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 dont la structure des étages est générée aléatoirement. 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 se forme de combats 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 trouver de l'équipement lui permettant d'augmenter ses statistiques (vie, armure, puissance).

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). Pour chaque élément, on comptera 2 propriétés que sont sa position et son identifiant (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 vitesse et une position. 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 ne disposent chacun que de 2 états : Alive et Dead, ils démarrent dans l'état Alive et passent à l'état Dead 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 (donc le retour au 1er étage). Comme détaillé dans la partie règle du jeu, ces éléments disposent de 3 caractéristiques sui déterminent leur puissance 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 x), 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 espaces vides 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.

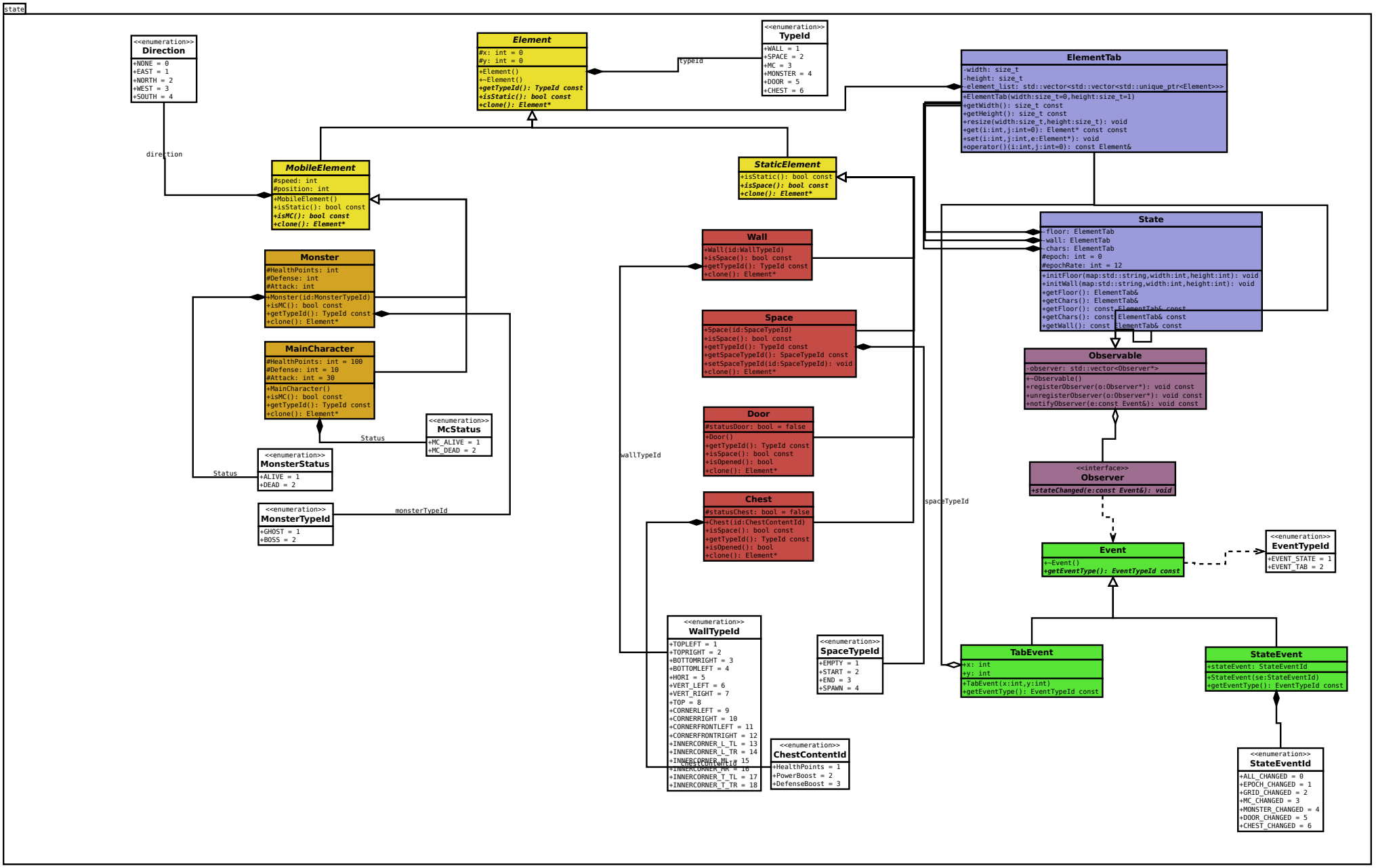
Il reste donc 3 classes :

Classe Element : Cette classe permet de définir chacun des éléments du jeu, qu'ils soient statiques ou mobiles. Ses seules caractéristiques sont des coordonnées (notés x et y) pour connaître la position de l'élément.

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.

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 immédiatement détruit.


state
<<enumeration>>
Direction
+NONE = 0
+EAST = 1
+NORTH = 2
+WEST = 3
+SOUTH = 4
direction
Element
#x: int = 0
#y: int = 0
+Element()
+~Element()
+getTypeId(): TypeId const
+isStatic(): bool const
+clone(): Element*
typeId
<<enumeration>>
TypeId
+WALL = 1
+SPACE = 2
+MC = 3
+MONSTER = 4
+DOOR = 5
+CHEST = 6
ElementTab
-width: size_t
-height: size_t
-element_list: std::vector<std::vector<std::unique_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: int
+MobileElement()
+isStatic(): bool const
+isMC(): bool const
+clone(): Element*
StaticElement
+isStatic(): bool const
+isSpace(): bool const
+clone(): Element*
State
-floor: ElementTab
-wall: ElementTab
-chars: ElementTab
#epoch: int = 0
#epochRate: int = 12
+initFloor(map:std::string,width:int,height:int): void
+initWall(map:std::string,width:int,height:int): void
+getFloor(): ElementTab&
+getChars(): ElementTab&
+getFloor(): const ElementTab& const
+getChars(): const ElementTab& const
+getWall(): const ElementTab& const
Monster
#HealthPoints: int
#Defense: int
#Attack: int
+Monster(id:MonsterTypeId)
+isMC(): bool const
+getTypeId(): TypeId const
+clone(): Element*
Wall
+Wall(id:WallTypeId)
+isSpace(): bool const
+getTypeId(): TypeId const
+clone(): Element*
Space
+Space(id:SpaceTypeId)
+isSpace(): bool const
+getTypeId(): TypeId const
+getSpaceTypeId(): SpaceTypeId const
+setSpaceTypeId(id:SpaceTypeId): void
+clone(): Element*
Observable
-observer: std::vector<Observer*>
+~Observable()
+registerObserver(o:Observer*): void const
+unregisterObserver(o:Observer*): void const
+notifyObserver(e:const Event&): void const
MainCharacter
#HealthPoints: int = 100
#Defense: int = 10
#Attack: int = 30
+MainCharacter()
+isMC(): bool const
+getTypeId(): TypeId const
+clone(): Element*
Door
#statusDoor: bool = false
+Door()
+getTypeId(): TypeId const
+isSpace(): bool const
+isOpened(): bool
+clone(): Element*
<<interface>>
Observer
+stateChanged(e:const Event&): void
<<enumeration>>
McStatus
+MC_ALIVE = 1
+MC_DEAD = 2
Status
<<enumeration>>
MonsterStatus
+ALIVE = 1
+DEAD = 2
Status
wallTypeId
Chest
#statusChest: bool = false
+Chest(id:ChestContentId)
+isSpace(): bool const
+getTypeId(): TypeId const
+isOpened(): bool
+clone(): Element*
spaceTypeId
Event
+~Event()
+getEventType(): EventTypeId const
<<enumeration>>
EventTypeId
+EVENT_STATE = 1
+EVENT_TAB = 2
<<enumeration>>
MonsterTypeId
+GHOST = 1
+BOSS = 2
monsterTypeId
<<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
<<enumeration>>
SpaceTypeId
+EMPTY = 1
+START = 2
+END = 3
+SPAWN = 4
TabEvent
+x: int
+y: int
+TabEvent(x:int,y:int)
+getEventType(): EventTypeId const
StateEvent
+stateEvent: StateEventId
+StateEvent(se:StateEventId)
+getEventType(): EventTypeId const
<<enumeration>>
ChestContentId
+HealthPoints = 1
+PowerBoost = 2
+DefenseBoost = 3
<<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. Le troisième plan nommé charsLayer contiendra les éléments mobiles (personnage contrôlé par le joueur ou les ennemis contrôlés par l'IA). Enfin, il reste un dernier plan stateLayer que nous n'utilisons pas encore mais qui servira à afficher des éléments liés à l'état du jeu à un instant T, par exemple la barre de vie du joueur, le score ...

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). 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.

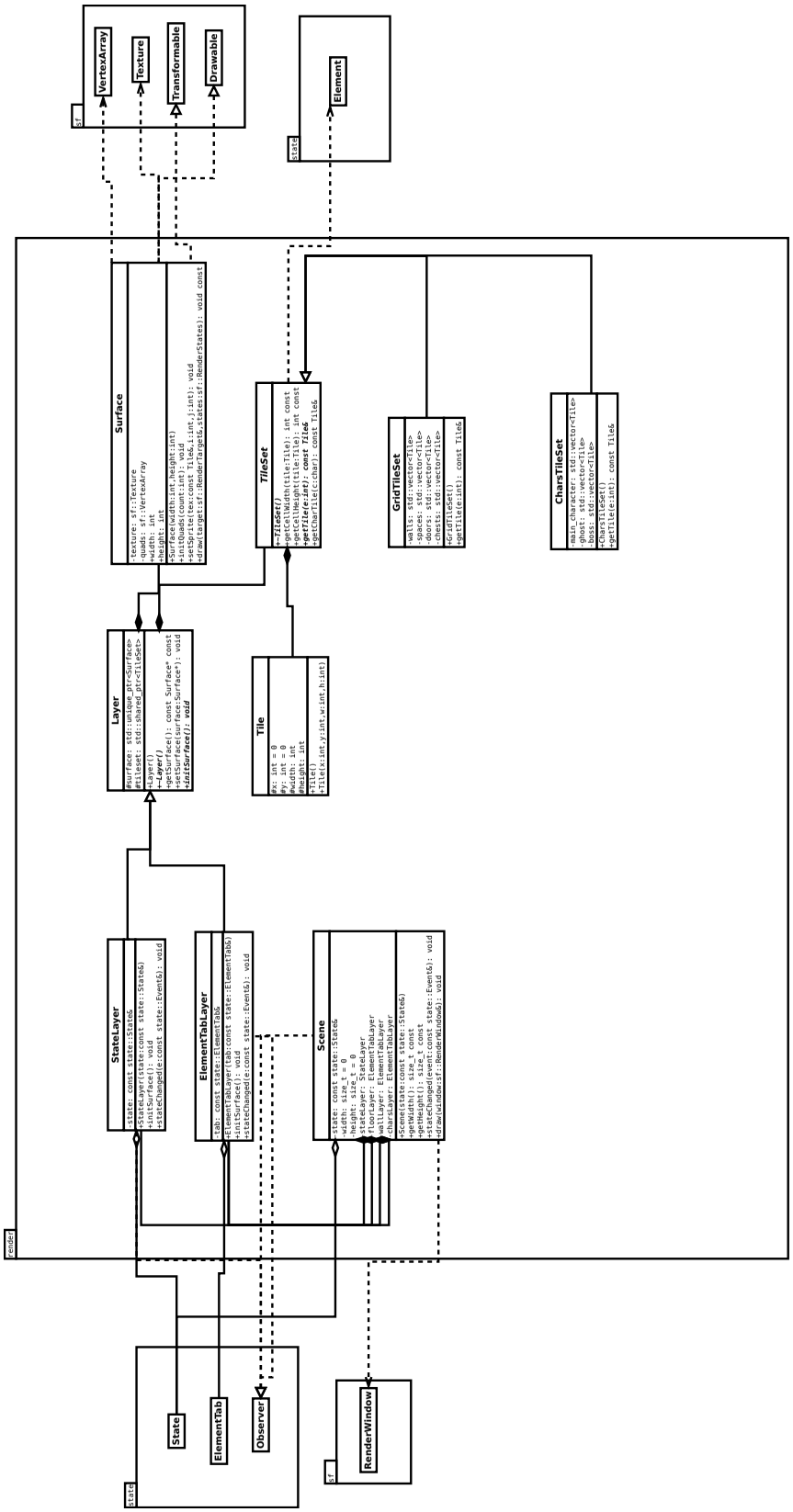
3.2 Conception logiciel

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
Layer
TileSet
Tile
GridTileSet
CharsTileSet
StateLayer
ElementTabLayer
Scene
render
State
ElementTab
Observer
RenderWindow

Sources

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