

Projet Logiciel Transversal

Nouhou KANE – Mustafa KARADAG – Alexandre LOUIS



PROFUS

Table des matières

1 Objectif.....	3
1.1 Présentation générale.....	3
1.2 Règles du jeu	3
1.3 Conception Logiciel	3
2 Description et conception des états	4
2.1 Description des états.....	4
2.2 Conception logiciel	4
2.3 Conception logiciel : extension pour le rendu.....	4
2.4 Conception logiciel : extension pour le moteur de jeu	4
2.5 Ressources	4
3 Rendu : Stratégie et Conception.....	6
3.1 Stratégie de rendu d'un état	6
3.2 Conception logiciel	6
3.3 Conception logiciel : extension pour les animations.....	6
3.4 Ressources	6
3.5 Exemple de rendu.....	6
4 Règles de changement d'états et moteur de jeu	8
4.1 Horloge globale	8
4.2 Changements extérieurs	8
4.3 Changements autonomes.....	8
4.4 Conception logiciel	8
4.5 Conception logiciel : extension pour l'IA.....	8
4.6 Conception logiciel : extension pour la parallélisation	8
5 Intelligence Artificielle	10
5.1 Stratégies	10
5.1.1 Intelligence minimale	10
5.1.2 Intelligence basée sur des heuristiques	10
5.1.3 Intelligence basée sur les arbres de recherche	10
5.2 Conception logiciel	10
5.3 Conception logiciel : extension pour l'IA composée.....	10
5.4 Conception logiciel : extension pour IA avancée.....	10
5.5 Conception logiciel : extension pour la parallélisation	10
6 Modularisation	11
6.1 Organisation des modules	11
6.1.1 Répartition sur différents threads	11
6.1.2 Répartition sur différentes machines	11
6.2 Conception logiciel	11
6.3 Conception logiciel : extension réseau	11
6.4 Conception logiciel : client Android	11

1 Objectif

1.1 Présentation générale

PROFUS est un jeu vidéo inspiré du populaire MMORPG DOFUS. Comme nous ne pouvions pas reproduire toutes les fonctionnalités de DOFUS, nous avons fait le choix de nous concentrer sur les combats présents dans le jeu. Ainsi, dans PROFUS, nous avons pour objectif de permettre à un ou plusieurs joueurs de combattre l'un contre l'autre ou ensemble face à des IA. Il s'agit de combats au tour par tour dans un univers 2D isométrique donnant l'impression d'évoluer dans un monde en 3 dimensions. Le but est d'y vaincre son ou ses adversaires en faisant tomber leurs points de vie à 0, et ainsi accumuler de l'expérience pour monter de niveau et devenir plus puissant. Pour ce faire les personnages disposent d'un ensemble d'attaques, de portées et puissances différentes dont l'utilisation est limitée par un nombre de points d'actions. Mais nous allons voir cela plus en détail avec les règles du jeu.

1.2 Règles du jeu

Splash screen :

Phase de lancement du jeu : logo du jeu au centre avec une barre de chargement en dessous par exemple.

Menu principal :

Page d'accueil au lancement du jeu. On pourrait concevoir un petit diaporama pour que le fond ne soit pas statique ou rester sur quelque chose de plus simple avec une image de fond classique. + Paramètres

Choix de l'arène et du personnage :

Au début on se contentera d'une seule map et d'un seul personnage mais si on a le temps on pourrait laisser le choix entre 4 ou 5 personnes et 2 ou 3 maps/arènes. Donc dans cette section, le joueur pourra naviguer d'une possibilité à une autre avec des flèches et passer à l'étape suivante grâce à un bouton en bas de page. Le joueur pourra avoir plus d'informations sur le personnage sélectionné (classe, stats, histoire) en cliquant sur le logo (!).

Phase de combat :

Cette phase se base intégralement sur le déroulement d'un combat sur Dofus. Au lancement du combat, le joueur a la possibilité de se placer. Il a le choix avec un certain nombre d'emplacements prédéfinis. Il peut ainsi se rapprocher le plus possible de son adversaire ou décider au contraire de s'en éloigner. Par défaut, le joueur est placé de manière aléatoire. Il a également la possibilité de changer son orientation. En effet un coup reçu de face fera moins de dégât qu'un coup reçu dans le dos.

Durant son tour, chaque joueur dispose de 60/90 secondes, durant lesquelles il pourra réaliser une série d'actions (se déplacer, attaquer, etc..), avant que ce ne soit le tour de son adversaire.

Les points importants à prendre en compte sont les suivants :

- Chaque joueur dispose d'un certain nombre de points de vie (PV) (dépendant de la classe du personnage), une fois les PV à 0, le personnage a perdu.
- Les déplacements des joueurs sont limités par leur nombre de points de mouvements (PM) qui dans Dofus s'élèvent par défaut à 3 (pas pour les mobs). En consommant des PM le joueur peut se déplacer d'une case à l'autre de la map.
- Les attaques/sorts réalisables sont limités par le nombre de points d'action (PA), par défaut les joueurs en ont 5 (pas pour les mobs)

Il faudra donc établir un certain nombre de sorts (disons 4 sorts actifs consommant des PA et 2 sorts passifs pour commencer) et pour cela nous pouvons reprendre les sorts existants sur Dofus pour ne pas trop nous prendre la tête.

Par exemple : <https://www.dofus.com/fr/mmorpg/encyclopedia/classes/8-iop>

A noter que les sorts ont des portées et zones de dégât différentes, qu'ils nécessitent plus ou moins de PA (parfois même de PM), qu'il peut y avoir une limite d'utilisation d'un sort durant un même tour voire un délai d'un certain nombre de tours entre 2 utilisations.

De plus, pour attaquer son adversaire, celui-ci ne doit pas se trouver derrière un obstacle.

En effet, un obstacle situé entre les 2 personnages, bloque la "vue" et empêche de lancer un sort. Cependant certains

sorts ne prennent pas en compte la présence d'obstacle. Il a 2 manières de mettre fin à son tour. Soit le temps accordé de 60/90 secondes est écoulé, soit le joueur y a mis fin manuellement. Il est en effet possible de terminer son tour en cliquant sur le bouton associé (dans l'interface de Dofus il s'agit d'une flèche). Le but est donc de vaincre son adversaire en faisant tomber ses PV à 0.

Tuto combat Dofus: <https://www.youtube.com/watch?v=PAsw9IOE3pg>

Ex de combats Dofus: <https://www.youtube.com/watch?v=gmiAcgAm9DQ>

1.3 Conception Logiciel

- **Archetype (DOFUS):**



- **synthetic grass :**



Source : <https://opengameart.org/content/synthetic-grass-texture-pack>

Pour le sol de notre map nous pensons utiliser une texture comme celle ci-dessus. Elle constituera la base graphique sur laquelle nous ajouterons les autres éléments.

Pour rester dans le thème original de Dofus, nous avons choisi des bâtiments médiévaux/fantasy en 3D :

- **Medieval Building 03 :**



Source : <https://opengameart.org/users/bleed>

- **Medieval-Tavern :**



Source : <https://opengameart.org/users/bleed>

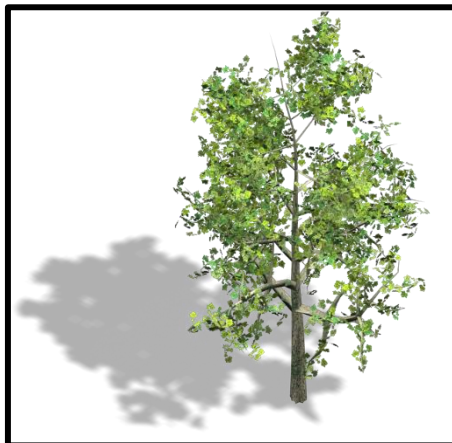
- **Timbered House_16 :**



Source : <https://opengameart.org/users/bleed>

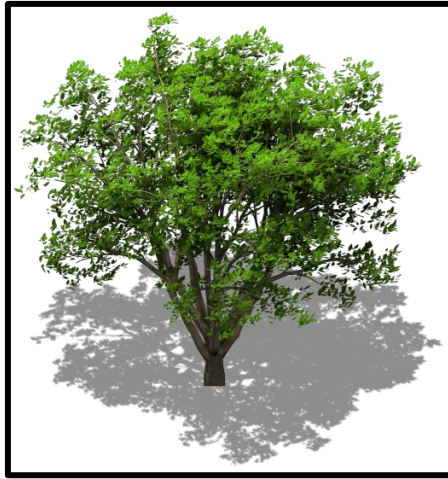
Nous avons sélectionné un peu de verdure avec différents types d'arbres :

- **_01 :**



Source : <https://opengameart.org/users/bleed>

- **Animated Swietenia :**



Source : <https://opengameart.org/users/bleed>

Et des éléments plus décoratifs mais qui serviront eux aussi d'obstacles comme :

- **Well:**



Source : <https://opengameart.org/users/bleed>

Concernant les personnages, pour le moment nous sommes partis sur 2 possibilités :

Projet Logiciel Transversal – Nouhou KANE – Mustafa KARADAG – Alexandre LOUIS

- Valla :



Source : <https://www.gamedevmarket.net/member/badim/>

- demon :



Source : <https://www.gamedevmarket.net/member/badim/>

Le joueur ayant la possibilité de changer l'orientation de son personnage, nous avons recherché des personnages ayant des sprites dans les 4 directions.

2 Description et conception des états

2.1 Description des états

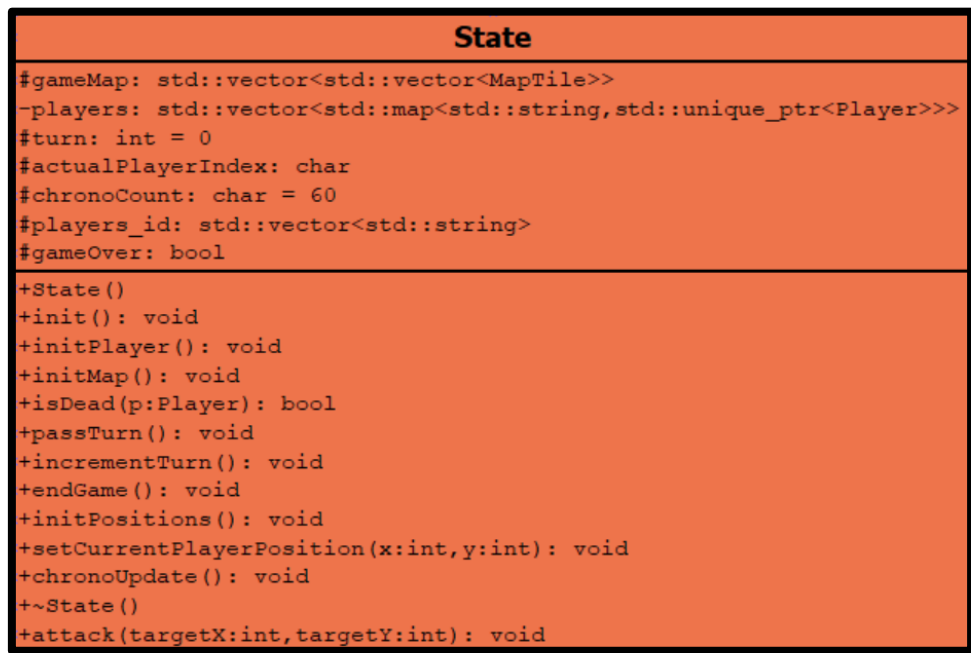
Notre projet comporte plusieurs classes. Chaque classe décrit un état ou un objet représentatif précis. L'état du jeu à un instant t est géré par une classe que l'on appelle State. Cette classe enregistre des informations telles que les joueurs de la partie (Héros, ennemis), le nombre de tours de jeu (le nombre de fois qu'on est passé d'un joueur à un autre), les identifiants de tous les joueurs et celui du joueur actuel, le temps de jeu passé et l'état de la map du jeux etc.

Chaque joueur est un objet de la classe Player identifié par un nom unique (attribué à l'exécution selon le choix du joueur {Héros} ou de manière automatique {ennemi}). Chaque joueur possède ses statistiques (points de vie, points de mouvement, niveau etc.), ses attaques (type et puissance d'attaque etc.) son orientation, sa position, son statut (mort, en train de jouer etc.).

2.2 Conception logiciel

Le diagramme d'état est composé de 6 classes, 2 structures ,8 énumérations et une interface. **State** et **Player** sont les 2 classes principales.

State



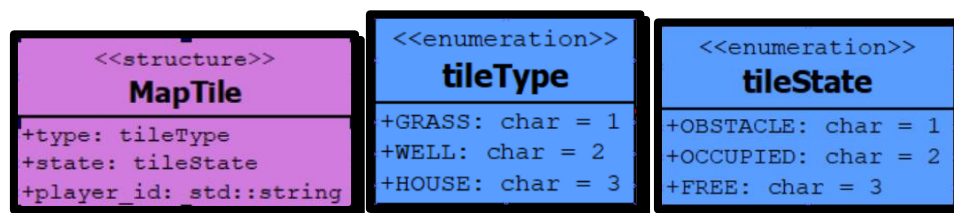
State est la classe principale du diagramme d'État. Elle permet de définir l'état dans lequel le jeu se trouve à tout instant et permet d'initialiser ces derniers. Dans cette classe, on trouve les vecteur *heroes* et *enemies* dans lesquels on liste les types de héros et ennemis.

gameMap est un vecteur de vecteur de **MapTile** et va permettre de définir chaque "tiles" ou "cases" de jeu pour obtenir une cartographie entière de notre zone de jeu. (vecteur de vecteur = les cases doivent être vues comme constituant un plateau, comparable à un échiquier, où l'on peut identifier une case par un couple (x,y))

La structure **MapTile** associée à **State** va permettre de décrire chacune de ces cases à l'aide de ses attributs *type* et *state* :

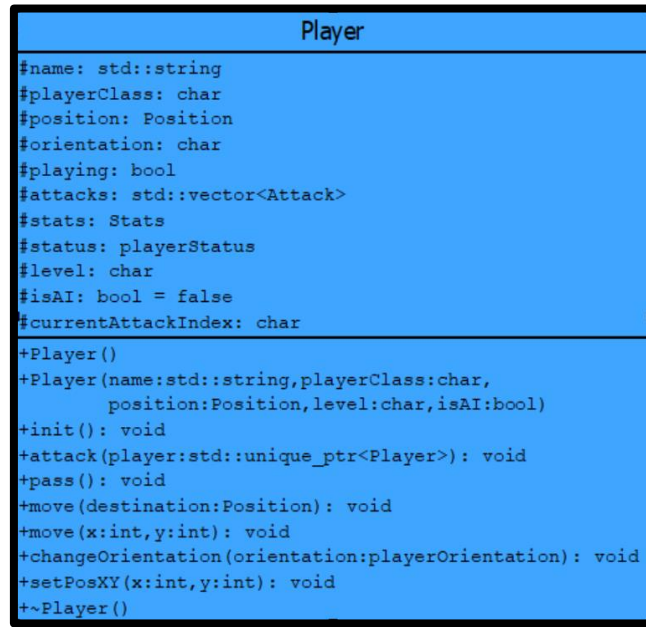
-type = choix de la forme de la case (de l'herbe, un puit, une maison) dans **tileType**

-state = disponibilité de la case (libre, occupée ou obstacle) dans **tileState**



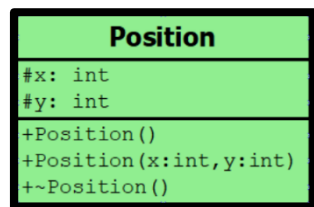
turn indique le tour de jeu, *player* indique l'entité qui doit jouer son tour, *chronoCount* définit le temps de jeu de chacun à chaque tour. Quant à *playerId*, il s'agit d'un vecteur de chaîne de caractères qui contient tous les identifiants des joueurs (*heroes* et *enemies*) et nous permettra de faire se succéder les différents tours simplement. Enfin le booléen *gameOver* permet d'indiquer si le jeu est fini ou non. Les différentes méthodes permettent d'initialiser le jeu, la carte, les joueurs, la position des joueurs, de gérer les rotations de tour de jeu et la fin du jeu.

Player

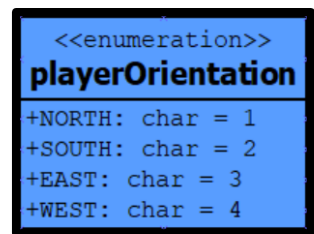


La classe **Player**, comme son nom l'indique, va gérer tout ce qui concerne un joueur :

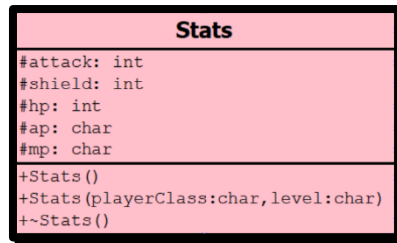
-sa position : Définie par la classe **Position** qui est une partie de **Player**, modifiable à l'aide de la méthode **move**. On utilise la méthode **move** quand un joueur souhaite se déplacer d'une case à l'autre par exemple.



-son orientation : Le personnage est tourné vers le nord, le sud, l'est ou l'ouest. On les retrouve dans l'énumération associée **playerOrientation**.

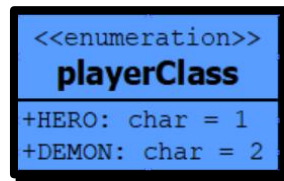


-sa *class* : Le joueur a le choix entre jouer un *Hero* ou un *Demon*. On les retrouve dans l'énumération associée **playerClass**.

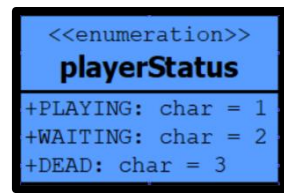


-son *level* : Le niveau du joueur correspond à son avancement dans le jeu. Cela affecte ses stats et les dégâts qu'il peut infliger avec ses attaques.

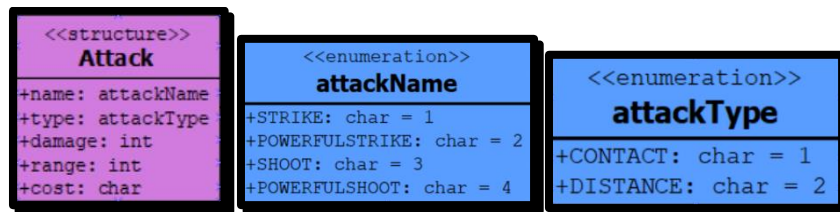
-ses stats (caractéristiques liées au combat): Définies par la classe **Stats**. Le joueur dispose de *mp* (movement point) consommables pour se déplacer ; de *ap* (action point) consommables pour effectuer une action comme une attaque ; de *hp* (health point = points de vie) représentant son niveau santé, s'ils tombent à 0 le joueur est vaincu ; de *shield* (bouclier) qui réduit les dégâts subis ; d'*attack* qui augmente les dégâts infligés par ses attaques.



-son status en combat : Indique si le joueur joue, s'il est en attente de son tour de jeu ou s'il est mort. On les retrouve dans l'énumération associée **playerStatus**.



-ses *attacks* en combat : La méthode attack prend en argument l'attaque choisie par le joueur. Les différentes attaques sont listées grâce à la structure associée **attack**. Chaque attaque est définie par son coût (le nombre de points d'action nécessaires pour l'utiliser), ses dégâts (le nombre de point de vie retiré à l'adversaire), sa portée (la distance maximale à laquelle le joueur peut lancer son attaque), son type (différents types dans l'énumération associée **attackType**) et son *name* (nom d'attaque) dans l'énumération associée **attackName**.



- Enfin, la variable *isAi* va nous permettre de faire la distinction entre les joueurs réels (*isAi = false*) et les intelligences artificielles (*isAi = true*).

Déroulé de jeu

La map est générée, les joueurs sont initialisés, chaque joueur a sa classe, l'ordre de jeu est défini et le combat peut commencer. Tuer la totalité des adversaires permet de remporter le combat. On meurt lorsque nos points de vie (*hp*) atteignent 0. Les joueurs se déplacent et effectuent des attaques avec les *ap* et *mp* qui leur sont associés. Ils devront faire preuve de tactique pour infliger plus de dégâts que l'adversaire.

Les joueurs peuvent se déplacer sur toutes les cases libres, et non sur les obstacles et les cases occupées par une autre entité.

Les joueurs peuvent attaquer sur toutes les cases libres et les cases occupées et non sur les obstacles. Les obstacles

obstruent la vue donc un joueur ne peut pas forcément attaquer une entité se trouvant derrière.
La fin de combat octroie des récompenses (augmentation de niveau du personnage)

2.3 Conception logiciel : extension pour le rendu

Rendu d'un état

Nous avons conçu une map de 20 tiles sur 20 tiles, avec plusieurs dizaines d'obstacles. L'espace de combat n'est donc pas trop vaste et permet la mise en place de stratégie (se protéger d'une attaque en bloquant la ligne de vue de l'adversaire grâce à un mur par exemple). Nous avons souhaité nous rapprocher le plus possible de l'atmosphère de Dofus, par conséquent au-delà des textures médiévales, nous avons opté pour une map avec une orientation isométrique pour un effet de jeu en semi-3D. Toutes les données de notre map sont stockées sous forme d'un fichier tmx. Nous l'avons imaginée à l'aide du logiciel Tiled avant de l'exporter. Grâce au render SFML nous pouvons ensuite charger notre map mais également la modifier au fur à mesure de la partie. Pour chaque état nous avons 3 éléments à prendre en compte et à afficher :

- La map en temps que plateau de tuiles, surface statique sur laquelle les personnages se déplacent.
- Les personnages en animation constante grâce à la succession rapide de sprites et qui ont également la possibilité de changer d'emplacement sur la map. Nous envisageons également une animation d'attaque si possible.
- Et enfin les informations de jeu tels que les points de vie, d'actions et de mouvements ou le temps restant au joueur, mais aussi les statistiques des personnages ou les options de jeu.

Lorsque l'état de jeu est modifié dans le cas du déplacement d'un personnage par exemple nous allons venir modifier notre variable `state::map` et notre render va actualiser l'affichage de la map en conséquence.



Exemple d'état de jeu

Nous avons une nouvelle fois réalisé un diagramme à l'aide du logiciel Dia pour décrire l'architecture de notre render. Le diagramme se compose de 9 classes : **GameWindow**, **Scene**, **Box**, **BoxFactory**, **Info**, **Button**, **FightScene**, **Background** et **GameFocus** et 1 énumération : **buttonState**.

GameWindow

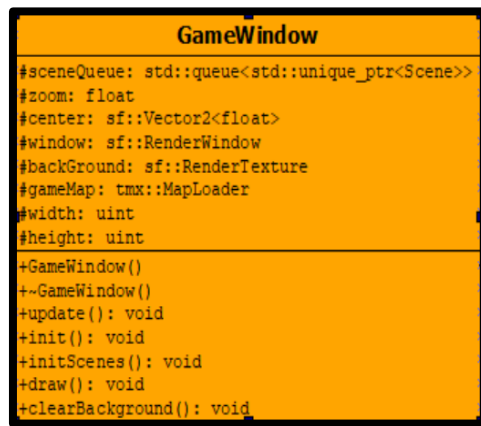
Cette classe est celle destinée à générer et gérer la fenêtre de jeu.

Elle possède 8 attributs :

- *window* de type `sf::RenderWindow` va permettre de générer à la fenêtre globale dans laquelle sera contenue tous les autres éléments.
- *zoom* de type `float` est une valeur réelle permettant d'ajuster l'agrandissement du contenu de la fenêtre.
- *center* de type `sf::Vector2f` est un couple de coordonnées indiquant la position qui sera prise comme centre de la fenêtre.
- *backGround* de type `sf::RenderTexture` qui correspond au background de notre map, c'est-à-dire la partie statique de notre map sur lequel évolueront les personnages.
- *gameMap* de type `tmx::MapLoader` sera chargée dans la variable background et contient comme son nom l'indique notre map conçue sur Tiled.
- *width* de type `uint` qui correspond à la largeur de la fenêtre.
- *height* de type `uint` qui correspond à la hauteur de la fenêtre.
- Enfin, *sceneQueue* regroupe les pointeurs sur de 3 objets **Scène**, qui sont les 3 pages qui susceptibles d'être affichées après le lancement du jeu : le menu/page d'accueil, la page de combat, la page de victoire ou de défaite.

En plus de son constructeur et son destructeur, elle possède 5 méthodes :

- `init()` pour initialise l'attribut *window* et donc la fenêtre qui sera affichée.
- `initScenes()` pour créer et initialiser les 3 scènes requises pour le menu, le combat et la fin de partie.
- `draw()` pour afficher la fenêtre avec les paramètres désirés ainsi que la scène actuelle.
- `clearBackground()` pour effacer le contenu de la fenêtre.
- Et `update()` permettant d'actualiser le contenu de la fenêtre.



```
GameWindow
#sceneQueue: std::queue<std::unique_ptr<Scene>>
#zoom: float
#center: sf::Vector2<float>
#window: sf::RenderWindow
#backGround: sf::RenderTexture
#gameMap: tmx::MapLoader
#width: uint
#height: uint
+GameWindow()
+~GameWindow()
+update(): void
+init(): void
+initScenes(): void
+draw(): void
+clearBackground(): void
```

Scene

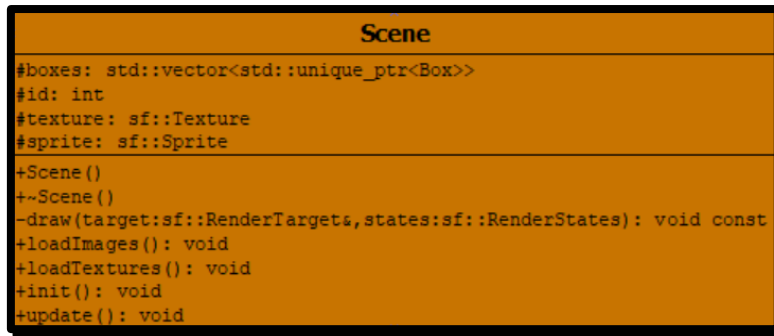
Cette classe est celle destinée à générer et gérer les scènes, c'est-à-dire les différentes pages du jeu.

Elle possède 3 attributs :

- *boxes* qui est un ensemble de pointeurs sur des objets **Box** qui sont les formes et boutons ou simplement les textes qui seront affichés pour une scène donnée.
- *id* de type `int` est une valeur unique permettant de distinguer les différentes scènes.
- *t* de type `sf::Texture` qui correspond à la texture utiliser pour une Scene donnée.

En plus de son constructeur et son destructeur, elle possède 5 méthodes :

- `init()` pour initialiser la scène.
- `draw()` pour dessiner et afficher à l'écran les éléments de la scène.
- `loadImages()` pour charger les images qui vont être affichées.
- `loadTextures()` pour charger les texture des éléments à afficher.
- `update()` pour mettre à jour l'affichage de la scène.



Box

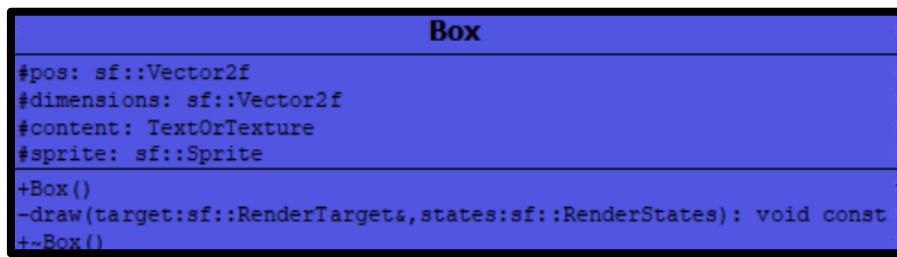
Cette classe est celle destinée à générer et gérer les boutons, formes, informations qui seront affichés sur les différentes scènes.

Elle possède 4 attributs :

- *pos* de type `sf::Vector2f` qui contient les coordonnées de l'objet **Box** afin de pouvoir le dessiner à l'endroit souhaité via la méthode `draw()`.
- *dimensions* également de type `sf::Vector` est un couple qui contiendra la longueur et la largeur de la Box à afficher.
- *content* de type `TextOrTexture` qui correspond à la texture ou le texte utilisé par la Box.
- *sprite*, de type `sf::Sprite` qui correspond à la sprite qui va contenir la Box.

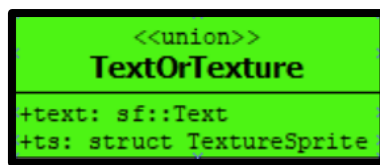
En plus de son constructeur et son destructeur, elle possède 1 méthode :

- `draw()` pour afficher la Box sur la scène à la position souhaitée.



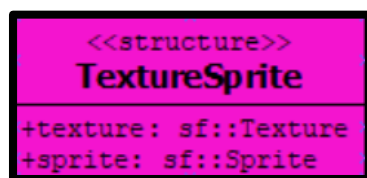
TextOrTexture

TextOrTexture est une union qui permet de choisir entre une texture et un texte pour le contenu d'une Box donnée. Cela permet de ne pas avoir deux données en mémoire.



TextureSprite

Il s'agit d'une structure ayant 2 attributs *texture* et *sprite*, on a ainsi une texture et la sprite qui lui est associée.

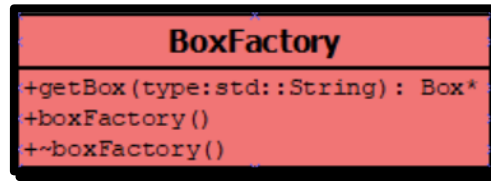


BoxFactory

Nous avons ajouté cette classe particulière dans l'optique d'utiliser le pattern Factory, il s'agit d'un « distributeur » permettant de créer un bouton (**Button**) ou une zone de texte (**Info**) en fonction du type d'objet que l'on cherche à générer. Elle ne possède pas d'attribut son rôle étant uniquement d'instancier des objets.

En plus de son constructeur et son destructeur, elle possède 1 méthode :

- `getBox(type:std::String)` qui prend une chaîne de caractère « Button » ou « Info » et générera l'objet correspondant.



Button

Cette classe est celle destinée à générer et gérer les scènes, c'est-à-dire les différentes pages du jeu.

Elle possède 17 attributs :

- `m_btnState` de type `buttonState` indiquant l'état du bouton (cliqué, survolé, normal).
- `m_text` de type `sf::Text` contenant le texte liée au bouton.
- `m_bgNormal` de type `sf::Color` qui indique la couleur du background quand le bouton est dans l'état normal.
- `m_bgHover` de type `sf::Color` qui indique la couleur du background quand le bouton est dans l'état survolé.
- `m_bgClicked` de type `sf::Color` qui indique la couleur du background quand le bouton est dans l'état cliqué.
- `m_textNormal` de type `sf::Color` qui indique la couleur du text quand le bouton est dans l'état normal.
- `m_textClicked` de type `sf::Color` qui indique la couleur du text quand le bouton est dans l'état cliqué.
- `m_textHover` de type `sf::Color` qui indique la couleur du text quand le bouton est dans l'état survolé.
- `m_border` de type `sf::Color` qui indique la couleur de la bordure/ le contour du bouton.
- `m_borderThickness` de type `float` qui quantifie l'épaisseur de la bordure.
- `m_borderRadius` de type `float` qui permet de quantifier le degré d'arrondissement des coins du bouton.
- `m_size` de type `sf::Vector2f` qui correspond au couple longueur/largeur caractérisant les dimensions du bouton.
- `m_style` de type `buttonStyle` qui indique le type de style à appliquer au bouton.
- `m_button` de type `sf::ConvexShape` qui définit la forme du bouton.
- `m_fontSize` de type `unsigned int` qui indique la taille de la police de texte.
- `m_font` de type `sf::Font` qui indique la police de texte à utiliser.
- Enfin, `m_shadow` de type `sf::Text` qui représente l'ombre du texte.

En plus de son constructeur et son destructeur, elle possède 1 constructeur à paramètres ainsi que 2 méthodes :

- `draw()` pour dessiner le bouton.
- `update()` pour réactualiser le bouton.

Button
<pre>#m_btnState: enum buttonState #m_text: sf::Text #m_bgNormal: sf::Color #m_bgHover: sf::Color #m_bgClicked: sf::Color #m_textNormal: sf::Color #m_textHover: sf::Color #m_textClicked: sf::Color #m_border: sf::Color #m_borderThickness: float #m_borderRadius: float #m_size: sf::Vector2f #m_style: enum buttonStyle #m_button: sf::ConvexShape #m_fontSize: unsigned int #m_font: sf::Font #m_shadow: sf::Text</pre>
<pre>+Button() +Button(s:std::string,font:sf::Font,position:sf::Vector2f, style:enum buttonStyle) +~Button() -draw(target:sf::RenderTarget,states:sf::RenderStates): void const +update(e:sf::Event,m_mousePosition:sf::Vector2i): void</pre>

buttonState

Il s'agit d'une énumération qui contient 3 valeurs de type char *CLICKED* (cliqué), *HOVERED* (survolé) et *NORMAL* (normal) représentant les différents états que peut prendre un bouton.

<<enumeration>> buttonState
+CLICKED: char = 0
+HOVERED: char = 1
+NORMAL: char = 2

buttonStyle

Il s'agit d'une énumération qui contient 4 valeurs de type char *NONE* (aucun), *SAVE* (sauvegarder), *CANCEL* (annuler) et *CLEAN* (nettoyer) représentant les différents styles que peut prendre un bouton.

<<enumeration>> buttonStyle
+NONE: char = 0
+SAVE: char = 1
+CANCEL: char = 2
+CLEAN: char = 3

Info

Cette classe est celle destinée à générer et gérer les scènes, c'est-à-dire les différentes pages du jeu.

Elle possède 1 attributs :

- *font* de type int est une valeur unique permettant de distinguer les différentes scènes.

En plus de son constructeur et son destructeur, elle possède 1 méthode, il s'agit de *draw()* pour dessiner les textes à l'écran.

Info
<pre>#font: sf::Font</pre>
<pre>+Info() -draw(target:sf::RenderTarget,states:sf::RenderStates): void const +~Info()</pre>

FightScene

Cette classe héritant de la classe **Scene** est celle destinée à générer et gérer la scène de combat, elle s'occupera en autres de l'affichage de la map et des personnages.

Elle possède 3 attributs :

- *state* de type `state::State*` qui contiendra l'état du jeu afin d'actualiser l'affichage au fur et à mesure du combat.
- *animatedObjectss* de type `std::vector<std::unique_ptr<AnimatedObject>>` qui est la liste des objets animés de la scène, c'est-à-dire les personnages.
- *framesInfo* de type `Json::Value` est fichier dans lequel sont inscrites les positions et les tailles des différentes frames des animations.

Elle possède deux méthodes en plus de son constructeur et son destructeur.

- `draw()`, pour dessiner la scène dans le fenêtre.
- `update()`, qui recharge la scène de combat.

```

FightScene
#state: state::State*
#animatedObjects: std::vector<std::unique_ptr<AnimatedObject>>
#frameInfos: Json::Value
+FightScene()
+~FightScene()
+update(): void
+draw(target:sf::RenderTarget&,states:sf::RenderStates): void const

```

AnimatedObject

Cette classe est celle destinée à générer et gérer les objets animés.

Elle possède 4 attributs :

- *mSprite*, de type `sf::Sprite` qui correspond à la sprite de l'objet qui sera animé pour une frame donnée.
- *mCurrentFrame*, de type `sf::size_t` à la frame qu'il faut afficher et passer à *mSprite*.
- *mElapsedTime* de type `sf::Time` permet de savoir le temps qui a passé depuis le début de l'animation.
- *mRepeat* est un booléen qui nous indique si l'animation de l'objet doit être répétée ou non.

Elle possède deux méthodes en plus de son constructeur et son destructeur.

- `draw()`, pour dessiner l'animation (afficher successivement les différents frames).
- `update()`, qui permet de mettre à jour l'objet animé, en actualisant la frame à afficher et le temps écoulé.

```

AnimatedObject
#mSprite: sf::Sprite
#mCurrentFrame: std::size_t
#mElapsedTime: sf::Time
#mRepeat: bool
+AnimatedObject(frames:sf::Texture &)
+~AnimatedObject()
+draw(target:sf::RenderTarget&,states:sf::RenderStates&): void const
+update(dt:sf::Time,framesInfos:Json::Value,
        frameKey:std::string,positions:sf::Vector2f): void

```

Déroulé de jeu

La map est générée, les joueurs sont initialisés, chaque joueur a sa classe, l'ordre de jeu est défini et le combat peut commencer. Tuer la totalité des adversaires permet de remporter le combat. On meurt lorsque nos points de vie (*hp*) atteignent 0. Les joueurs se déplacent et effectuent des attaques avec les *ap* et *mp* qui leurs sont associés. Ils devront faire preuve de tactique pour infliger plus de dégâts que l'adversaire.

Les joueurs peuvent se déplacer sur toutes les cases libres, et non sur les obstacles et les cases occupées par une autre entité.

Les joueurs peuvent attaquer sur toutes les cases libres et les cases occupées et non sur les obstacles. Les obstacles obstruent la vue donc un joueur ne peut pas forcément attaquer une entité se trouvant derrière.

La fin de combat octroie des récompenses (augmentation de niveau du personnage)

2.4 Conception logiciel : extension pour le moteur de jeu

Règles de changement d'états.

Plusieurs évènements peuvent provoquer le passage d'un état à l'autre, nous les avons listés ci-dessous :

- **Déplacement :**

Lors de son tour, comme nous l'avons vu, le joueur peut se déplacer d'une case à un autre en consommant des points de mouvements. Il ne peut pas se déplacer de plus de cases qu'il n'a de points de mouvements. Pour se déplacer, il peut ainsi cliquer sur les cases entourant son personnage dans un périmètre de maximum *nb_de_points_de_mouvement* cases. Lors d'un clic sur une case, nous récupérons alors la position du clic grâce aux fonctions disponibles dans la bibliothèque SFML. On vérifie que le joueur peut effectivement se déplacer à cette case et si c'est le cas on va générer une commande (**move**). Cette commande *move* va modifier la position du personnage concerné en agissant sur le state (*state::Player::position*). Il est donc essentiel que l'engine ait en permanence un accès au state afin de le mettre à jour. Enfin le render réactualisant sans cesse l'affichage, on observe le personnage se déplacer à l'écran.

- **Attaque :**

Durant son tour, le joueur peut également attaquer les ennemis auxquels il fait face en échange de points d'action. Il peut lancer un sort/une attaque uniquement si le coût du sort est inférieur au nombre de points d'action restants au joueur. Pour attaquer, il peut ainsi réaliser un premier clic pour choisir son attaque parmi les différents sorts, l'engine va alors modifier la variable *currentAttackIndex* du personnage attaquant dans le state, afin d'enregistrer le choix de l'attaque. Puis dans un second temps, il faudra réaliser un second clic, sur la case où l'on souhaite jeter le sort. Comme pour les déplacements, le sort ne peut être lancé que sur les cases entourant son personnage dans un périmètre de maximum *valeur_portée_attaque* cases. Lors d'un clic sur une case, nous récupérons alors la position du clic grâce aux fonctions disponibles dans la bibliothèque SFML. On regarde l'attribut *state* de la case et on agit de la manière suivante :

- S'il s'agit d'un obstacle ou que la case est vide, alors l'attaque est exécutée mais la seule conséquence est l'actualisation des points d'action du joueur. La commande **attack** est ainsi générée et va modifier dans le state le nombre de points d'actions du personnage concerné (*state::Player::stats.ap*).
- Si la case est occupée par un autre joueur alors dans ce cas en plus d'actualiser les points d'action du joueur attaquant, il faut mettre à jour les points de vie du joueur attaqué. Pour se faire, on va lire l'attribut *player_id* de la case, accéder aux joueurs correspondant dans notre liste de joueurs, et diminuer ses hp en tenant compte des stats offensives et défensives des deux joueurs.

- **Changement de tour :**

Enfin, un changement d'état à évidemment lieu quand le joueur passe son tour (ou que les 60 secondes qui lui sont accordées sont écoulées). Si le joueur clique sur le bouton « Passer mon tour », on va générer la commande **passTurn**. Cette commande *passTurn* va modifier la variable *Player::playerStatus* à *WAITING* et *playing* passe à *false*. Ensuite, *State::actualPlayerIndex* est mis à jour pour donner la main au joueur suivant pour qui *Player::playerStatus* passera à *PLAYING* et *playing* passe à *true*.

De plus, lorsque les points de vie de l'un des joueurs tombent à 0, un changement d'état à également lieu. En effet, les points de vie des différents personnages étant surveillés continuellement, quand ils chutent à 0, la variable *Player::playerStatus* passe à *DEAD* et dans ce cas, la fonction *State::isDead()* retournera *true*, *State::gameOver* passera à *true* et le combat se terminera.

Engine

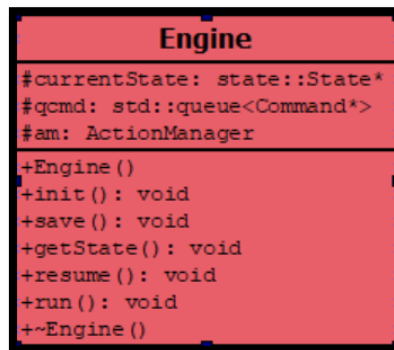
Cette classe est celle destinée à représenter le moteur de jeu et contenir ses fonctionnalités.

Elle possède 3 attributs :

- *currentState*, qui est un pointeur sur l'état actuel du jeu du package State.
- *qcmd* est une queue de commandes qui va contenir les commandes à exécuter. Elle se remplit et se vide continuellement.
- *am* de type ActionManager qui comme son nom l'indique va permettre de gérer/écouter les différentes actions de l'utilisateur et générer des commandes en conséquence.

Elle possède cinq méthodes en plus de son constructeur et son destructeur.

- *init()*, pour correctement initialiser l'engine au lancement du jeu.
- *save()*, pour sauvegarder l'avancement du jeu (tour actuel, stats des joueurs, etc..)
- *getState()*, pour mettre à jour *currentState* et c'est-à-dire actualiser l'état de jeu dans l'engine.
- *resume()*, pour charger et reprendre une partie qui a été sauvegardée.
- Et enfin *run()*, qui va mettre en route le moteur de jeu (écoute des actions des joueurs + mise à jour du state + exécution des IA).



ActionManager

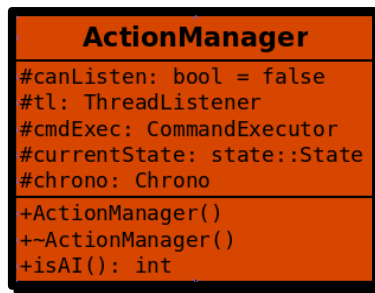
Cette classe va permettre de surveiller les actions des joueurs durant leur tour et générer les commandes adéquates.

Elle possède 5 attributs :

- *canListen*, qui permet d'indiquer si l'on doit écouter les actions réalisés par le joueur ou non. Quand les 60 secondes allouées à son tour sont écoulées, on ne doit plus écouter les saisies réalisées par le joueur.
- *tl* est un *threadListener* qui sera justement chargé d'écouter les actions et saisies du joueur, s'il y est autorisé.
- *cmdExec* qui exécute les commandes contenues dans la queue.
- *currentState* *currentState*, qui est un pointeur sur l'état actuel du jeu du package State.
- *chrono* qui comme son nom l'indique va mesurer le temps écoulé et alerter l'engine quand un certain temps sera passé. Cela va nous permettre de plafonner le temps accordé à chaque joueur à 60 secondes comme souhaité.

Elle possède une méthode en plus de son constructeur et son destructeur.

- *isAi()*, pour correctement initialiser l'engine au lancement du jeu.



Command

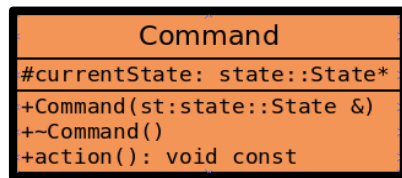
Cette classe est celle destinée à représenter une commande, transcription d'une action réalisée par un joueur.

Elle possède qu'un seul attribut :

- *currentState*, qui est un pointeur sur l'état actuel du jeu du package State. C'est sur lui que la classe agira pour modifier l'état en fonction des saisies du joueur.

Elle possède une unique méthode en plus de son constructeur et son destructeur.

- *action()*, qui en fonction du type de commande va modifier l'état de la manière souhaitée (dans le cas d'une commande **move** modifier la position du joueur, etc..)

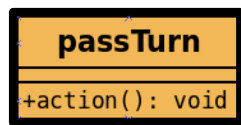


passTurn

Cette classe représente la commande générée suite à un clic du joueur sur le bouton « Passer mon tour »

Elle ne possède qu'une méthode :

- *action()*, qui redéfinit la méthode *action()* de la classe mère, de sorte à réaliser dans le state, les modifications impliquées par un changement de tour.



Move

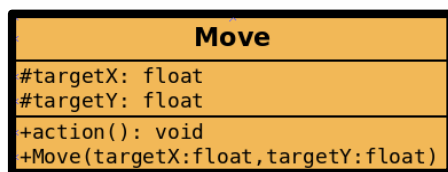
Cette classe représente la commande générée lorsque le joueur souhaite se déplacer sur la map.

Elle possède deux attributs :

- *targetX* et *targetY* contenant simplement les coordonnées de la souris lors du clic sur la map.

Elle possède une méthode en plus de son constructeur :

- *action()*, qui redéfinit la méthode *action()* de la classe mère, de sorte à réaliser dans le state, les modifications impliquées par un déplacement du joueur.



Attack

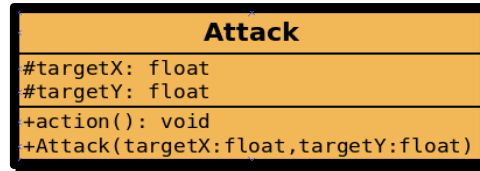
Cette classe représente la commande générée lorsque le joueur souhaite attaquer.

Elle possède deux attributs :

- *targetX* et *targetY* contenant simplement les coordonnées de la souris lors du clic sur la map.

Elle possède une méthode en plus de son constructeur :

- *action()*, qui redéfinit la méthode *action()* de la classe mère, de sorte à réaliser dans le state, les modifications impliquées par l'attaque du joueur.



ThreadListener

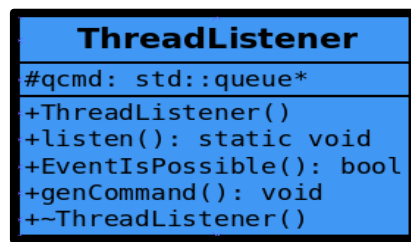
Cette classe permet de surveiller les actions du joueur, écouter les saisies, vérifier leur validité, générer les commandes correspondantes et les ajouter à la queue de commandes.

Elle possède un attribut :

- *qcmd* un pointeur sur la queue de commande du engine.

Elle a trois méthodes en plus de son constructeur et son destructeur :

- *listen()*, pour écouter les actions réalisés par le joueur.
- *EventIsPossible()*, qui va vérifier la faisabilité de l'action demandé par le joueur (vérifier que le déplacement est possible, que les stats du joueur sont suffisantes, etc...).
- *genCommand()*, qui génère la commande associée à l'action réalisée par le joueur, une fois que celle-ci a été validée par *EventIsPossible()*.



CommandExecutor

Cette classe permet d'exécuter les commandes contenues dans la queue.

Elle possède un attribut :

- *qcmd* un pointeur sur la queue de commande du engine.

Elle ne possède pas de méthodes en plus de son constructeur et son destructeur :

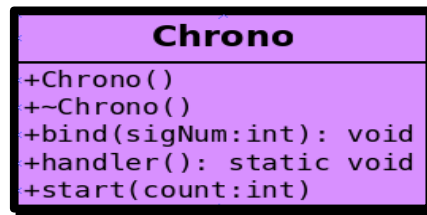


Chrono

Cette classe est celle destinée à surveiller le temps passé et prévenir l'engine toutes les 60 secondes pour passer le tour du joueur.

Elle ne possède pas d'attribut et a trois méthodes en plus de son constructeur et son destructeur :

- `bind()`, lie le handler à l'alarme, de sorte que, quand il y a une interruption d'alarme, le handler est exécuté.
- `handler()`, met à jour le chrono quand le timer a fini de compter.
- `start()`, qui lance le comptage du temps.



ThreadJail

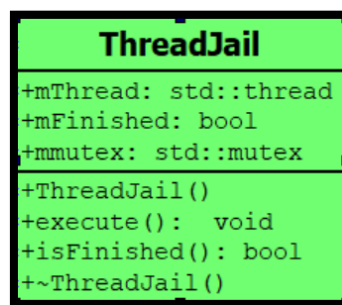
Cette classe permet réaliser un thread pour effectuer des tâches spécifiques en parallèle.

Elle possède quatre attributs :

- `mThread` le thread qui va contenir les tâches à réaliser.
- `mFinished` permettant de connaître l'état du thread, en cours ou achevé.
- `mElapsedTime` qui contiendra le temps écoulé depuis le lancement du thread.
- et `mutex` un mutex pour éviter que les threads n'interfèrent entre eux.

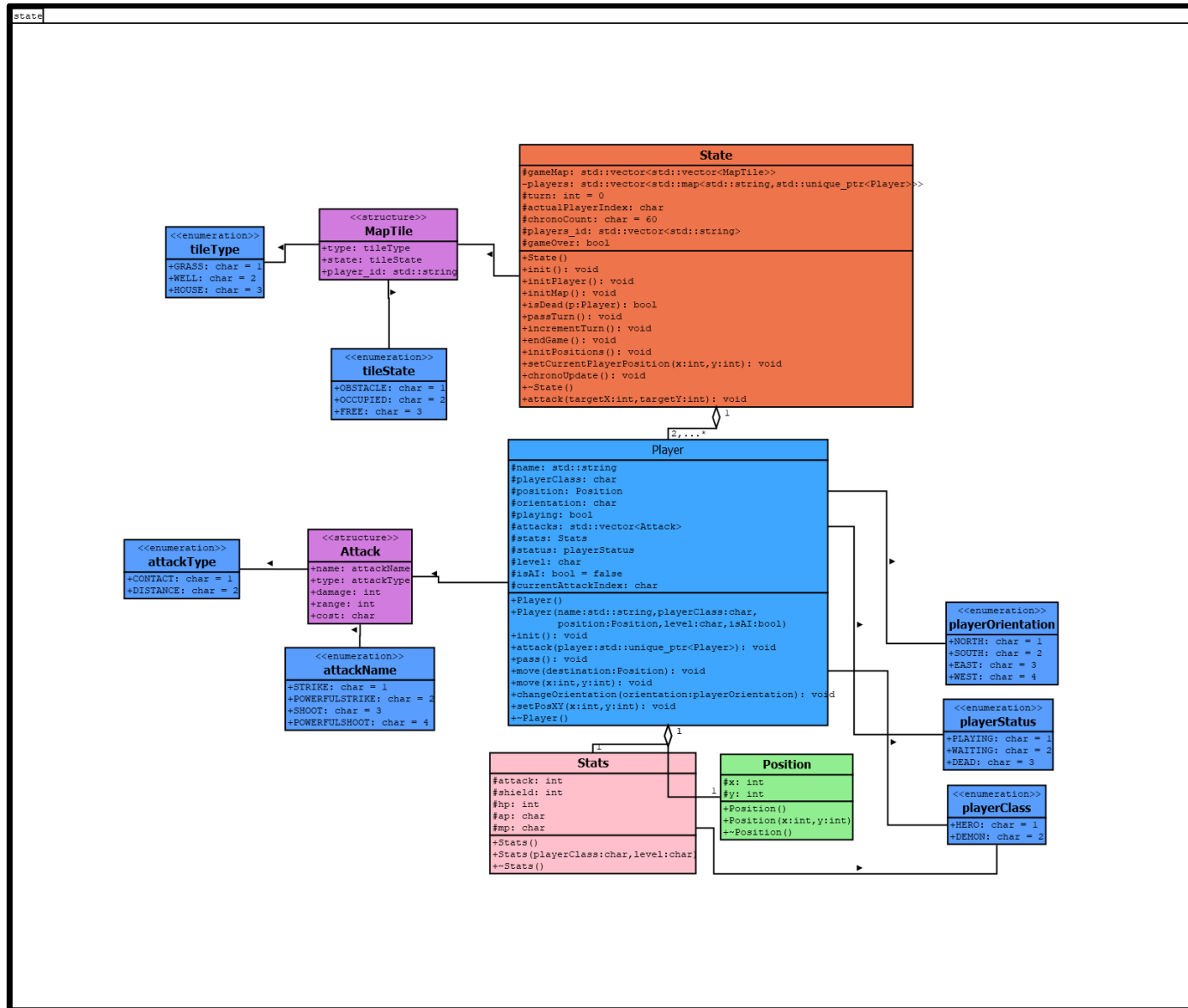
Elle a trois méthodes en plus de son constructeur et son destructeur :

- `execute()`, pour démarrer le thread.
- `isFinished()`, qui renvoie true si le thread est terminé, false sinon



2.5 Ressources

Illustration 1: Diagramme des classes d'état



3 **Rendu : Stratégie et Conception**

Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémentent pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

3.1 Stratégie de rendu d'un état

3.2 Conception logiciel

3.3 Conception logiciel : extension pour les animations

3.4 Ressources

3.5 Exemple de rendu

4 Règles de changement d'états et moteur de jeu

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects liés au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

4.1 Horloge globale

4.2 Changements extérieurs

4.3 Changements autonomes

4.4 Conception logiciel

4.5 Conception logiciel : extension pour l'IA

4.6 Conception logiciel : extension pour la parallélisation

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

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 une 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

