Projet Logiciel Transversal

Alexandre Génot - Anthony Peloille

FIGURE 1 – The Legend of Zelda (1986)

Table des matières

1 Présentation Générale

1.1 Archétype

Les mécaniques du jeu s'inspirent de The Legend of Zelda (1986) qui est un jeu de type action/aventure.

1.2 Règles du jeu

Le joueur évolue dans un donjon avec plusieurs étages. A chaque étage il doit atteindre un boss, pour cela il se déplace de case en case à la manière d'un jeu de plateau. Les cases peuvent contenir différents types d'évènements : combat, récompense, bonus, malus.... Le but est de sortir du donjon en ayant complété chaque étage, c'est-à-dire avoir vaincu chacun des boss.

Les phases de combats se déroulent sous forme de combats au tour par tour, à chaque tour le personnage peut attaquer (différents coups d'épée) puis le tour suivant, l'ennemi attaque. Chaque personnage (personnage principal ou ennemi) dispose de 3 caractéristiques : - Les points de vie/Health Points, lorsque ceux-ci arrivent à 0 le personnage meurt. - L'attaque , conditionne les dégâts qu'inflige un coup. - La défense, détermine la quantité de point de vie perdue après avoir reçu une attaque.

Le personnage peut, au cours de son aventure, augmenter ses statistiques.

1.3 Ressources

FIGURE 2 – Tuiles du décor et personnages

FIGURE 3 – Police de caractères

2 Description et conception des états

2.1 Description des états

Pour chacun des états du jeu, on retrouve une combinaison d'éléments fixes (la structure du labyrinthe) et d'éléments mobiles (personnage principal et ennemis). Seuls les personnages mobiles possèdent une position sous forme de coordonnées (x et y). Pour les éléments statiques, nous initialisons un tableau avec des éléments. Tous les éléments ont en commun un identifiant qu'est le TypeId (un id pour les murs, un pour les ennemis, le personnage principal ...).

2.1.1 État éléments fixes

Chaque étage du donjon possède un certain nombre fixe de cases. Toutes les cases ont un type parmi ces 4 : Mur, Espace (vide), Porte, Coffre.

Cases "Mur" (Wall) : Elles servent à délimiter les contours de chaque étage du donjon, les personnages ne pouvant pas les traverser.

Cases "Espace" (Space) : Il s'agit des cases sur lesquelles peuvent se déplacer librement les éléments mobiles, on trouve plusieurs sous-types de cases Espace : - Une case de départ où apparaît le joueur. - Une case de fin qui permet au joueur de passer à l'étage suivant (située après le boss de l'étage). - Plusieurs cases "spawn" où apparaissent les ennemis. - Des cases vides de tout élément.

Cases "Porte" (Door) : Ce sont des cases infranchissables par le joueur sauf si le joueur remplit les conditions requises (avoir tué un certain ennemi, avoir obtenu une clé ...), la case devient alors franchissable et donne accès à une nouvelle zone.

Cases "Coffre" (Chest) : Ce dernier type de case correspond à des cases qui permettent au joueur d'obtenir des éléments lui permettant de booster ses statistiques (Vitalité, Attaque ou Défense).

2.1.2 État éléments mobiles

Les éléments mobiles ont 3 caractéristiques, une direction (les 4 points cardinaux et une direction nulle), une position et une variable (isFighting) indiquant si l'élément est en combat ou non. Lorsque la position est à 0 alors l'élément est exactement sur la case, sinon il se trouve entre sa case originale et la case qui correspond à sa direction, quand la position à la même valeur que la vitesse, le personnage se déplace dans la direction correspondante.

Nous avons défini 2 éléments mobiles que sont les ennemis (2 types pour le moment : Ghost, l'ennemi basique et Boss, l'ennemi plus puissant qui permet d'accéder à l'étage suivant) et le personnage principal (MainCharacter) qui n'est autre que le personnage contrôlé par le joueur. Ces éléments mobiles démarrent avec un certain nombre de PV (exemple 6 pv pour le joueur), si leurs points de vie atteignent 0, pour les ennemis, ils disparaissent du jeu et pour le joueur, cela signifie la fin de la partie. Comme détaillé dans la partie règle du jeu, ces éléments disposent de 3 caractéristiques sui déterminent leurs puissances de combat (HealthPoints, Attack et Defense).

2.1.3 État général

On rajoute enfin 2 éléments généraux à ceux définis auparavant, il s'agit des variables epoch et epochRate. La première sert simplement à connaître le nombre de "tour" depuis le début de la partie. La seconde permet de régler la vitesse du jeu en réglant le nombre d'époque par seconde.

2.2 Conception Logiciel

On peut voir notre diagramme des classes sur la page suivante (Figure 4), on y retrouve donc toutes les classes mentionnées auparavant qui permettent de définir un état du jeu : - MobileElement qui permet de définir des éléments mobiles et dont héritent les classes Monstre (ennemis) et MainCharacter (personnage jouable). - StaticElement qui à le même rôle pour les éléments statiques qu'on retrouvera dans les étages de notre donjon : murs, portes, coffres ou "sol" permettant aux personnages de se déplacer. - State qui gère la vitesse du jeu et le nombre de tours effectués. - Il y aussi de nombreuses classes dédiées à l'énumération de statuts ou de types.

Il reste donc 3 classes :

Classe Element : Cette classe permet de définir chacun des éléments du jeu, qu'ils soient statiques ou mobiles. Elle contenait à l'origine les coordonnées (x,y) de chaque élément mais ceux-ci n'étaient utiles que pour les éléments mobiles. Nous avons tout de même gardé la classe element pour sa liaison avec TypeId et pour éviter de changer une grande partie des fonctions déjà implémentés.

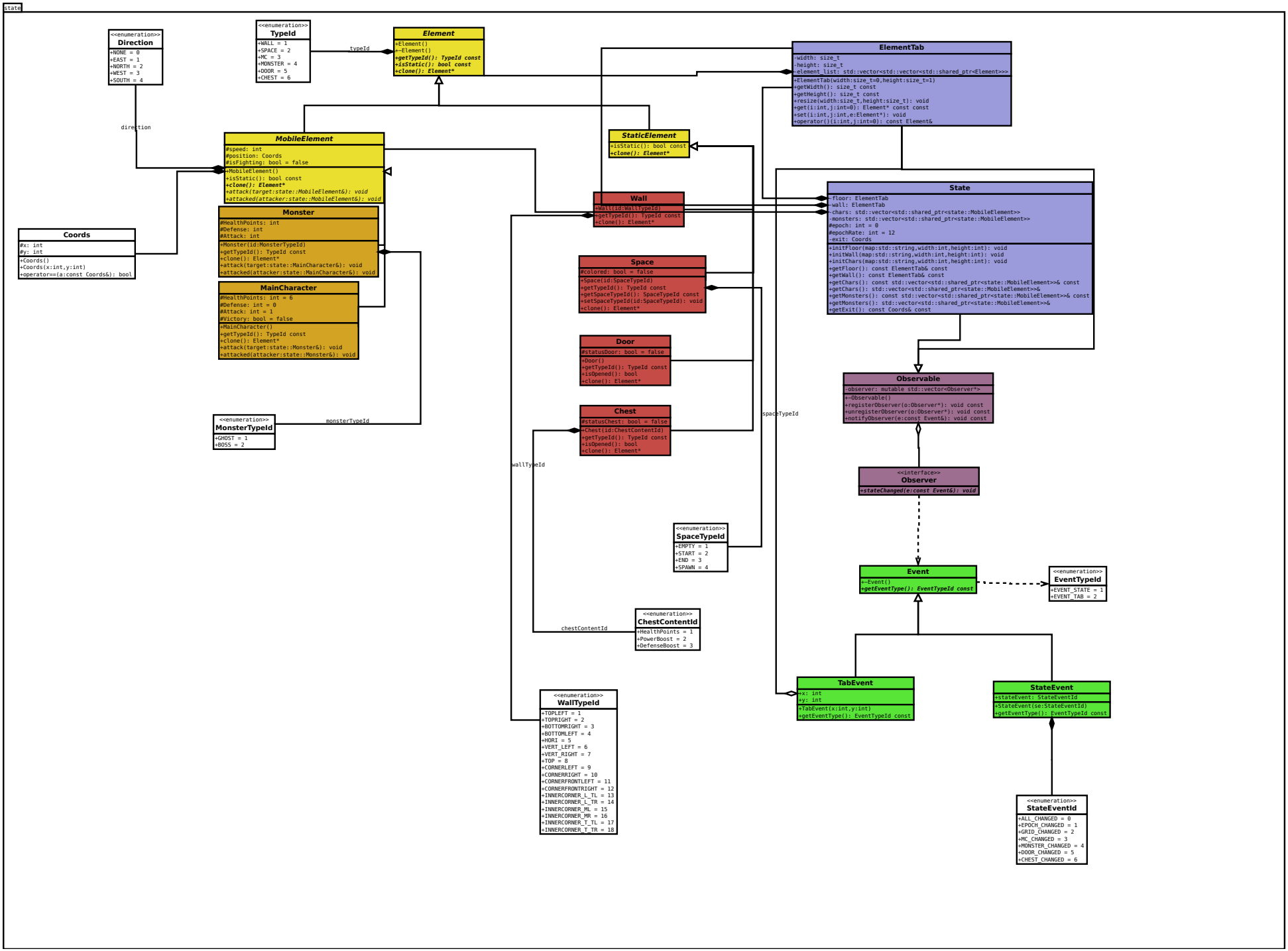
Classe ElementTab : Comme son nom le laisse entendre, on a ici, un conteneur d'éléments qui n'est autre qu'un tableau en deux dimensions. Celui-ci contient donc l'intégralité des éléments dispersés sur un étage du donjon. Son format est un vecteur de vecteur (donc un tableau) de unique ptr, ce format nous a paru pratique puisque nous n'avons pas à préciser la taille dès la définition de l'attribut et le unique ptr nous permet de facilement accéder à chacun des éléments dans le tableau. A noter que le conteneur principal est la classe State qui contient la totalité des données liées à un état donné.

Classe Observable : Le but de cette classe va être de réagir à des modifications de l'état du jeu (donc de la classe state), cette classe s'avère surtout utile pour les éléments de rendus qui mettront à jour les sprites. On a 2 types d'événements : TabEvent et StateEvent qui héritent de la classe Event.

StateEvent correspond aux événements qui touchent ce qu'on trouvera dans la classe StateLayer, c'est-à-dire le personnage principal, les monstres, les éléments annexes affichés (boutons, vitalité du perso, texte ...).

TabEvent correspond de son côté aux éléments contenus dans les tableaux ElementTab, donc les sols, murs, coffres ou portes (il y aura notamment des event liés à l'ouverture d'un coffre ou d'une porte).

Il est à noter qu'afin de pouvoir fabriquer des nouvelles instances d'éléments, nous avons décidé d'implémenter une méthode clone() pour chacun des éléments (statiques ou mobiles) que nous pouvons construire. L'implémentation de cette méthode répond à un problème lié au format de notre tableau d'élément (vector(vector(unique ptr(Element)))), lorsque nous souhaitons ajouter un élément au tableau, l'élément doit être cloné avant d'être ajouté au tableau sinon celui-ci est doublement détruit (double free).


state
<<enumeration>>
Direction
+NONE = 0
+EAST = 1
+NORTH = 2
+WEST = 3
+SOUTH = 4
direction
<<enumeration>>
TypeId
+WALL = 1
+SPACE = 2
+MC = 3
+MONSTER = 4
+DOOR = 5
+CHEST = 6
typeId
Element
+Element()
+~Element()
+getTypeId(): TypeId const
+isStatic(): bool const
+clone(): Element*
ElementTab
-width: size_t
-height: size_t
-element_list: std::vector<std::vector<std::shared_ptr<Element>>>
+ElementTab(width:size_t=0,height:size_t=1)
+getWidth(): size_t const
+getHeight(): size_t const
+resize(width:size_t,height:size_t): void
+get(i:int,j:int=0): Element* const const
+set(i:int,j:int,e:Element*): void
+operator()(i:int,j:int=0): const Element&
MobileElement
#speed: int
#position: Coords
#isFighting: bool = false
+MobileElement()
+isStatic(): bool const
+clone(): Element*
+attack(target:state::MobileElement&): void
+attacked(attacker:state::MobileElement&): void
StaticElement
+isStatic(): bool const
+clone(): Element*
Coords
#x: int
#y: int
+Coords()
+Coords(x:int,y:int)
+operator==(a:const Coords&): bool
Monster
#HealthPoints: int
#Defense: int
#Attack: int
+Monster(id:MonsterTypeId)
+getTypeId(): TypeId const
+clone(): Element*
+attack(target:state::MainCharacter&): void
+attacked(attacker:state::MainCharacter&): void
MainCharacter
#HealthPoints: int = 6
#Defense: int = 0
#Attack: int = 1
#Victory: bool = false
+MainCharacter()
+getTypeId(): TypeId const
+clone(): Element*
+attack(target:state::Monster&): void
+attacked(attacker:state::Monster&): void
<<enumeration>>
MonsterTypeId
+GHOST = 1
+BOSS = 2
monsterTypeId
Wall
+Wall(id:WallTypeId)
+getTypeId(): TypeId const
+clone(): Element*
Space
#colored: bool = false
+Space(id:SpaceTypeId)
+getTypeId(): TypeId const
+getSpaceTypeId(): SpaceTypeId const
+setSpaceTypeId(id:SpaceTypeId): void
+clone(): Element*
Door
#statusDoor: bool = false
+Door()
+getTypeId(): TypeId const
+isOpened(): bool
+clone(): Element*
Chest
#statusChest: bool = false
+Chest(id:ChestContentId)
+getTypeId(): TypeId const
+isOpened(): bool
+clone(): Element*
spaceTypeId
wallTypeId
chestContentId
<<enumeration>>
SpaceTypeId
+EMPTY = 1
+START = 2
+END = 3
+SPAWN = 4
<<enumeration>>
ChestContentId
+HealthPoints = 1
+PowerBoost = 2
+DefenseBoost = 3
<<enumeration>>
WallTypeId
+TOPLEFT = 1
+TOPRIGHT = 2
+BOTTOMRIGHT = 3
+BOTTOMLEFT = 4
+HORI = 5
+VERT_LEFT = 6
+VERT_RIGHT = 7
+TOP = 8
+CORNERLEFT = 9
+CORNERRIGHT = 10
+CORNERFRONTLEFT = 11
+CORNERFRONTRIGHT = 12
+INNERCORNER_L_TL = 13
+INNERCORNER_L_TR = 14
+INNERCORNER_ML = 15
+INNERCORNER_MR = 16
+INNERCORNER_T_TL = 17
+INNERCORNER_T_TR = 18
State
-floor: ElementTab
-wall: ElementTab
-chars: std::vector<std::shared_ptr<state::MobileElement>>
-monsters: std::vector<std::shared_ptr<state::MobileElement>>
#epoch: int = 0
#epochRate: int = 12
-exit: Coords
+initFloor(map:std::string,width:int,height:int): void
+initWall(map:std::string,width:int,height:int): void
+initChars(map:std::string,width:int,height:int): void
+getFloor(): const ElementTab& const
+getWall(): const ElementTab& const
+getChars(): const std::vector<std::shared_ptr<state::MobileElement>>& const
+getChars(): std::vector<std::shared_ptr<state::MobileElement>>&
+getMonsters(): const std::vector<std::shared_ptr<state::MobileElement>>& const
+getMonsters(): std::vector<std::shared_ptr<state::MobileElement>>&
+getExit(): const Coords& const
Observable
-observer: mutable std::vector<Observer*>
+~Observable()
+registerObserver(o:Observer*): void const
+unregisterObserver(o:Observer*): void const
+notifyObserver(e:const Event&): void const
<<interface>>
Observer
+stateChanged(e:const Event&): void
Event
+~Event()
+getEventType(): EventTypeId const
<<enumeration>>
EventTypeId
+EVENT_STATE = 1
+EVENT_TAB = 2
TabEvent
+x: int
+y: int
+TabEvent(x:int,y:int)
+getEventType(): EventTypeId const
StateEvent
+stateEvent: StateEventId
+StateEvent(se:StateEventId)
+getEventType(): EventTypeId const
<<enumeration>>
StateEventId
+ALL_CHANGED = 0
+EPOCH_CHANGED = 1
+GRID_CHANGED = 2
+MC_CHANGED = 3
+MONSTER_CHANGED = 4
+DOOR_CHANGED = 5
+CHEST_CHANGED = 6

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Pour la stratégie de rendu d'un état, nous avons choisi de diviser les éléments que nous affichons à l'écran en plusieurs plans/layers.

Le premier plan nommé floorLayer contient le sol, il recouvre la totalité de la map avec des éléments de type Floor. Le second plan nommé wallLayer contient les autres éléments statiques : mur, coffres ou portes. Enfin, il reste un dernier plan stateLayer qui sert à afficher les personnages (personnage principal ou monstre) contenus dans des listes et non des tableaux car ils possèdent des coordonnées en argument, ce plan sert aussi à afficher des éléments secondaires comme une barre de vie, du texte, des boutons pour le déplacement ou l'attaque ...

Pour créer notre map, nous utilisons des fichiers .txt contenus dans le dossier src (donc pas à la racine du projet), pour l'instant nous avons écrit 2 fichiers txt, un par layer que nous avons crée (Floor et Wall), à noter que le fichier wall.txt contient des informations sur tous les éléments affichés à l'écran autres que le sol. Nous avons affecté à chacune des tuiles un ID qui se lit de la manière suivante : - Le premier chiffre correspond au TypeID de l'objet (1 = wall, 2 = space ...). - Le second (peut être supérieur à 10) correspond au sous-type d'élément, par exemple si c'est un mur de type vertical, on a le WallTypeID vert = 6 (on écrira alors 16 dans le fichier txt). Si c'est un élément space "vide" on a le SpaceTypeID empty = 1 (on écrira cette fois-ci 21 dans le fichier txt). Ces fichiers texte sont lus dans les fonctions initWall et initFloor de la classe State, ces méthodes initialise des tableaux d'éléments à partir du contenu de ces fichiers avant de les transmettre aux classes utilisées pour le rendu.

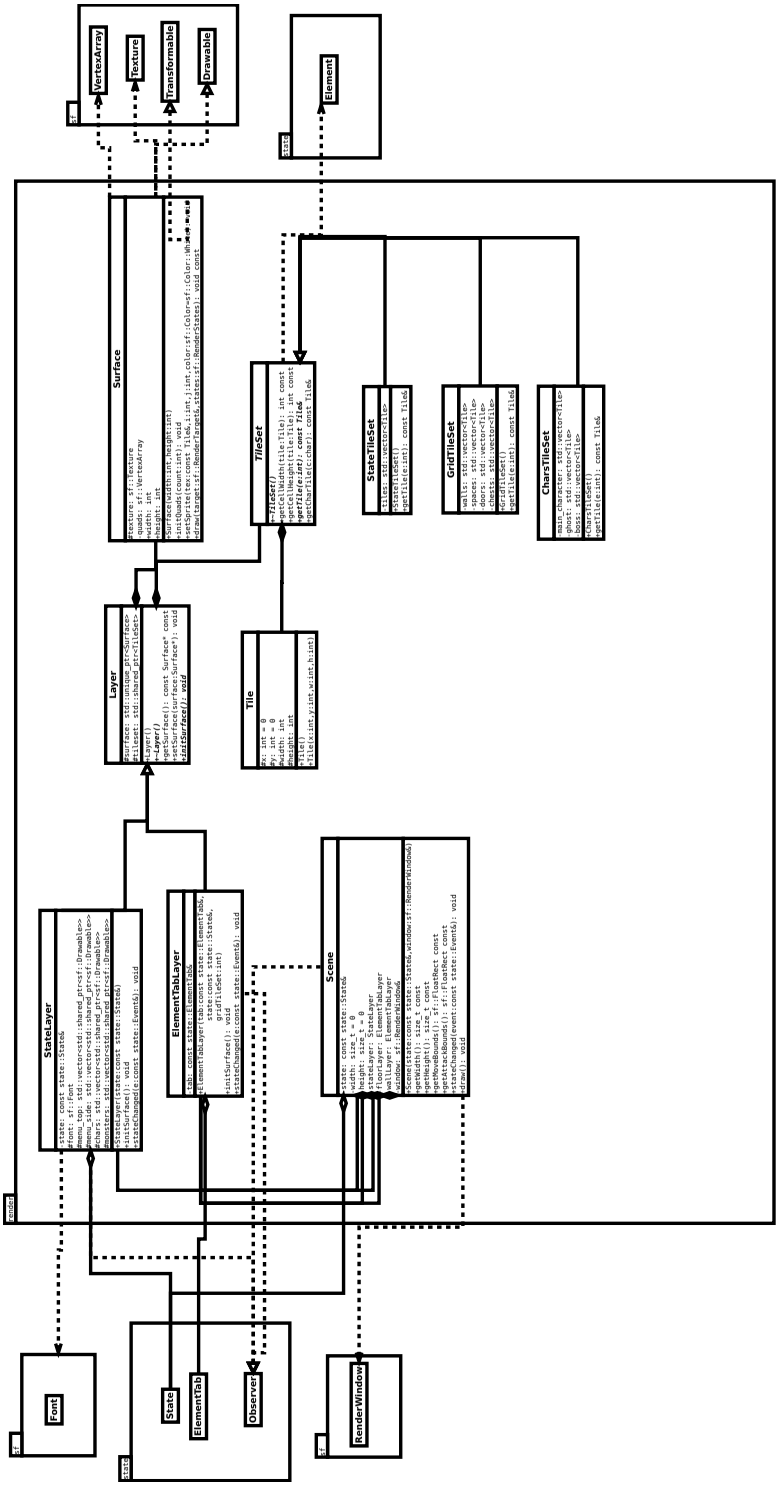
3.2 Conception logiciel

Le diagramme des classes pour notre rendu est visible sur la page suivante.

ElementTabLayer et State Layer : Il s'agit des 2 classes héritant de la classe Layer et permettant de définir les plans que nous allons afficher, les valeurs contenus dans ces tableaux sont définis auparavant dans la classe State. Ces informations sont données à une instance de Surface et la définition des tuiles est contenue dans une instance de TileSet. La méthode InitSurface() fait la plus grosse partie du travail : elle fabrique une nouvelle surface puis initialise le nombre de quads/sprites avec la méthode InitQuads() avant de fixer une tuile et une position pour chacun des sprites avec la méthode setSprite().

Surfaces : Chaque surface contient une texture du plan et un nombre de quadruplets de vecteurs 2D. Les éléments texCoords de chacun des quadruplets contient les coordonnées de 4 coins de la tuile à sélectionner dans la texture. Les éléments position de chaque quadruplet contiennent les coordonnées des quatre coins du carré où doit être dessiné la tuile à l'écran.

Tuiles : Les classes intéressantes sont les classes filles de la classe TileSet, ces classes contiennent des listes contenant toutes les tuiles d'un certain élément à afficher, par exemple, la liste walls de GridTileSet contient une liste des tuiles correspondant à tous les différents types de murs (coin en haut à gauche, à droite, mur horizontal, vertical ...). Toute les tuiles sont définies à l'aide de 4 entiers qui représentent sa position dans l'image source (x,y) et sa taille (width, height). Pour obtenir n'importe laquelle de ces tuiles, nous utilisons la fonction getTile() qui prend en argument un ID, le fonctionnement de ces IDs est détaillé dans la partie précédente.


sf
VertexArray
Texture
Transformable
Drawable
state
Element
Surface
TileSet
StateTileSet
GridTileSet
CharsTileSet
Layer
Tile
StateLayer
ElementTabLayer
Scene
RenderWindow
Font
State
ElementTab
Observer

4 Règles de changement d'états et moteur de jeu

4.1 Changements extérieurs

Nous avons des changements liés à des facteurs extérieurs, notamment avec la souris : - Une pression avec la souris sur le bouton "déplacement" (sur la droite de l'écran) affiche les possibilités de déplacement pour le personnage principal (les cases prennent une couleur verte) - Une pression sur l'une de ces cases (donc uniquement après avoir pressé le bouton "déplacement") provoque le déplacement du personnage sur la case sélectionné. - Lorsqu'un monstre se trouve sur une case adjacente de celle sur laquelle se trouve le personnage principal (pas les cases en diagonale), le joueur peut presser le bouton attaque pour infliger des dégâts au monstre, après plusieurs attaques, les pv de celui-ci tombent à 0 et le monstre disparaît de l'affichage.

4.2 Changements autonomes

Certains changements sont appliquées à chaque création ou mise à jour d'un état : - Si le joueur se déplace sur une case qui se trouve à côté d'un monstre, alors il ne peut plus se déplacer avant la fin du combat (la mort du monstre), une fois le monstre vaincu et disparu, le personnage peut à nouveau se déplacer. - Le joueur ne peut pas se déplacer sur des cases de type "mur" - Les pv d'un monstre attaqué sont modifiés et s'ils arrivent à 0, le monstre disparaît de l'affichage.

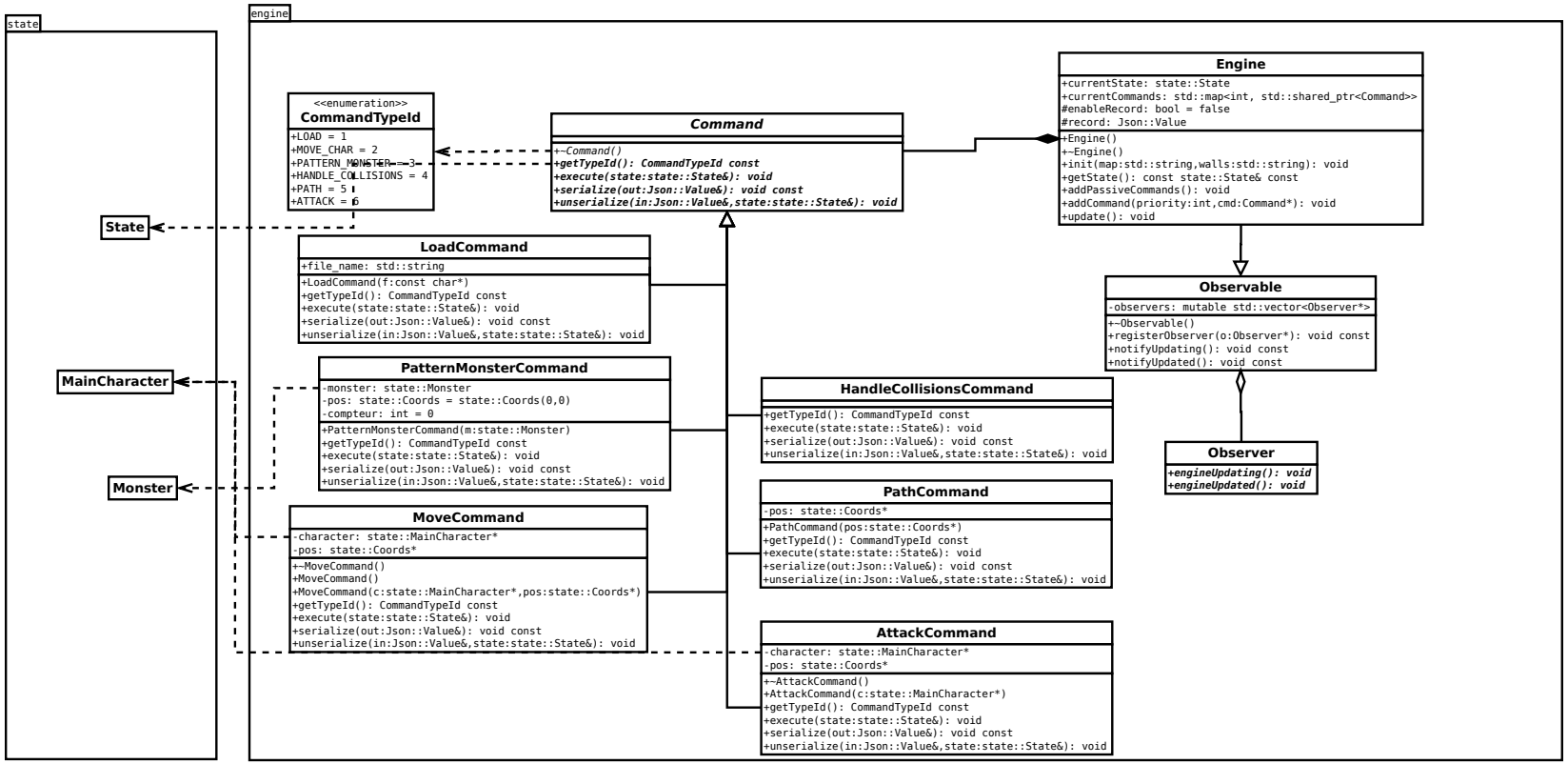
4.3 Conception logiciel

Le diagramme des classes pour notre moteur de jeu est visible sur la page suivante. Celui-ci s'appuie sur différentes classes de type Command et a pour but l'exécution, dans un certain ordre, de différents types de commandes extérieures sur l'état du jeu.

Classe Command et ses classes filles : Chacune de ces classe vise à représenter un certain type de commande qui changera d'une manière ou d'une autre l'état de notre jeu, il est à noter que ce n'est pas dans ces classes que se déroule la gestion des entrées utilisateurs (appui sur une touche du clavier, de la souris ..). On à également défini un type CommandTypeId pour identifier précisément la classe de n'importe quelle instance d'une commande. Voici une description des différentes classes de commandes qui se trouvent actuellement dans notre diagramme des classes : - MoveCommand, il s'agit de la classe qui gère les déplacements de notre personnage. - PathCommand, cette classe s'occupe d'illuminer en vert les cases sur lesquelles le joueur peut se déplacer (n'illumine donc pas les cases "mur"). - AttackCommand, permet d'attaquer un monstre se trouvant sur une case adjacente (pas les diagonales) Il reste 2 classes pas encore implémentés : - LoadCommand, permet de charger une carte à partir d'un fichier, pour l'instant nous n'avons qu'une seule carte qui est chargée dès le lancement du jeu donc elle n'est pas encore implémenté. - HandleCollisionsCommand, gère les collisions si le personnage et le monstre se trouvent sur la même case.

Classe engine : Cette classe représente le cœur de notre moteur de jeu. L'attribut currentState stocke l'état du jeu et l'attribut currentCommands est une std : :map qui stocke les commandes à exécuter. Grâce au format std : :map, on introduit une priorité entre les commandes, les clés représentent la priorité, et plus elles sont petites, plus la commande sera exécuté tôt dans la liste . A chaque appel de la méthode update(), le moteur va appeler la méthode execute de chacune des commandes stockées dans currentCommands puis nettoyer la liste (retirer toutes les commandes).

A noter que les events liés à des actions du joueur (appui sur le clavier, souris ...) sont pour l'instant traitées dans le main(), nous comptons déplacer toute cette partie dans une classe dédiée à cela dans le render.


state
engine
<<enumeration>>
CommandTypeId
+LOAD = 1
+MOVE_CHAR = 2
+PATTERN_MONSTER = 3
+HANDLE_COLLISIONS = 4
+PATH = 5
+ATTACK = 6
Command
+~Command()
+getTypeId(): CommandTypeId const
+execute(state:state::State&): void
+serialize(out:Json::Value&): void const
+unserialize(in:Json::Value&,state:state::State&): void
Engine
+currentState: state::State
+currentCommands: std::map<int, std::shared_ptr<Command>>
#enableRecord: bool = false
#record: Json::Value
+Engine()
+~Engine()
+init(map:std::string,walls:std::string): void
+getState(): const state::State& const
+addPassiveCommands(): void
+addCommand(priority:int,cmd:Command*): void
+update(): void
State
LoadCommand
+file_name: std::string
+LoadCommand(f:const char*)
+getTypeId(): CommandTypeId const
+execute(state:state::State&): void
+serialize(out:Json::Value&): void const
+unserialize(in:Json::Value&,state:state::State&): void
Observable
-observers: mutable std::vector<Observer*>
+~Observable()
+registerObserver(o:Observer*): void const
+notifyUpdating(): void const
+notifyUpdated(): void const
MainCharacter
PatternMonsterCommand
-monster: state::Monster
-pos: state::Coords = state::Coords(0,0)
-compteur: int = 0
+PatternMonsterCommand(m:state::Monster)
+getTypeId(): CommandTypeId const
+execute(state:state::State&): void
+serialize(out:Json::Value&): void const
+unserialize(in:Json::Value&,state:state::State&): void
HandleCollisionsCommand
+getTypeId(): CommandTypeId const
+execute(state:state::State&): void
+serialize(out:Json::Value&): void const
+unserialize(in:Json::Value&,state:state::State&): void
Observer
+engineUpdating(): void
+engineUpdated(): void
Monster
PathCommand
-pos: state::Coords*
+PathCommand(pos:state::Coords*)
+getTypeId(): CommandTypeId const
+execute(state:state::State&): void
+serialize(out:Json::Value&): void const
+unserialize(in:Json::Value&,state:state::State&): void
MoveCommand
-character: state::MainCharacter*
-pos: state::Coords*
+~MoveCommand()
+MoveCommand()
+MoveCommand(c:state::MainCharacter*,pos:state::Coords*)
+getTypeId(): CommandTypeId const
+execute(state:state::State&): void
+serialize(out:Json::Value&): void const
+unserialize(in:Json::Value&,state:state::State&): void
AttackCommand
-character: state::MainCharacter*
-pos: state::Coords*
+~AttackCommand()
+AttackCommand(c:state::MainCharacter*)
+getTypeId(): CommandTypeId const
+execute(state:state::State&): void
+serialize(out:Json::Value&): void const
+unserialize(in:Json::Value&,state:state::State&): void

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence artificielle basée sur l'aléatoire

Comme son nom l'indique, cette première intelligence artificielle est très basique et repose uniquement sur de l'aléatoire, à chaque nouveau tour, le personnage se déplace sur une case adjacente aléatoire

5.1.2 Intelligence artificielle basée sur des heuristiques

Cette intelligence artificielle est déjà plus développée que la précédente puisque cette fois-ci, les commandes possibles pour le personnage seront affectées d'un poids. Ce poids permettra d'effectuer des mouvements qui l'aideront à remplir son objectif : combattre et tuer un monstre disposé sur la map.

5.1.3 Intelligence artificielle avancée

L'IA avancée a pour but d'utiliser des algorithmes de résolution de problèmes à états finis vus en cours d'algorithmique. Nous choisissons donc d'utiliser un algorithme minmax pour obtenir une intelligence artificielle plus complexe, ceppendant, celle-ci s'avère aussi beaucoup plus gourmande en terme de calculs, c'est pourquoi, en cas de quantité trop importante, on bascule sur l'IA heuristique afin de tout de me voir notre personnage bouger et réaliser son objectif.

5.2 Conception logiciel

Le diagramme des classes pour notre intelligence artificielle (ai.dia) est visible sur la page suivante.

La classe AI est la classe mère de toutes les différentes intelligences artificielles que nous implémenterons dans différentes classes :

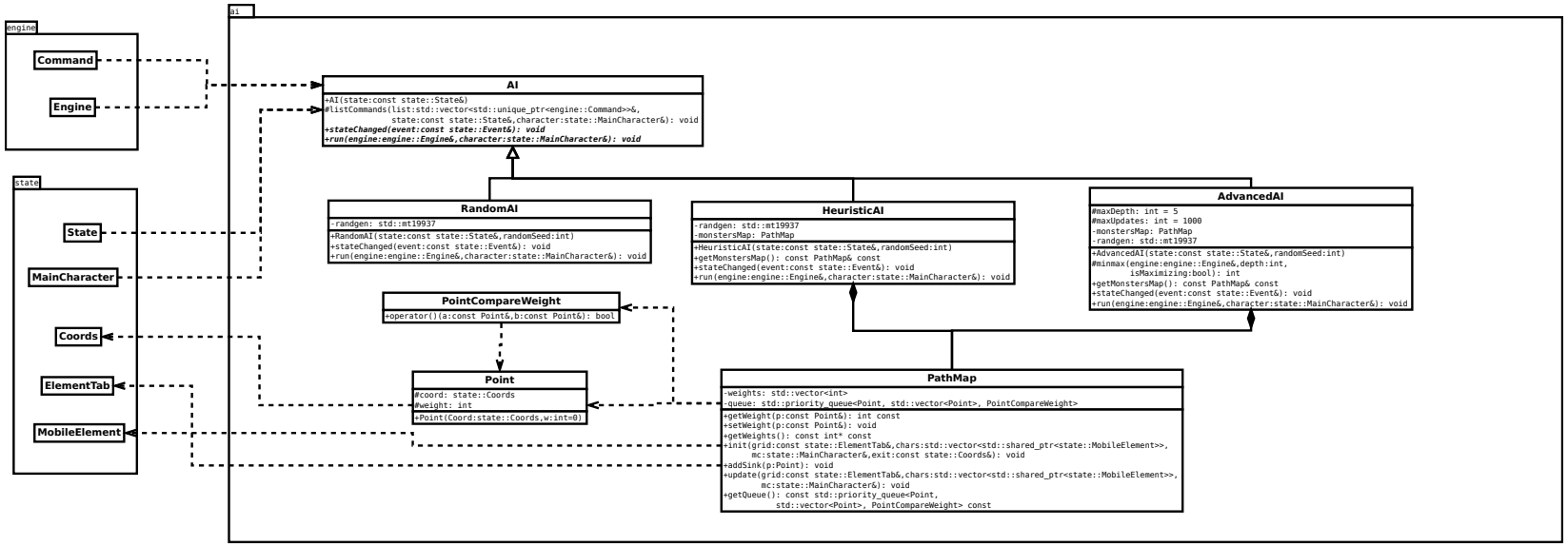
- RandomAI : intelligence aléatoire, son fonctionnement est très peu complexe et cette classe ne nécessite qu'un attribut généré aléatoirement (ici randgen).

-HeuristicAI : L'IA heuristique repose sur l'attribution de poids aux positions auxquelles le personnage peut accéder (les cases adjacentes), on définit donc l'attribut weight dans une classe Point qui importe les coordonnées (notée coord). La classe PathMap permet ensuite d'attribuer des poids aux points en fonction de la distance qui sépare le point de l'objectif du personnage (les monstres), plus la distance est petite, plus le poids est élevé.

-AdvancedAI : Les algorithmes de notre IA avancée se divisent en 2 parties, la première partie consiste simplement à reprendre les algorithmes de l'IA heuristique pour s'assurer d'obtenir un comportement logique de notre IA même dans le cas où le minmax requiert trop de ressources/calculs. Pour savoir quand basculer d'une IA à l'autre, on utilise 2 variables définies dans la classe AdvancedAI que sont maxDepth, (pour la profondeur de l'arbre de recherche) et maxUpdates (pour le nombre d'étapes de recherche dans l'arbre).

Deuxièmement, une méthode minmax qui reprend l'algorithme vu en cours pour calculer les actions optimales à réaliser pour notre personnage en utilisant un système de score, nous nous sommes particulièrement inspirés d'un exemple sur un jeu de morpion.Cette méthode est récursive et va étudier toutes les cases autour de notre personnage.

Finalement, nous n'avons pas réussi à implémenter un minmax qui fonctionne, nous pensons que le problème est dû au trop grand nombre d'appels récursifs de la fonction minmax et au trop grand nombre de vérifications à effectuer aux alentours de notre personnage.


engine
Command
Engine
state
State
MainCharacter
Coords
ElementTab
MobileElement
ai
AI
+AI(state:const state::State&)
#listCommands(list:std::vector<std::unique_ptr<engine::Command>>&,
state:const state::State&,character:state::MainCharacter&): void
+stateChanged(event:const state::Event&): void
+run(engine:engine::Engine&,character:state::MainCharacter&): void
RandomAI
-randgen: std::mt19937
+RandomAI(state:const state::State&,randomSeed:int)
+stateChanged(event:const state::Event&): void
+run(engine:engine::Engine&,character:state::MainCharacter&): void
HeuristicAI
-randgen: std::mt19937
-monstersMap: PathMap
+HeuristicAI(state:const state::State&,randomSeed:int)
+getMonstersMap(): const PathMap& const
+stateChanged(event:const state::Event&): void
+run(engine:engine::Engine&,character:state::MainCharacter&): void
AdvancedAI
#maxDepth: int = 5
#maxUpdates: int = 1000
-monstersMap: PathMap
-randgen: std::mt19937
+AdvancedAI(state:const state::State&,randomSeed:int)
#minmax(engine:engine::Engine&,depth:int,
isMaximizing:bool): int
+getMonstersMap(): const PathMap& const
+stateChanged(event:const state::Event&): void
+run(engine:engine::Engine&,character:state::MainCharacter&): void
PointCompareWeight
+operator()(a:const Point&,b:const Point&): bool
Point
#coord: state::Coords
#weight: int
+Point(Coord:state::Coords,w:int=0)
PathMap
-weights: std::vector<int>
-queue: std::priority_queue<Point, std::vector<Point>, PointCompareWeight>
+getWeight(p:const Point&): int const
+setWeight(p:const Point&): void
+getWeights(): const int* const
+init(grid:const state::ElementTab&,chars:std::vector<std::shared_ptr<state::MobileElement>>,
mc:state::MainCharacter&,exit:const state::Coords&): void
+addSink(p:Point): void
+update(grid:const state::ElementTab&,chars:std::vector<std::shared_ptr<state::MobileElement>>,
mc:state::MainCharacter&): void
+getQueue(): const std::priority_queue<Point,
std::vector<Point>, PointCompareWeight> const

6 Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

On cherche à modulariser les tâches en créant un nouveau thread dans la partie client qui va s'occuper de gérer le moteur de notre jeu. D'autre part, on va utiliser la librairie Json pour sérialiser toutes nos commandes afin d'avoir la possibilité de les exporter facilement.

6.1.2 Répartition sur différentes machines : création d'un lobby

Afin de pouvoir jouer en réseau, la première étape consiste à créer une liste de clients/joueurs pour le serveur. Afin de ce faire, on va utiliser une API web avec plusieurs types de requêtes (basées sur le modèle REST) :
Liste des requêtes possibles :

GET /player/<id>

- Si le joueur <id> existe : Statut OK et retourne {"name" :"player_name"}
- Si le joueur <id> n'existe pas : Statut NOT_FOUND

PUT /player
Données en entrée : {"name" :"player_name"}

- S'il reste une place libre : Statut CREATED et retourne {"id" :"player_id"}
- Sinon Statut OUT_OF_RESSOURCES

POST /player/<id>
Données en entrée : {"name" :"player_name"}

- Si le joueur <id> existe : Statut NO_CONTENT
- Si le joueur <id> n'existe pas : Statut NOT_FOUND

DELETE /player/<id>

- Si le joueur <id> existe : Statut NO_CONTENT et joueur retiré
- Si le joueur <id> n'existe pas : Statut NOT_FOUND

GET /game

- S'il y a des joueurs dans le lobby : Statut OK et retourne un tableau json contenant les noms des joueurs
- S'il n'y a pas de joueurs dans le lobby : Statut OK et retourne null

6.1.3 Répartition sur différentes machines : échange des commandes

Concernant la gestion des commandes, elles ne sont plus exécutées localement mais sur le serveur. Lorsqu'un joueur effectue une action, la commande associée est sérialisée puis envoyée au serveur et c'est

lui qui s'en occupe : le moteur étant sur le serveur, la commande y est effectué et le joueur peut voir le nouvel état du jeu. On rajoute la requête suivante à l'API afin d'envoyer les commandes au serveur :

PUT /commands
Données en entrée : La commande serialisée.

Renvoie dans tous les cas le Statut CREATED.

6.2 Conception logiciel

Pour la modularisation, le travail se passe dans la classe client, dans celle-ci, on crée un nouveau thread qui appelle une fonction thread engine, cette fonction va simplement effectuer le travail du moteur, c'est-à-dire appeler la fonction update qui se charge de remettre à jour les éléments du jeu. On a d'ailleurs à présent un deuxième système d'oserveur en plus de celui que nous utilisions pour le rendu, ce deuxième observeur relie l'engine au client pour s'assurer que toutes les commandes soient bien par notre moteur dans son thread dédié.

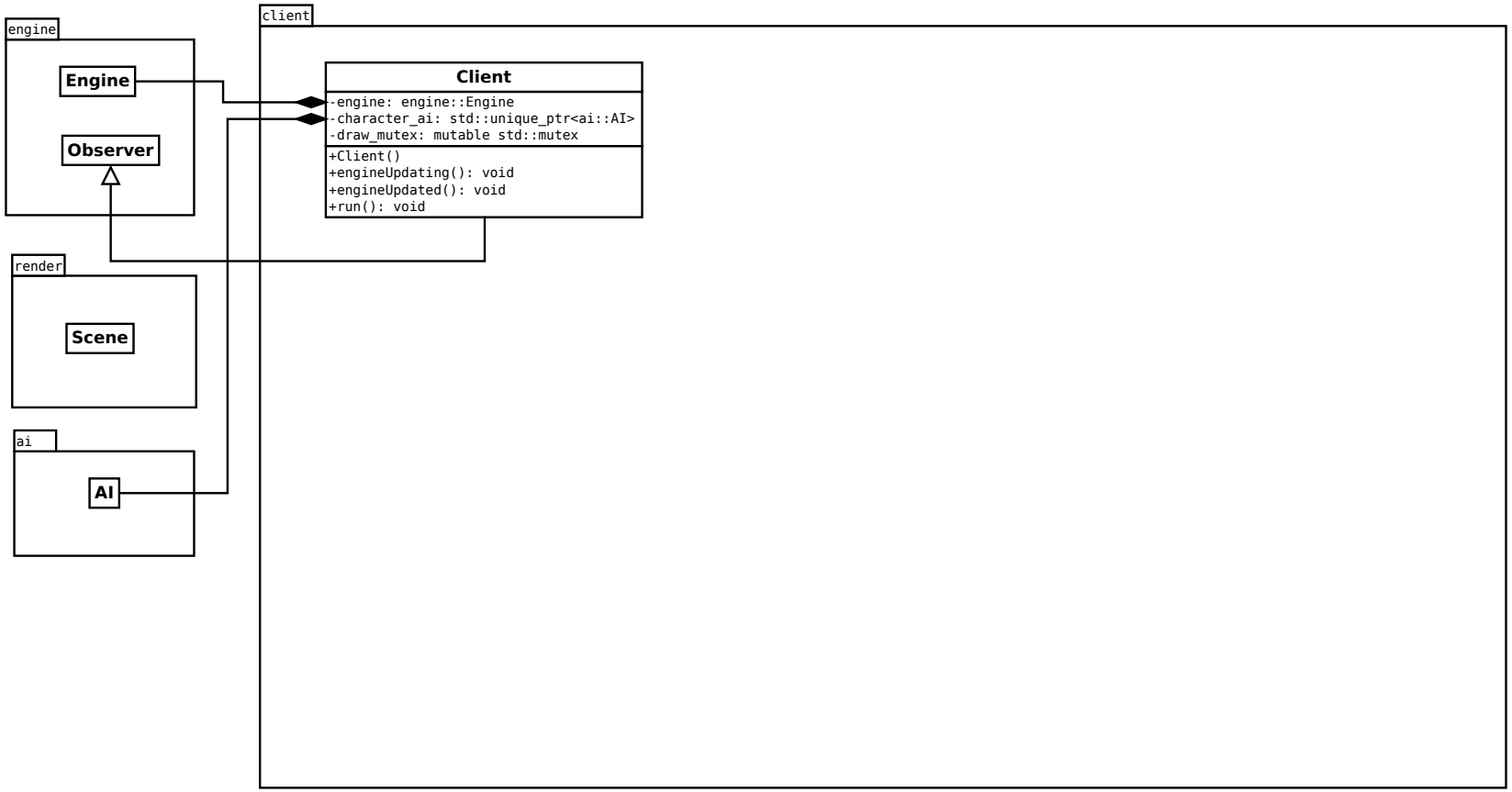
Toujours dans le fichier client.cpp, on écrit aussi tout le contenu qui se trouvait dans le main.cpp auparavant et qui sert à actualiser le rendu en continu.

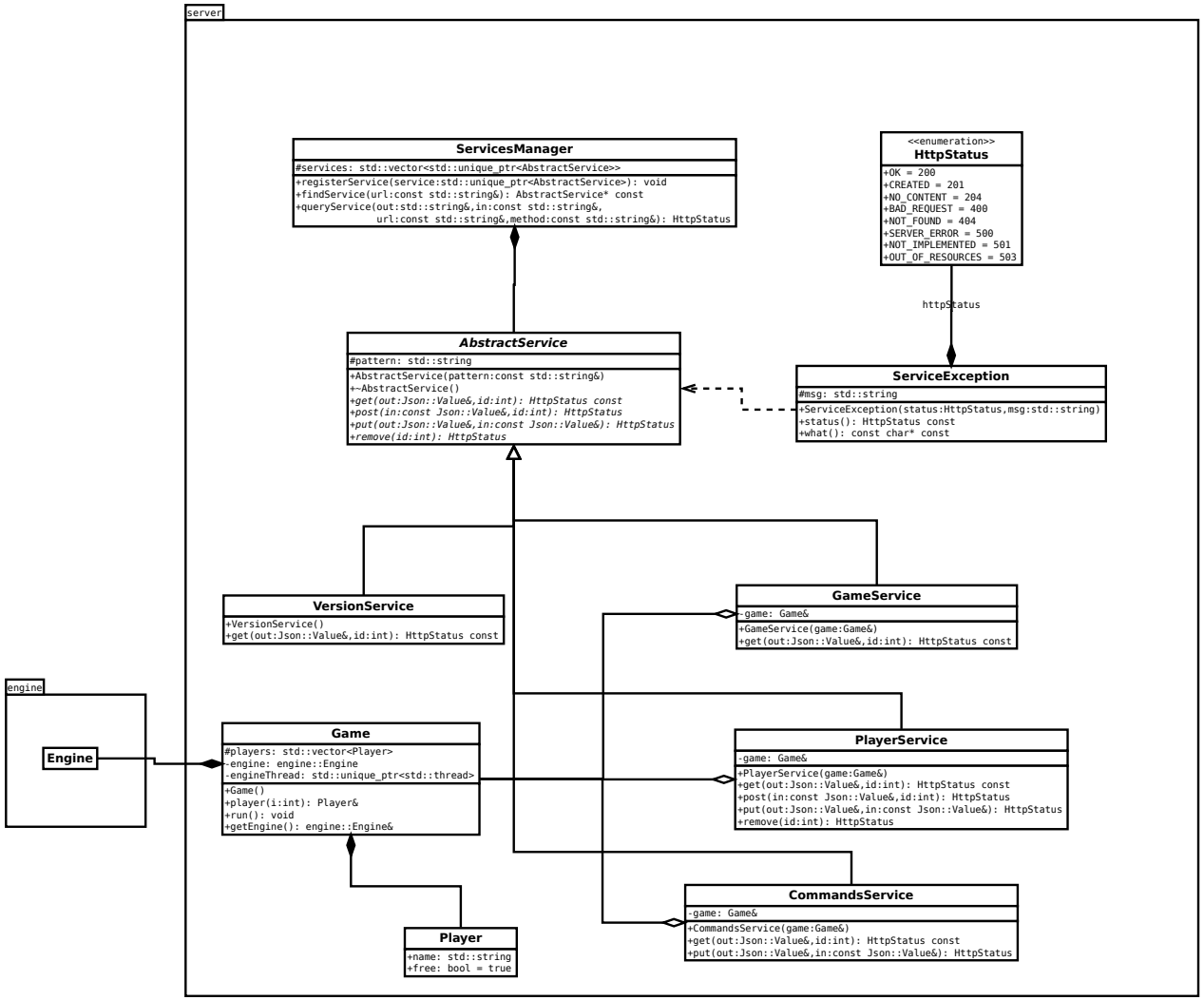
Concernant l'utilisation de Json pour nos commandes, on ajoute une méthode serialize pour chacun des types de commande qu'on peut trouver dans l'engine. Dans ces fonctions serialize, on récupère les données les plus importantes pour décrire et identifier la commande en question. On a aussi une variable notée "record" directement dans l'engine qui garde une trace de toutes les commandes qui ont été effectués afin de pouvoir, par exemple, répéter une même séquence de jeu.

Dans le client, la classe NetworkClient s'occupe de faire le lien entre le client et le serveur, le client peut, à tout moment, envoyer une commande pour qu'elle soit exécutée par le moteur du côté serveur ou recevoir des informations sur le statut du serveur/lobby (joueur(s) connecté(s)).

La classe Game se trouve côté serveur et contient la liste des joueurs connectés ainsi que le moteur du jeu. Dans un premier temps, tout se déroule dans un seul thread, le but est ensuite d'utiliser du multi-threading pour le serveur comme on l'a fait localement auparavant : un thread dédié au fonctionnement et à la mise à jour du moteur et un second thread pour le reste.

Enfin, on a différentes classes qui représentent des services disponibles sur le serveur, toutes ses classes héritent d'une classe mère notée AbstractService et sont gérés par la classe ServiceManager. Les 2 principaux services sont ceux qui permettent de gérer les joueurs connectés aux lobbys (PlayerService) et les commandes transmises par le client (CommandService). Les 2 autres servent à consulter l'état du jeu (GameService) et à renvoyer la version de l'API (VersionService) pour éviter tout problème de versions non compatibles.


client
engine
Engine
Observer
render
Scene
ai
AI
Client
-engine: engine::Engine
-character_ai: std::unique_ptr<ai::AI>
-draw_mutex: mutable std::mutex
+Client()
+engineUpdating(): void
+engineUpdated(): void
+run(): void


server
ServicesManager
#services: std::vector<std::unique_ptr<AbstractService>>
+registerService(service:std::unique_ptr<AbstractService>): void
+findService(url:const std::string&): AbstractService* const
+queryService(out:std::string&,in:const std::string&,
url:const std::string&,method:const std::string&): HttpStatus
<<enumeration>>
HttpStatus
+OK = 200
+CREATED = 201
+NO_CONTENT = 204
+BAD_REQUEST = 400
+NOT_FOUND = 404
+SERVER_ERROR = 500
+NOT_IMPLEMENTED = 501
+OUT_OF_RESOURCES = 503
httpStatus
AbstractService
#pattern: std::string
+AbstractService(pattern:const std::string&)
+~AbstractService()
+get(out:Json::Value&,id:int): HttpStatus const
+post(in:const Json::Value&,id:int): HttpStatus
+put(out:Json::Value&,in:const Json::Value&): HttpStatus
+remove(id:int): HttpStatus
ServiceException
#msg: std::string
+ServiceException(status:HttpStatus,msg:std::string)
+status(): HttpStatus const
+what(): const char* const
VersionService
+VersionService()
+get(out:Json::Value&,id:int): HttpStatus const
GameService
-game: Game&
+GameService(game:Game&)
+get(out:Json::Value&,id:int): HttpStatus const
engine
Engine
Game
#players: std::vector<Player>
-engine: engine::Engine
-engineThread: std::unique_ptr<std::thread>
+Game()
+player(i:int): Player&
+run(): void
+getEngine(): engine::Engine&
PlayerService
-game: Game&
+PlayerService(game:Game&)
+get(out:Json::Value&,id:int): HttpStatus const
+post(in:const Json::Value&,id:int): HttpStatus
+put(out:Json::Value&,in:const Json::Value&): HttpStatus
+remove(id:int): HttpStatus
CommandsService
-game: Game&
+CommandsService(game:Game&)
+get(out:Json::Value&,id:int): HttpStatus const
+put(out:Json::Value&,in:const Json::Value&): HttpStatus
Player
+name: std::string
+free: bool = true

Sources

Tiles : DungeonTiles II https://0x72.itch.io/dungeontileset-ii par 0x72
Font : Pixel Font
par Ajay Karat | Devil's Work.shop