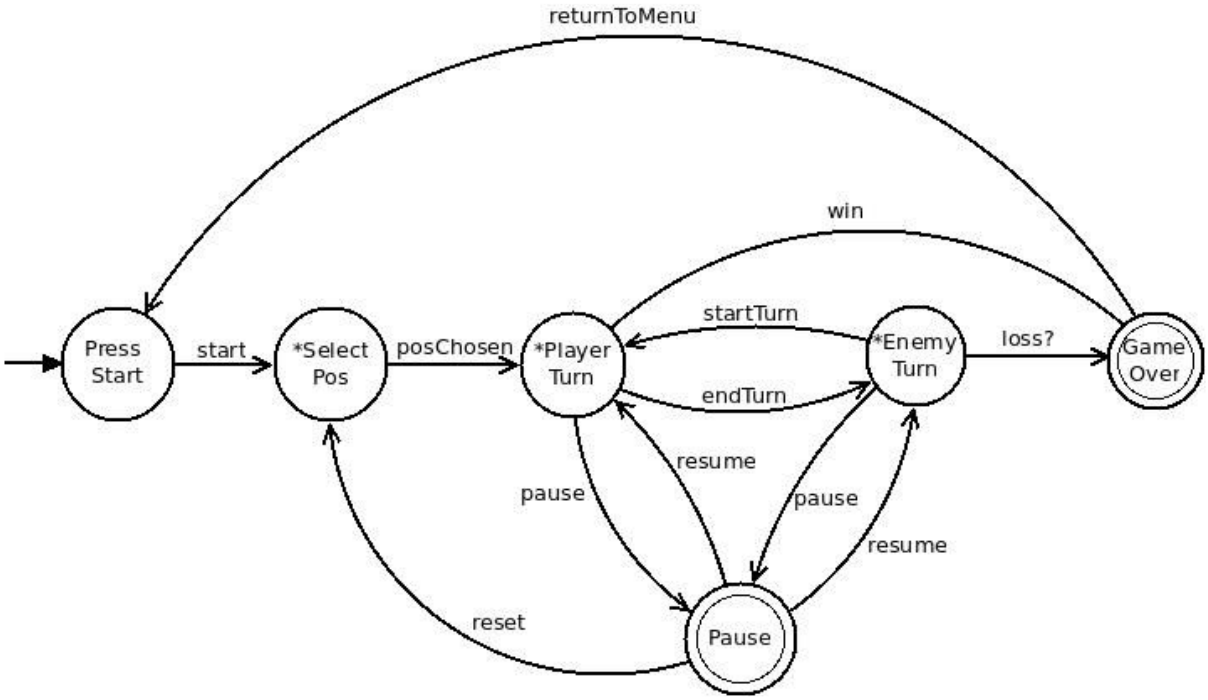
On a eu le même genre de soucis avec le decorator, bon, déjà soucis trivial de faire une division sur des entiers, qui semble simple mais qu'il faut penser à checker. Mais aussi beaucoup de temps passé à l'implémentation, car l'appel récursif directement sur le noyau du decorator ne donnait pas les effets attendus à cause notamment de son héritage des méthodes de l'Entity, il a donc fallu redéfinir dans decorator toute ces méthodes (moult getters, setters et updaters de stats), plusieurs heures ont été passées là-dessus, à tester une implémentation, puis une autre, le tout pour finalement revenir à l'ancienne, la mort dans l'âme.

Conclusion

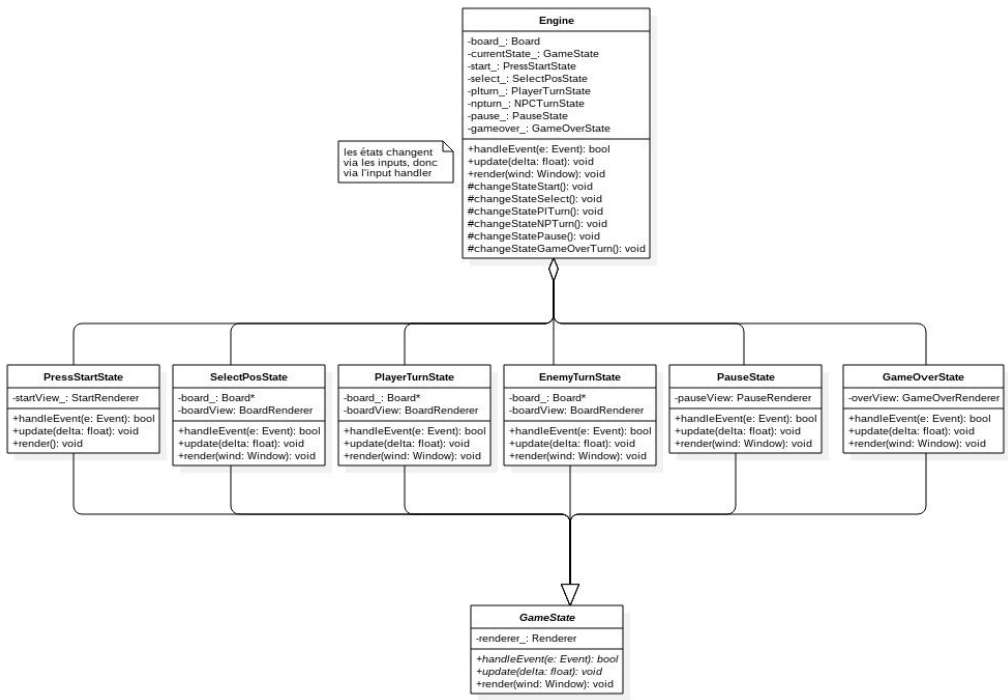
Il s'est avéré que faire un jeu est un bien plus gros challenge que prévu, surtout en tant que débutants complets en la matière, on veut toujours faire de nouvelles fonctionnalités dont on imagine pas la complexité potentielle d'implémentation, sans oublier la fameuse patternite aigüe, dont on espère avoir été préservés (du moins, on a pas l'impression d'avoir fait d'overkill avec nos patterns). De plus, coder enfin un gros projet en c++ n'a vraiment pas le même impact que de faire un petit projet en c++ comme ceux des années passées, on a donc pu se rendre compte de la complexité de ce langage, mais aussi qu'il est aussi polyvalent qu'on pourrait le vouloir, pour peu de savoir s'en servir.

On a pu aussi se rendre compte des nombreuses façon d'implémenter un jeu, par exemple, au lieu d'un MVC, une chain of responsibility aurait pu gérer les évènements, la création d'entités aurait pu être gérée par un prototype de mob de base instancié au préalable, ou aussi par le pattern ECS pour gérer une spécialisation de mobs par composants.

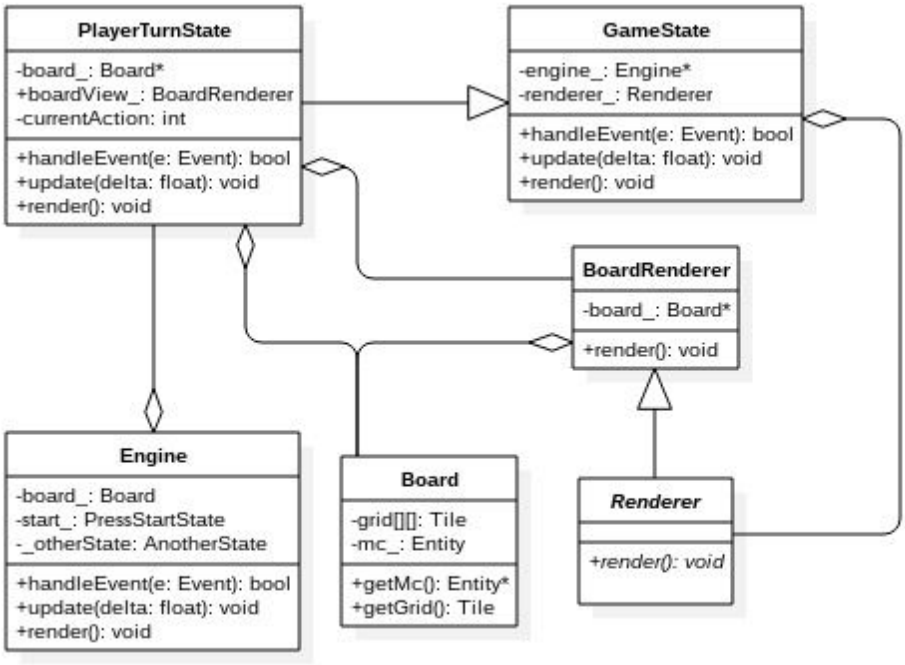
Annexes



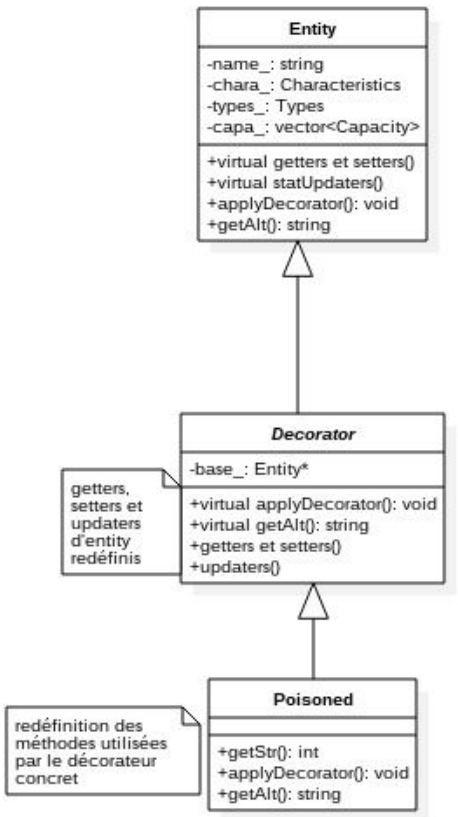
Automate à état fini du jeu



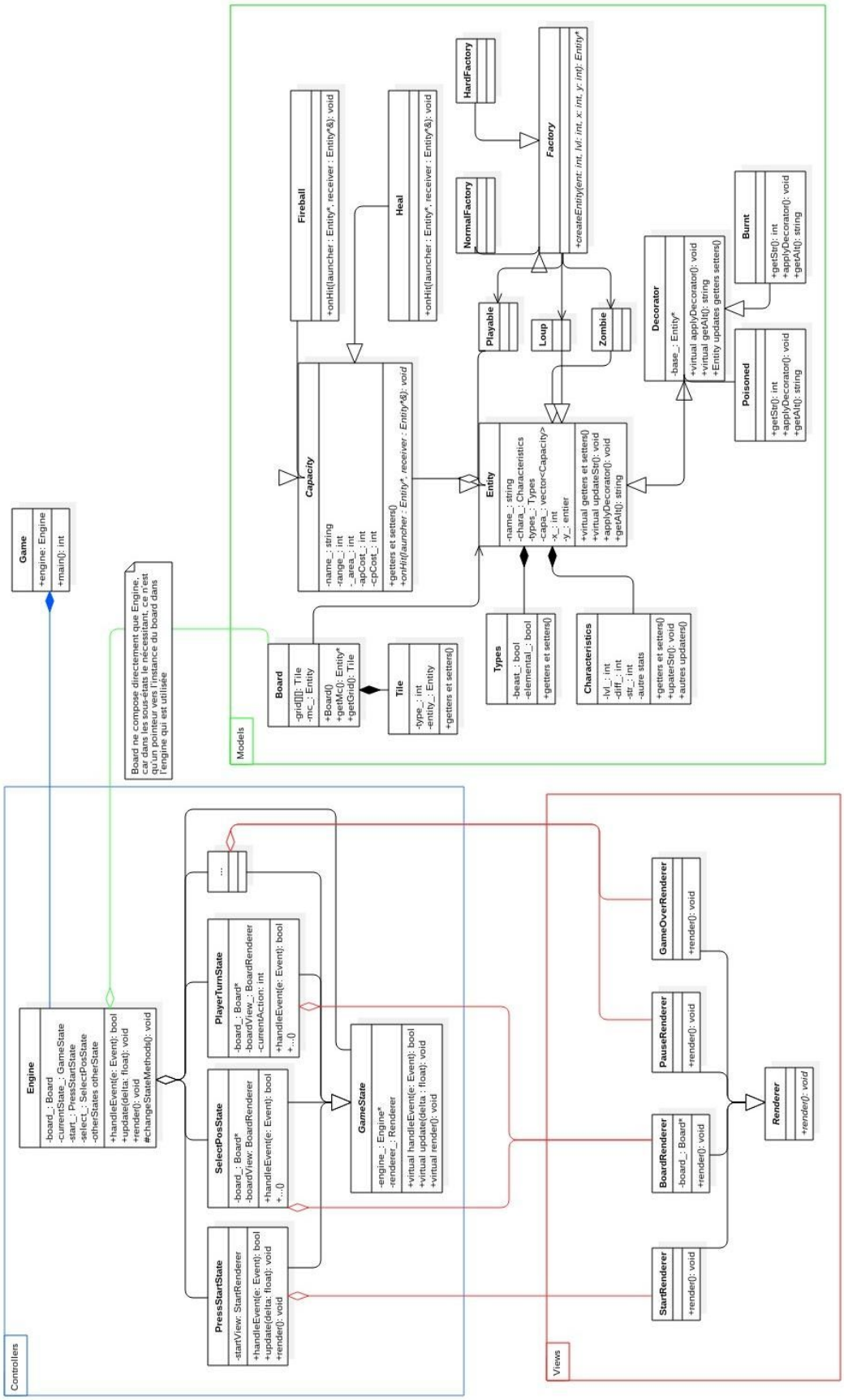
UML détaillé du pattern State



UML détaillé Observer



UML détaillé Decorator



UML Complet du projet selon l'architecture MVC