Projet Logiciel Transversal

Pierre-Bernard LE ROUX – Corentin MORISSE

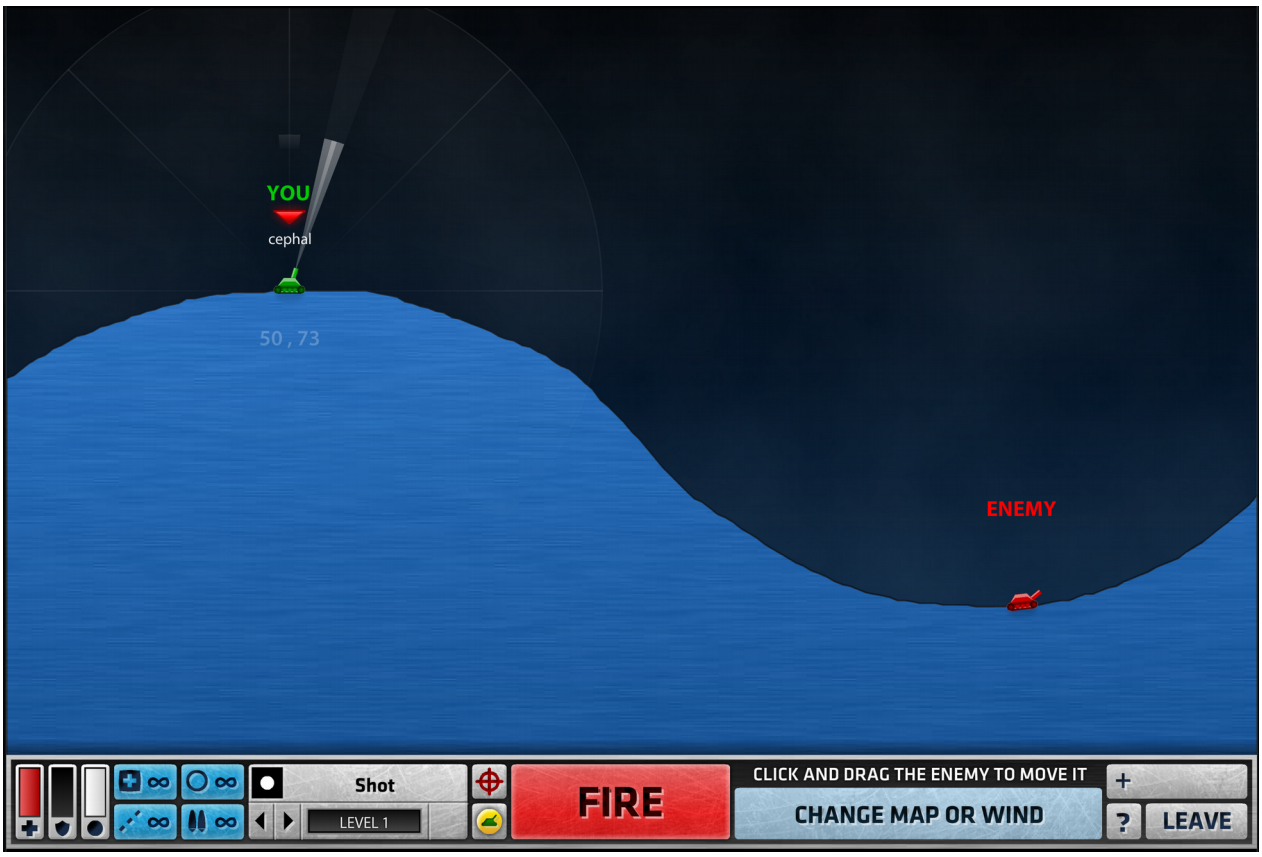


Illustration 1: ShellShock Live 2, archétype de notre jeu

Table des matières

1 Objectif

1.1 Présentation générale

Présenter ici une description générale du projet. On peut s'appuyer sur des schémas ou croquis pour illustrer cette présentation. Éventuellement, proposer des projets existants et/ou captures d'écrans permettant de rapidement comprendre la nature du projet.

Notre projet consiste à créer un jeu tour par tour dont le gameplay s'inspire du jeu *ShellShock Live 2* représenté sur l'Illustration 1 et disponible à l'adresse : http://*www.shellshocklive2.com/* (inscription requise). Il y a deux modes de jeu. Dans le premier mode, l'utilisateur jouera contre une IA. Le second mode sera multijoueur (2 joueurs) : un joueur A affrontera un joueur B.

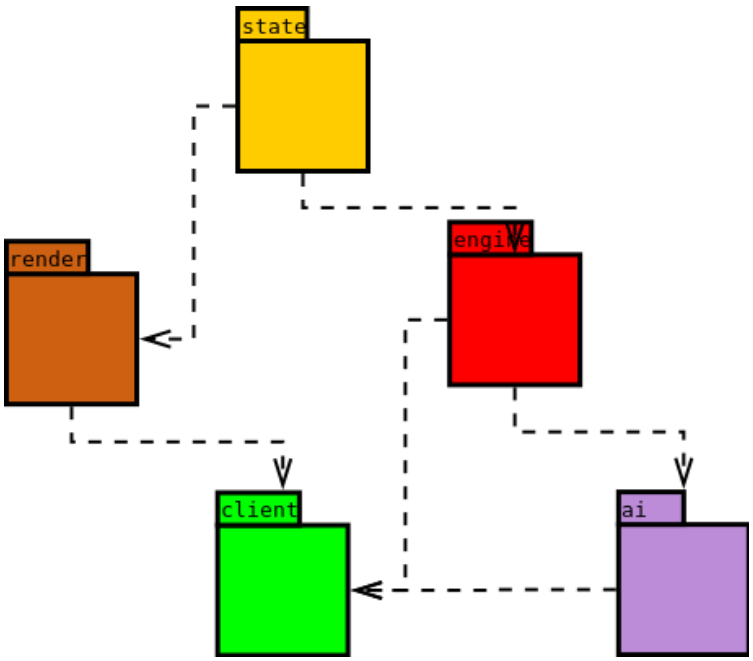
1.2 Règles du jeu

Présenter ici une description des principales règles du jeu. Il doit y avoir suffisamment d'éléments pour pouvoir former entièrement le jeu, sans pour autant entrer dans les détails . Notez que c'est une description en « français » qui est demandé, il n'est pas question d'informatique et de programmation dans cette section.

Les deux joueurs (humain et IA ou humain A et humain B) peuvent déplacer un petit tank sur un décor 2D. Chaque tank dispose du même niveau de vie initial. Le tank possède deux types de tirs pour toucher le tank adverse : le tir horizontal où la visée est simple mais les dégâts sont moindres et le tir vertical où la visée est plus difficile mais les dégâts plus importants. Le premier joueur qui n'a plus de vie a perdu et cela met fin à la partie.

1.3 Conception Logiciel

Présenter ici les packages de votre solution, ainsi que leurs dépendances.



2 Description et conception des états

L'objectif de cette section est une description très fine des états dans le projet. Plusieurs niveaux de descriptions sont attendus. Le premier doit être général, afin que le lecteur puisse comprendre les éléments et principes en jeux. Le niveau suivant est celui de la conception logiciel. Pour ce faire, on présente à la fois un diagramme des classes, ainsi qu'un commentaire détaillé de ce diagramme. Indiquer l'utilisation de patron de conception sera très appriécé. Notez bien que les règles de changement d'état ne sont pas attendues dans cette section, même s'il n'est pas interdit d'illustrer de temps à autre des états par leur possibles changements.

2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (le paysage) et un ensemble d'éléments mobiles (les tanks et les projectiles). Tous les éléments possèdent les propriétés suivantes :

- coordonnées (x,y) dans la grille ;
- identifiant de type d'élément : ce nombre indique la nature de élément (ie classe).

2.1.1 État Éléments fixes

Le paysage est formé par une grille d'éléments nommés « cases ». La taille de cette grille est fixée au démarrage du niveau. Les types de cases sont :

Cases « Obstacle ». Les cases « obstacle » sont des éléments infranchissables pour les éléments mobiles. Le choix de la texture est purement esthétique, et n'a pas d'influence sur l'évolution du jeu.

Cases « Espace». Les cases « espace » sont les éléments franchissables par les éléments mobiles. On considère les types de cases « espace » suivants :

- les espaces « vides » ;
- les espaces « départ joueur », qui définissent la position initiale du joueur (le cas échéant) ;
- les espaces « départ AI», qui définissent les position initiale des AI.

2.1.2 État éléments mobiles

Les éléments mobiles possède une orientation (horizontal gauche, vertical gauche, horizontal droite, vertical droite).

Élément mobile « Tank » . Cet élément est dirigé par le joueur, qui commande la propriété d'orientation et de tir (le tir étant gérer dans le package « engine »). Les tanks disposent également d'un « nombre de pv », qui sert à déterminer les points de vie restant avant de mourir, et d'un « TankTypeID » qui donne leurs couleurs et types (leurs texture). Enfin, on utilise une propriété que l'on nommera « PlayerType », et qui peut prendre les valeurs suivantes :

- PlayerType « realPlayer » : il s'agit du tank sur lequel agit les commandes du joueur humain.
- PlayerType « AIPlayer » : les tanks dirigés par l'AI.

2.2 Conception logiciel

Le diagramme des classes pour les états est présenté en Illustration 2: Diagramme des classes d'état, dont nous pouvons mettre en évidence les groupes de classes suivants :

Classes Element. Toute la hiérarchie des classes filles d'*Element* (en jaune) permettent de représenter les différentes catégories et types d'élément. Etant donné qu'il n'y a pas de solution standard efficace en C++ pour faire de l'introspection, nous avons opté pour des méthodes comme *isStatic()* qui indiquent la classe d'un objet. Ainsi, une fois la classe identifié, il suffit de faire un simple cast.

Conteneurs d'élément. Viennent ensuite les classes *State*, *ElementList et ElementGrid* qui permettent de contenir des ensembles d'éléments. *ElementList* contient une liste d'éléments, et *ElementGrid* étends ce conteneur pour lui ajouter des fonctions permettant de gérer une grille. Enfin, la classe *State* est le conteneur principal, avec une grille pour le niveau, et une liste pour les éléments mobiles (les tanks).

2.3 Conception logiciel : extension pour le rendu

Observateurs de changements. Dans le diagramme de classes, nous présentons en rouge les classes permettant à des tiers de réagir lorsqu'un événement se produit dans l'un des éléments d'état. Les premiers intéressés par cette fonctionnalité sont les clients de rendu, qui pourront mettre à jour les textures/sprites en fonction des changements au sein de l'état actuel.
Les observateurs implantent l'interface *StateObserver* pour être avertis des changements de propriétés d'état. Pour connaître la nature du changement, ils analysent l'instance de *StateEvent*. La conception de ces outils suit le patron ***Observer***.

2.4 Conception logiciel : extension pour le moteur de jeu

Au sein des différentes classes d'*Element*, ainsi que leurs conteneurs comme *ElementList* et *State*, nous avons ajouté des méthodes de clonage et de comparaison :

- Méthodes *clone()* : elles renvoie une copie complète (les sous-éléments sont également copiés) de l'instance.

- Méthodes *equals()* : elles permettent de comparer l'instance actuelle avec une autre, et renvoie *true* si les deux instances ont un contenu identique.

2.5 Ressources

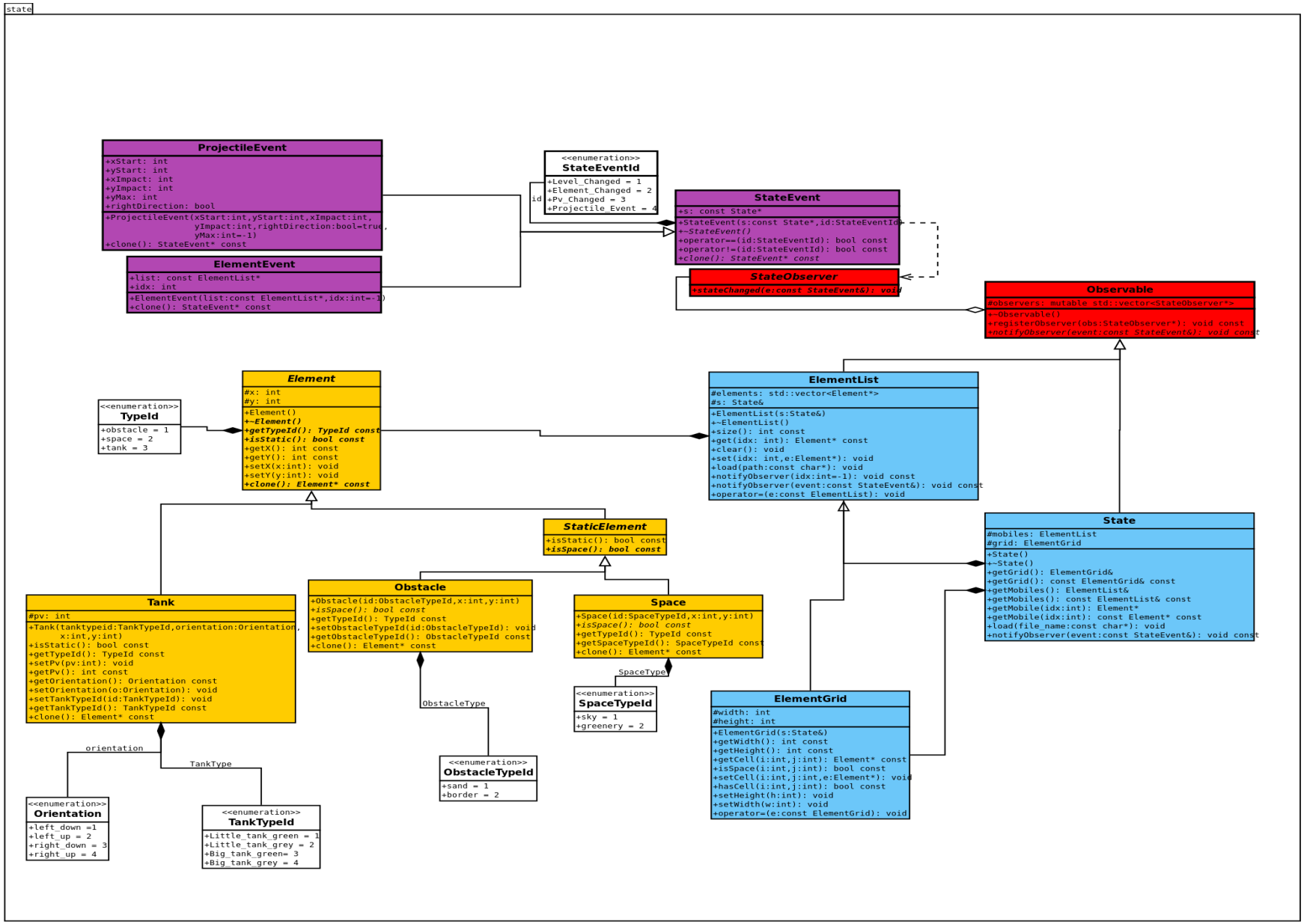


Illustration 2: Diagramme des classes d'état

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'implémente 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

Pour le rendu d'un état, nous découpons la scène à rendre en plans (ou «layers»): un plan pour le niveau (sable, ciel, etc.) et un plan pour les éléments mobiles (tank et projectiles). Chaque plan contiendra deux informations bas-niveau qui seront transmises à la carte graphique: une unique texture contenant les tuiles (ou «tiles»), et une unique matrice avec la position des éléments et les coordonnées dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.

3.2 Conception logiciel

Le diagramme des classes pour le rendu général, indépendante de toute librairie graphique, est présenté en Illustration 4.

Plans et Surfaces.

Le cœur du rendu réside dans le groupe (en rouge) autour de la classe Layer. Le principal objectif des instances de Layer est de donner les informations basiques pour former les éléments bas-niveau à transmettre à la carte graphique. Ces informations sont données à une implantation de Surface. Cette implantation non représentée dans le diagramme, dépendra de la librairie graphique choisie (SFML ici).

La première information donnée est la texture du plan, via la méthode loadTexture(). Les informations qui permettrons à l'implantation de Surface de former la matrice des positions seront données via la méthode setSprite().

Scène.

Tous les plans sont regroupés au sein d'une instance de Scene. Les instances de cette classe seront liées («bind») à un état particulier. L'implantation pour SFML fourni des surfaces via les méthodes setXXXSurface(). Notons bien que les instances ont pour rôle de remplir les surfaces, mais pas de les rendre : cela restera le travail de la librairie choisie.

Tuiles.

La classe Tile a pour rôle la définition de tuiles au sein d'une texture particulière avec leur position dans cette texture.

3.3 Conception logiciel : extension pour les animations

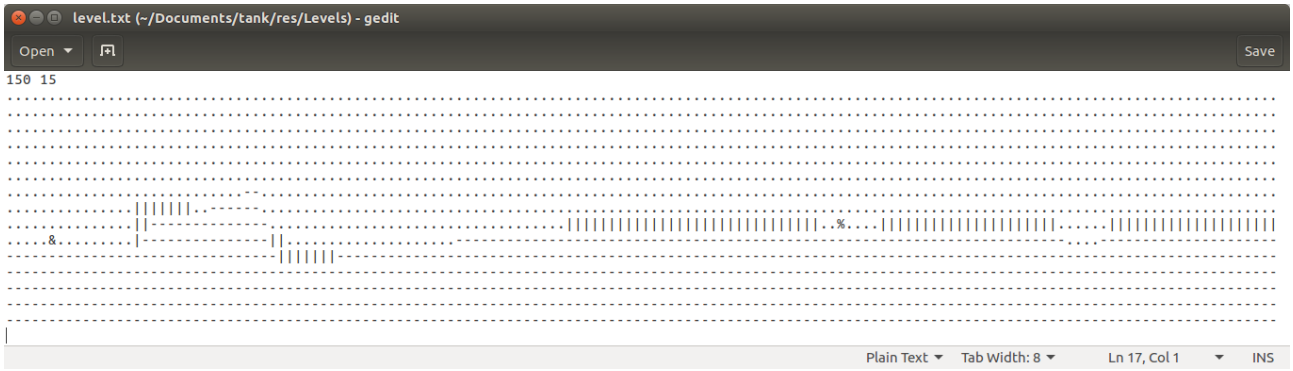
Animation. Les animations sont gérés par la classe *Animation*. Chaque plan tient une liste de ces animations, et le client graphique fait appel au méthodes de mise à jour pour faire évoluer ses surfaces (via les méthodes setNextTile() d'*Animation* puis update() du layer). Nous faisons ce choix car leurs évolutions ne dépendent pas de l'état (néanmoins elle empêche de passer au tour suivant).
Le temps « d'explosion » est défini via la variable countTimeExplosion qui est incrémenter à chaque update().

3.4 Ressources

Voici un exemple de textures pour le plan grille : 

3.5 Codage d'un niveau

Le rendu de chaque niveau du jeu est défini dans un fichier .txt. Voici un exemple :



Les différents symboles qui représentent les éléments du rendu (Illustration 3) sont :

- « & » : représente le tank 1 commandé par un joueur humain ;
- « % » : réprésentent le tank 2 commandé par un joueur humain ou par l'IA ;
- « . » : représente le ciel bleu ;
- « - » : représente le sol ;
- « | » : représente les éléments de verdure (en vert sur Illustration 3).

3.6 Exemple de rendu

Voici typiquement un exemple de rendu pour un niveau :

Illustration 3: Aperçu du rendu

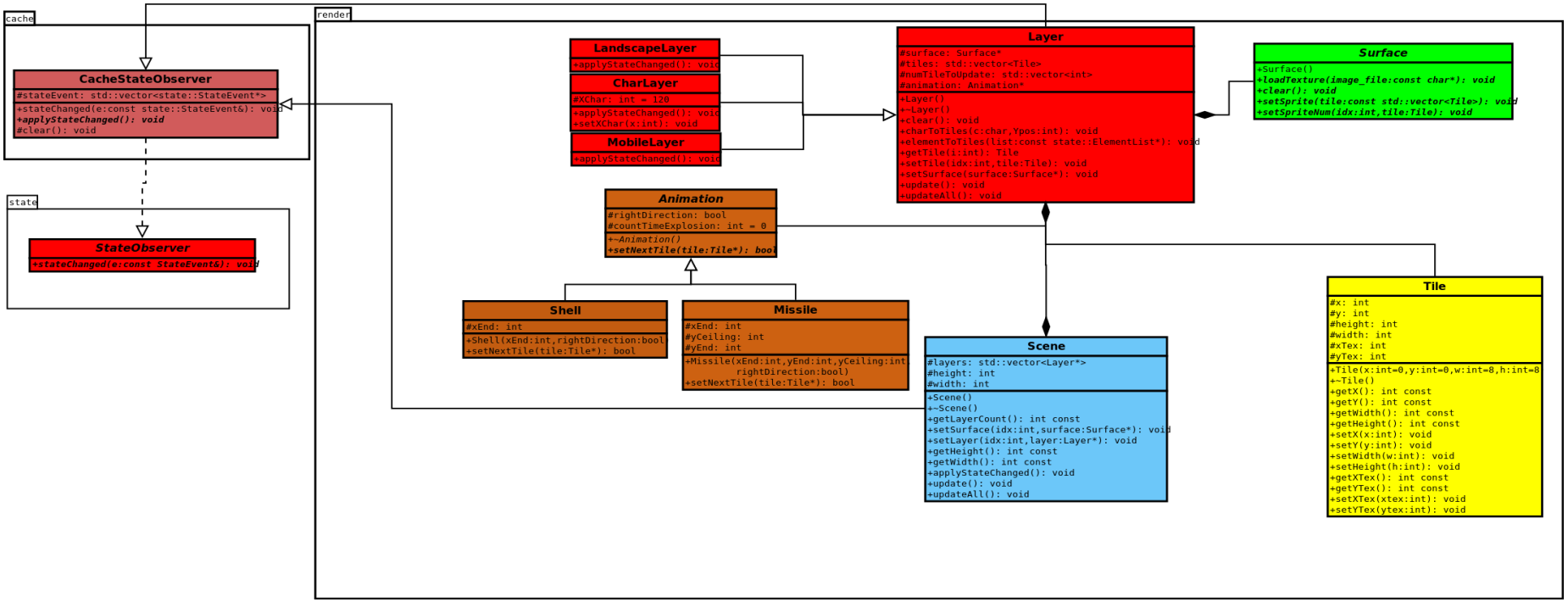


Illustration 4: Diagramme de classes pour le rendu

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 lié 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

Les changements d'état suivent deux horloge : de manière régulière, le thread chargé de l'engine vérifira les commandes pour charger un niveau et celle pour changer de mode ; et lorsque l'on appuit sur « fin du tour » les autres actions stockés sont effectuer (si licite).

4.2 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieurs, comme la pression sur une touche ou un ordre provenant du réseau :

1. Commandes principales : « Charger un niveau » : On fabrique un état initial à partir d'un fichier.

2. Commande « Mode » : On modifie le mode actuel du jeu, comme «normal » ou « pause ».

3. Commande « déplacement vers » : Si cela est possible (pas de mur), la position du personnage est modifiée.

4. Commande « Orientation » : On change l'orientation du personnage.

5. Commande « Shot » : On tire un projectile, calcule et anime son déplacement.

6. Commande « fin du tour » : On finit le tour et applique les commandes stockés

4.3 Changements autonomes

Les changements autonomes sont le déplacement des projectiles après leurs création via la commande « shot ».

4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Illustration 5.
L'ensemble du moteur de jeu repose sur un patron de conception de type Command, et a pour but la mise en œuvre différée de commandes extérieures sur l'état du jeu.

Classes Command.
Le rôle de ces classes est de représenter une commande extérieure, provenant
par exemple d'une touche au clavier (ou tout autre source). Notons bien que ces classes ne gère

absolument pas l'origine des commandes, ce sont d'autres éléments en dehors du moteur de jeu qui fabriquerons les instances de ces classes. A ces classes, on a défini un type de commande avec CommandCategory pour identifier précisément la classe d'une instance et s'assurer que chaque commande est exclusive. Par exemple, toutes les commandes d'orientation pour un personnage sont exclusives : on ne peut pas demander d'aller à la fois à gauche et à droite. Pour l'assurer, toutes ces commandes ont la même catégorie, et par la suite, on ne prendra toujours qu'une seule commande par catégorie (la plus récente).

Engine.
C'est le coeur du moteur. Elle stocke les commandes dans une instance de CommandSet.
Lorsqu'une nouvelle époque démarre, ie lorsqu'on a appelé la commande « fin de tour », le principal travail du moteur est de transmettre les commandes à une instance de Ruler.
C'est cette classe qui applique les règles du jeu. Plus précisément, et en fonction des commandes ou de règles de mises à jour automatiques, elle construit une liste d'actions. Ces actions transforment l'état courant pour le faire évoluer vers l'état suivant.

Action.
Le rôle de ces classes est de représenter une modification particulière d'un état du jeu.
Notons bien que ce ne sont pas les règles du jeu : chaque instance de ces classes applique la modification qu'elle contient, sans se demander si cela a un sens.

4.5 Conception logiciel : extension pour l'IA

4.6 Conception logiciel : extension pour la parallélisation

Engine.
Dans le but de pouvoir utiliser le moteur de jeu dans un thread séparé, nous avons ajouté un deuxième jeu de commandes appelé waitingCommands qui récupère les commandes envoyés au moteur de jeu lorsque celui-ci met à jour l'état du jeu. Plus de détails dans la section 6.1.1.

4.7 Commande

Tir : touche espace

Déplacement : touche right/left

Orientation : touche Q (gauche), Z (haut gauche), E (haut droit), D (droite)

Mode : touche échap pour la pause, P pour lancer l'IA (le joueur 1 joue HeuristicAI, le joueur 2 DumbAI)

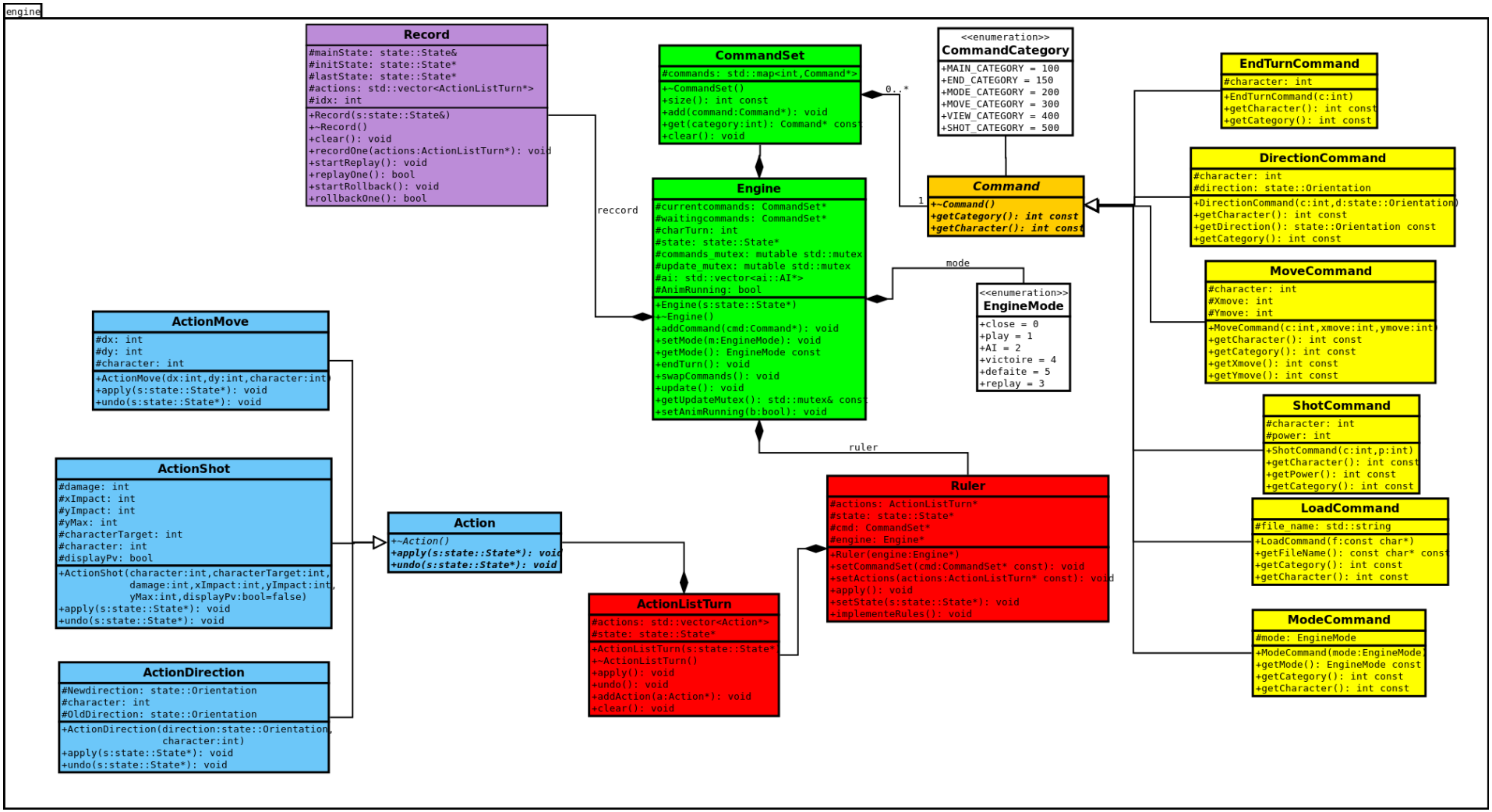


Illustration 5: Diagrammes des classes pour le moteur de jeu

5 Intelligence Artificielle

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

Pour élaborer l'intelligence artificielle qui différa selon le niveau de difficulté choisi par le joueur avant le lancement de la partie. En effet, par ordre de chronologie dans l'avancement du projet et difficulté, on aura la possibilité d'affronter une IA minimale (niveau facile), une IA basée sur des heuristiques (niveau intermédiaire) et une IA basée sur les arbres de recherche.

5.1.1 Intelligence minimale

La conception de l'IA minimale n'est pas très complexe puisqu'elle consiste à faire des déplacements aléatoires et des tirs dont la direction et le type de projectile est aussi aléatoire. Les commandes des classes associées (DirectionCommand, MoveCommand et ShotCommand) sont donc aléatoires à chaque tour. Il est en conséquence aisé pour le joueur humain de remporter la partie.

5.1.2 Intelligence basée sur des heuristiques

La conception de l'IA basée sur des heuristiques, plus élaborée, repose sur une stratégie que nous avons établi et à adopter par l'IA pour chaque partie. A l'aide de conditions, on peut décomposer les différentes situations dans lesquelles peut se retrouver le char dirigée par l'IA. Par exemple, si le char de l'IA se trouve à la gauche du char du joueur humain, on tourne le canon à droite en premier lieu. En évaluant la distance du char de l'IA par rapport au char du joueur humain, on peut facilement prévoir le dosage approximatif dans la puissance du tir pour faire mouche. Orienter le dosage du tir ne signifie pas que l'IA touchera sa cible à chaque tour mais cela évitera que le missile se dirige à l'autre bout de la carte alors que le char du joueur humain est au milieu.

5.1.3 Intelligence basée sur les arbres de recherche

L'IA basée sur les arbres de recherche la plus performante puisqu'elle touchera sa cible à chaque tour en infligeant un nombre de dégâts maximum. En effet, sa conception repose l'algorithme minimax. En parcourant dans un graphe l'ensemble des états possibles (définis selon les différentes commandes que l'IA peut faire à chaque tour à savoir les commandes de mouvement et de tir), on pourra trouver le bon scénario/série de commandes qui inflige le maximum dégâts au joueur humain. Pour trouver, ce qu'on appelle la « valeur du jeu », il nous faut remplir chaque feuille (nœuds situés à la profondeur maximale ou nœuds qui correspondent à un état où la partie est erminée) de l'arbre de recherche utilisée puis remonter l'arbre selon l'algorithme du minimax. Pour un nœud donné, on doit donc déterminer le calcul de cette valeur qui permettra d'indiquer si l'état correspondant est favorable ou défavorable à l'AI. Nous avons opté pour la formule suivante : valeur du nœud = Dégâts infligeable à l'AI – Dégâts infligés par l'AI au joueur humain. Nous avons choisi une profondeur de 5 pour l'algorithme. De plus, si un feuille de l'arbre correspond à un état de fin de partie est atteinte par l'algorithme alors le nœud vaut -1000 si l'état correspond à la défaite de l'AI et +1000 s'il correspond à la victoire de l'AI.
Par ailleurs, nous n'envisageons pas de dupliquer l'état du jeu à chaque sommet du graphe d'état : compte tenu du nombre de nœuds que nous allons traiter, nous aurions rapidement des problèmes de

mémoire. Nous n'allons considérer qu'un seul état que nous modifion suivant la direction choisie par la recherche. Si le sommet suivant est à une époque suivante, i.e. on descend dans l'arbre de recherche en appliquant sur l'état les commandes associées. Si on atteint une feuille de l'arbre et que l'on souhaite le remonter ie revenir à l'état précédent, on annule les commandes associées grâce à la fonction undo() du Ruler, et notre état retrouve sa forme passée.

5.2 Conception logiciel

Classes AI. Toutes les formes d'intelligence artificielle implantent la classe abstraite AI visible sur Illustration 6: Diagramme des classes l'IA. Le rôle de ces classes est de fournir un ensemble de commandes à transmettre au moteur de jeu. Notons qu'il n'y a pas une instance par personnage, mais qu'une instance doit fournir les commandes pour tous les personnages. La classe DumbAI implante l'intelligence minimale, telle que présentée ci dessus.

5.3 Conception logiciel : extension pour l'IA composée

Classe DistanceUtility. Ensemble des fonctions concernant les distances nécésaire à la determination d'une stratégie ; elle permet de pouvoir facilement l'utilisé sur une autre IA.

Classe HeuristicAI. Effectue l'action optimale pour le tour en cours (pas d'anticipation).
Pour ce faire, elle utilise les fonctions de **DistanceUtility** afin de terminer si elle est à porté de tir et si elle peut utiliser ses missiles (suffisamment près).

Classe EvolvedAI. Permet de disposer d'une base disposant d'un object DistanceUtility sur laquel branché une éventuelle AI inteligente en vue d'une évolution (une IA avancée par exemple).

5.4 Conception logiciel : extension pour IA avancée

Classe TreeAI. C'est la classe mère de l'IA avancée. Elle contient le nœud racine (root) qui représente l'état actuel du jeu à partir duquel l'IA va joué son prochain coup.

Classe Gardener. C'est la classe qui va gérer entièrement l'intelligence de l'IA avancée en utilisant l'algorithme du minimax contenu dans le tableau nodeWarehouse du Gardener. Le Gardener contient un pointeur vers une instance de Ruler car ce dernier permet d'appliquer les commandes lorsqu'on descend de l'arbre ainsi que la méthode undo() (via ActionListTurn accessible à partir du Ruler) qui permet de remonter l'arbre/annuler une commande.

Classe Node. Il s'agit de la classe qui représente un nœud de l'arbre et par conséquent contient un score et la profondeur du nœud notamment.

5.5 Conception logiciel : extension pour la parallélisation

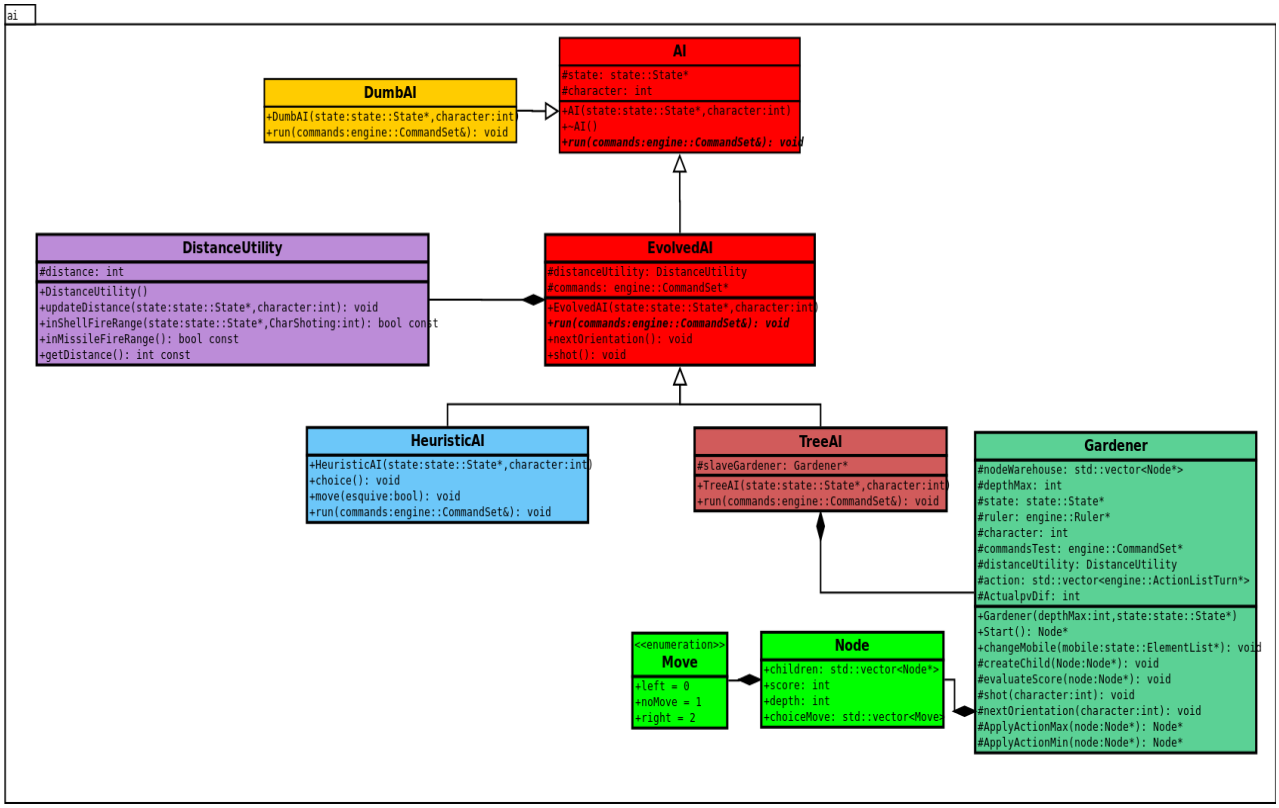


Illustration 6: Diagramme des classes l'IA

6 Modularisation

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 un 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

Notre objectif ici est de placer le moteur de jeu sur un thread, puis le moteur de rendu sur un autre thread. Nous avons deux type d'information qui transite d'un module à l'autre : les commandes et les notifications de rendu.

Commandes. Il s'agit ici des touches du clavier et des boutons pressées par l'utilisateur. Celle-ci peuvent arriver à n'importe quel moment, y compris lorsque l'état du jeu est mis à jour. Pour résoudre ce problème, nous proposons d'utiliser un double tampon de commandes. L'un contiendra les commandes actuellement traitées par une mise à jour de l'état du jeu, et l'autre accueillera les nouvelles commandes. A chaque nouvelle mise à jour de l'état du jeu, on permute les deux tampons : celui qui accueillait les commandes devient celui traité par la mise à jour, et l'autre devient disponible pour accueillir les futures commandes. Ainsi, il existe toujours un tampon capable de recevoir des commandes, et cela sans aucun blocage. Il en résulte une parfaite répartition des traitements, ainsi qu'une latence au plus égale au temps entre deux époques de jeu.

Notifications de rendu. Ce cas est problématique, car il n'est pas raisonnable d'adopter une approche par double tampon. En effet, cela implique un doublement de l'état du jeu (un en cours de rendu, l'autre en cours de mise à jour), ce qui augmente de manière significative l'utilisation mémoire et processeurs, ainsi que la latence du jeu. Nous nous sommes donc tournés donc vers une solution avec recouvrement entre les deux modules, en proposant une approche qui le minimise autant que possible.
Pour ce faire, nous ajoutons un tampon (*CacheStateObserver*) qui va «absorber» toutes les notifications de changement qu'émet une mise à jour de l'état du jeu. Puis, lorsqu'une mise à jour est terminée, le moteur de jeu envoie un signal au moteur de rendu. Celui-ci, lorsqu'il s'apprête à envoyer ses données à la carte graphique, regarde si ce signal a été émis. Si c'est le cas, il vide le tampon de notification pour modifier ses données graphiques avant d'effectuer ses tâches habituelles. Lors de cette étape, une mise à jour de l'état du jeu ne peut avoir lieu, puisque le moteur de rendu a besoin des données de l'état pour mettre à jour les siennes. Nous avons donc ici un recouvrement entre les deux processus. Cependant, la quantité de mémoire et de processeur utilisée est très faible devant celles utilisées par la mise à jour de l'état du jeu et par le rendu.

Classe Pilote. Les threads *engine* et *rendu* tournent sur la classe pilote, et plus précisement sur la fonction *runEngine()* et la fonction *runRendu()*. Pour pouvoir facilement changer l'intégration du rendu, on a utilisé une interface *PiloteRendu* qui nous permet de n'avoir qu'à remplacer la classe *PiloteSFML* pour changer de librairie graphique.

6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau

6.4 Conception logiciel : client Android

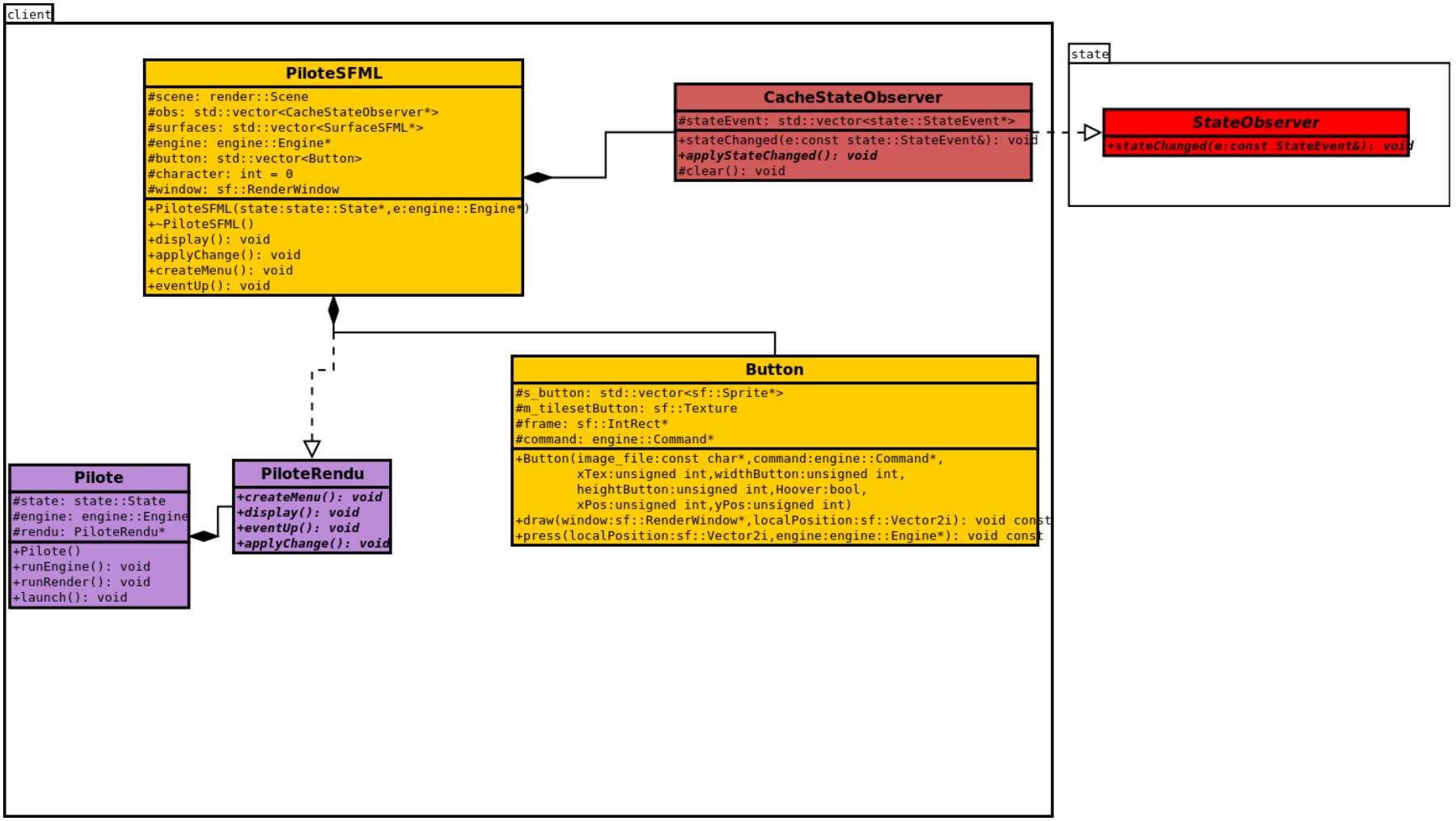


Illustration 7: Diagramme de classes pour la modularisation