Projet Logiciel Transversal

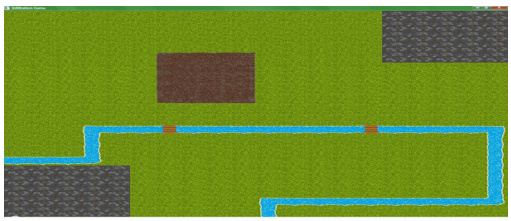
Nicolas BENOIT – Clément GODBILLE

Table des matières

1 Objectif

1.1 Présentation générale

Le projet est un jeu vidéo de stratégie au tour par tour, baptisé « Medieval Minimal War ». Le jeu simule une bataille rangée entre 2 armées à l'époque médiévale, les soldats de chaque armée évoluant sur un damier représentant le champ de bataille. Les joueurs devront remplir certains objectifs configurables pour remporter la partie (élimination de l'armée adverse, capture d'un drapeau…).



Ci-contre : un aperçu de la future allure du champ de bataille. Les armées des joueurs se déploient en bas à gauche et en haut à droite dans les zones grisées. Le décor est généré aléatoirement (rivières, montagnes, etc.) et offre une certaine dimension tactique.

1.2 Règles du jeu

Depuis l'écran-titre, le joueur peut choisir entre différents modes de jeu: bataille contre l'IA ou bataille en multijoueur à 1 contre 1. Une fois un mode lancé, le joueur-hôte (le joueur humain en partie contre l'IA ou l'un des deux joueurs humains en multijoueur) détermine toutes les options de la partie lors de la **phase de préparation** : points d'armée disponibles pour chaque joueur, objectif à accomplir (capture de drapeau, élimination), nature de la carte (terrain montagneux, forestier...).

Dans un second temps, les joueurs sélectionnent leurs armées en utilisant les points dont ils disposent : les joueurs peuvent par exemple avoir un capital de 50 points à dépenser pour acheter quatre archers à 12 points l'unité, ou cinq lanciers à 10 points l'unité. Différentes unités seront ainsi disponibles dont le coût dépendra de leur puissance. Lorsque chaque joueur se déclare prêt, la partie est lancée.

Une fois la partie lancée, les joueurs rentrent dans la **phase de combat** qui se déroule sur le champ de bataille généré pour la partie suivant les options choisies en préparation. Chaque joueur déploie alors ses troupes dans sa zone de départ. Dès que les troupes sont placées, la personne qui commence est tirée aléatoirement et peut commencer son tour.

Durant le tour d'un joueur, ce dernier peut déplacer ses soldats suivant leurs Points de Mouvements (PM) : un fantassin avec 4 PM pourra se déplacer d'au mieux 4 cases durant ce tour. Un soldat peut ensuite effectuer une unique attaque par tour vers une unité ennemie sous réserve qu'il soit à portée (par exemple au corps-à-corps pour les guerriers ou à une certaine distance pour les archers).

Une fois que le joueur est satisfait de son tour, il peut terminer son tour, et son adversaire peut jouer. La partie s'achève lorsque les objectifs définis en début de partie ont été remplis par l'un des deux joueurs.

2 Description et conception des états

L'objectif de cette section est de dégager l'ensemble des éléments qui permettent de définir entièrement un état du jeu donné. Les états seront décrits dans un premier temps de manière générale puis avec davantage de précision via un diagramme des classes.

2.1 Description des états

Un état du jeu consiste en une combinaison de deux types d'éléments : des personnages (les soldats de chaque joueur) et des cases du décor (rivières, château, montagne…). Les éléments Personnages évoluent sur les cases du Décor.

Pour tout élément on définit :
_ sa position dans la grille (coordonnée x,y) ;
_ son identifiant (quel type de personnage, quel type de terrain…).

Les éléments Décors sont des éléments immobiles du jeu. Les éléments Personnages peuvent évoluer sur le Décors suivant ce que l'on appelle leurs Accès :

Les Accès dépendent de l'élément du décor : la case rempart (en gris) n'est accessible que par les côtés gauche et droit ; la case montagne (en marron) n'est accessible que par droite et bas.

Un personnage ne peut passer d'une case à une autre que si les accès des deux cases sont compatibles.

Les éléments Décors possèdent également l'attribut Hauteur qui servira à simuler la notion de relief dans le jeu, qui sera pris en compte lors des tirs des archers ou des canons notamment (un archer situé à une hauteur 0 ne pourra pas tirer au-delà d'un rempart de hauteur 2 car il lui bloque la vue).

Les éléments Personnages possèdent un ensemble de caractéristiques : Points de Vie (PV), Points d'Action (PA), Points d'Attaque, Orientation (vers la gauche ou la droite), Equipe (rouge ou bleue), Portée d'attaque Maximale et Portée d'attaque Minimale. Les valeurs pour chacun des différents types de personnages (Lancier, Spadassin, Cavalier, Roi, Archer et Canon).

2.2 Conception logiciel

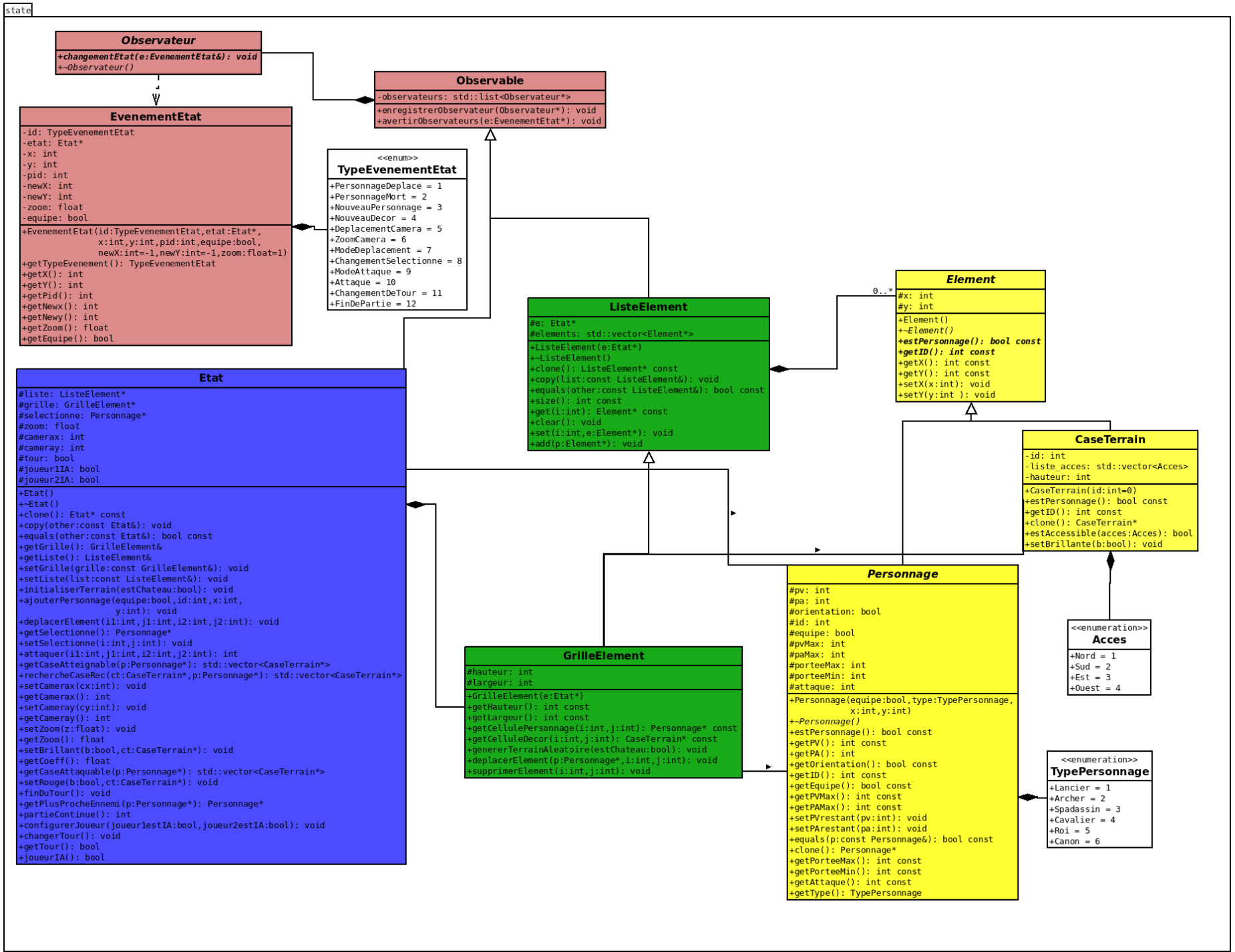
La gestion des différents éléments est facilité par l'existence de 2 classes conteneurs : ListeElement et GrilleElement.

- ListeElement stocke des Elements et servira à lister les Elements du décor et les soldats présents.
- GrilleElement fourni des renseignements complémentaires sur le positionnement des Elements sur le plateau.

Une classe Etat contient une ListeElement et une GrilleElement et gère l'ensemble de l'état du jeu.

Nous définissons de plus un pattern Observer afin d'envoyer aux packages intéressés (notamment le rendu) les changements d'état.

Classe	Description
Etat	Définie la totalité de l'état du jeu. Contient une ListeElement et une GrilleElement
ListeElement	Contient les Elements du jeu.
GrilleElement	Contient des informations liées à la grille de jeu.
Element	Contient Personnage et CaseTerrain
Personnage	Définie un personnage
CaseTerrain	Définie une case du terrain
Observable	Est héritée par Etat et ListeElement. Envoie les avertissements de changement d'état via l'Observateur.
Observateur	Reçois les notifications d'Observable.
Evenement Etat	Définie la nature d'un changement d'état.
TypeEvenementEtat	Discrimine les différents types de changement d'état.
Acces	Discrimine les différentes possibilités d'accès d'une case donnée


state
Observateur
+changementEtat(e:EvenementEtat&): void
+~Observateur()
Observable
-observateurs: std::list<Observateur*>
+enregistrerObservateur(Observateur*): void
+avertirObservateurs(e:EvenementEtat*): void
EvenementEtat
-id: TypeEvenementEtat
-etat: Etat*
-x: int
-y: int
-pid: int
-newX: int
-newY: int
-zoom: float
-equipe: bool
+EvenementEtat(id:TypeEvenementEtat,etat:Etat*, x:int,y:int,pid:int,equipe:bool, newX:int=-1,newY:int=-1,zoom:float=1)
+getTypeEvenement(): TypeEvenementEtat
+getX(): int
+getY(): int
+getPid(): int
+getNewx(): int
+getNewy(): int
+getZoom(): float
+getEquipe(): bool
<<enum>>
TypeEvenementEtat
+PersonnageDeplace = 1
+PersonnageMort = 2
+NouveauPersonnage = 3
+NouveauDecor = 4
+DeplacementCamera = 5
+ZoomCamera = 6
+ModeDeplacement = 7
+ChangementSelectionne = 8
+ModeAttaque = 9
+Attaque = 10
+ChangementDeTour = 11
+FinDePartie = 12
ListeElement
#e: Etat*
#elements: std::vector<Element*>
+ListeElement(e:Etat*)
+~ListeElement()
+clone(): ListeElement* const
+copy(list:const ListeElement&): void
+equals(other:const ListeElement&): bool const
+size(): int const
+get(i:int): Element* const
+clear(): void
+set(i:int,e:Element*): void
+add(p:Element*): void
0..*
Element
#x: int
#y: int
+Element()
+~Element()
+estPersonnage(): bool const
+getID(): int const
+getX(): int const
+getY(): int const
+setX(x:int): void
+setY(y:int): void
Etat
#liste: ListeElement*
#grille: GrilleElement*
#selectionne: Personnage*
#zoom: float
#camerax: int
#cameray: int
#tour: bool
#joueur1IA: bool
#joueur2IA: bool
+Etat()
+~Etat()
+clone(): Etat* const
+copy(other:const Etat&): void
+equals(other:const Etat&): bool const
+getGrille(): GrilleElement&
+getListe(): ListeElement&
+setGrille(grille:const GrilleElement&): void
+setListe(list:const ListeElement&): void
+initialiserTerrain(estChateau:bool): void
+ajouterPersonnage(equipe:bool,id:int,x:int, y:int): void
+deplacerElement(i1:int,j1:int,i2:int,j2:int): void
+getSelectionne(): Personnage*
+setSelectionne(i:int,j:int): void
+attaquer(i1:int,j1:int,i2:int,j2:int): int
+getCaseAtteignable(p:Personnage*): std::vector<CaseTerrain*>
+rechercheCaseRec(ct:CaseTerrain*,p:Personnage*): std::vector<CaseTerrain*>
+setCamerax(cx:int): void
+getCamerax(): int
+setCameray(cy:int): void
+getCameray(): int
+setZoom(z:float): void
+getZoom(): float
+setBrillant(b:bool,ct:CaseTerrain*): void
+getCoeff(): float
+getCaseAttaquable(p:Personnage*): std::vector<CaseTerrain*>
+setRouge(b:bool,ct:CaseTerrain*): void
+finDuTour(): void
+getPlusProcheEnnemi(p:Personnage*): Personnage*
+partieContinue(): int
+configurerJoueur(joueur1estIA:bool,joueur2estIA:bool): void
+changerTour(): void
+getTour(): bool
+joueurIA(): bool
CaseTerrain
-id: int
-liste_acces: std::vector<Acces>
-hauteur: int
+CaseTerrain(id:int=0)
+estPersonnage(): bool const
+getID(): int const
+clone(): CaseTerrain*
+estAccessible(acces:Acces): bool
+setBrillante(b:bool): void
GrilleElement
#hauteur: int
#largeur: int
+GrilleElement(e:Etat*)
+getHauteur(): int const
+getLargeur(): int const
+getCellulePersonnage(i:int,j:int): Personnage* const
+getCelluleDecor(i:int,j:int): CaseTerrain* const
+genererTerrainAleatoire(estChateau:bool): void
+deplacerElement(p:Personnage*,i:int,j:int): void
+supprimerElement(i:int,j:int): void
Personnage
#pv: int
#pa: int
#orientation: bool
#id: int
#equipe: bool
#pvMax: int
#paMax: int
#porteeMax: int
#porteeMin: int
#attaque: int
+Personnage(equipe:bool,type:TypePersonnage, x:int,y:int)
+~Personnage()
+estPersonnage(): bool const
+getPV(): int const
+getPA(): int
+getOrientation(): bool const
+getID(): int const
+getEquipe(): bool const
+getPVMax(): int const
+getPAMax(): int const
+setPVrestant(pv:int): void
+setPArestant(pa:int): void
+equals(p:const Personnage&): bool const
+clone(): Personnage*
+getPorteeMax(): int const
+getPorteeMin(): int const
+getAttaque(): int const
+getType(): TypePersonnage
<<enumeration>>
Acces
+Nord = 1
+Sud = 2
+Est = 3
+Ouest = 4
<<enumeration>>
TypePersonnage
+Lancier = 1
+Archer = 2
+Spadassin = 3
+Cavalier = 4
+Roi = 5
+Canon = 6

3 Rendu : stratégie et conception

L'objectif de cette section est d'exposer la stratégie et les techniques de conception utilisées pour la création du package Rendu, responsable de l'affichage des textures.

3.1 Stratégie de rendu d'un état

Le rendu d'un état de notre projet n'utilise pas de techniques de pré-traitement du rendu par le CPU et envoie en permanence des appels sf ::draw(). C'est une technique barbare qui est susceptible de ralentir fortement le déroulement du programme mais nous pensons que la simplicité du jeu, sa nature (un jeu au tour par tour où il y a peu d'actions par minute) et la puissance des cartes graphiques actuels font que l'optimisation de la vitesse d'affichage n'est pas prioritaire pour le moment.

Le rendu est reproduit sous la forme d'une Scène qui possède deux Couches : une couche pour le décor et une couche pour les personnages. La couche personnage est positionnée par dessus la couche décor. Chaque couche est composée de Tuiles, et on différencie les Tuiles Statiques des Tuiles Animées. Les différentes textures du jeu sont chargées dans Tuile Statique, et Tuile Animée est composée de plusieurs Tuiles Statiques qui défilent à une vitesse de rafraîchissement donnée.

A tout instant un état donné est reproduit en dessinant la totalité des tuiles de la carte. Un pattern Observer communique entre les packages State et Render pour mettre à jour le rendu en cas de changement d'état.

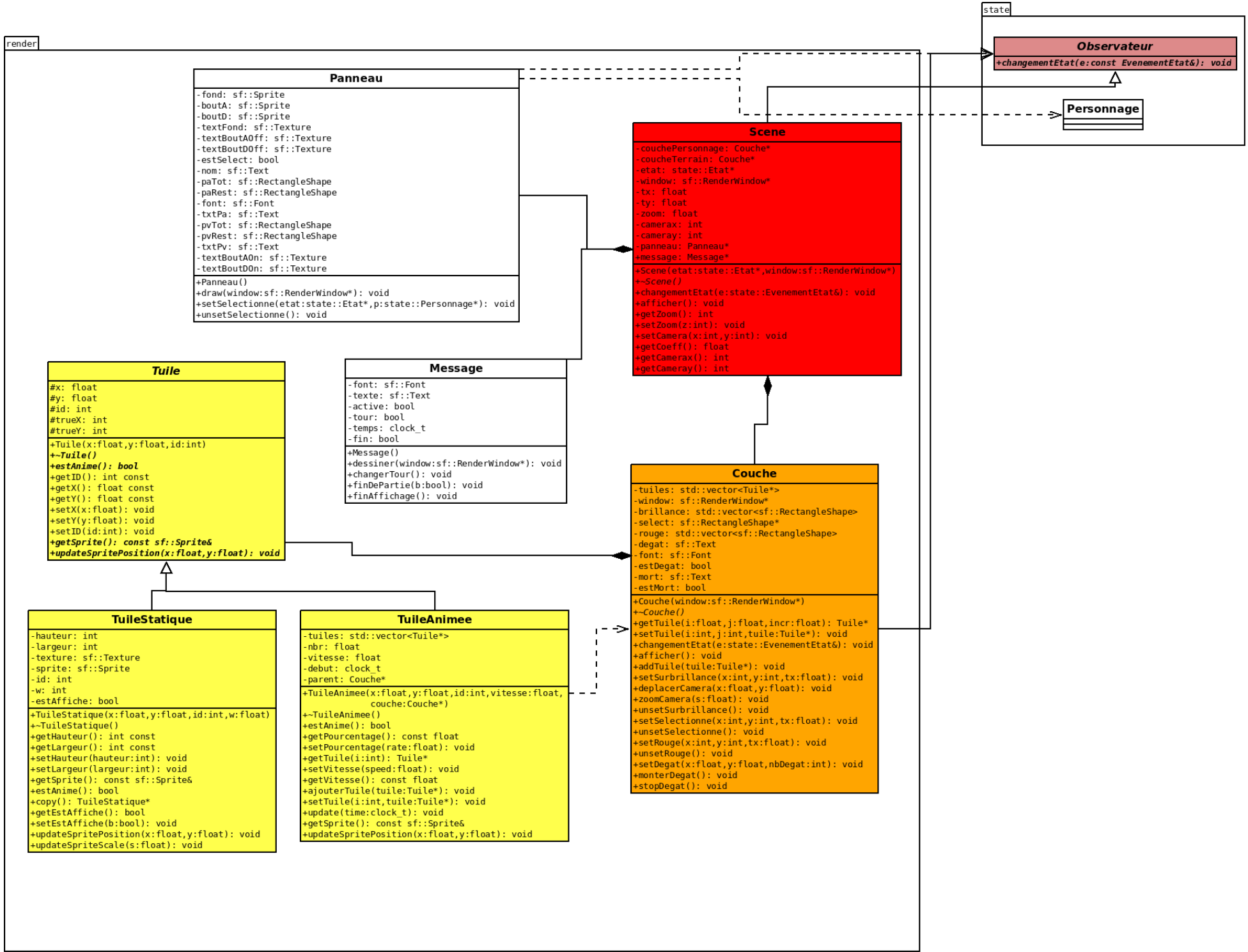
3.2 Conception logiciel

La classe **Scene** (en rouge sur le diagramme des classes) a un rôle similaire dans Render à la classe **Etat** dans State : elle joue le rôle de chef d'orchestre du rendu.

- Elle regroupe deux instances de **Couches** pour les personnages et pour le décor.
- Elle reçoit les notifications du pattern Observer pointé sur l'**Etat** du jeu, et met à jour le rendu en conséquence.
- Elle initialise la fenêtre de l'application.

La classe **Couche** (en orange) sert à stocker les coordonnées des différentes textures à afficher. Elle contient une liste de **Tuiles** et gère leurs données.

La classe **Tuile** (en jaune) est héritée par **TuileStatique** et **TuileAnimée**.
Dans **TuileStatique**, les textures sont chargées et associées à une id précise. Dans **TuileAnimée**, une animation est définie par une liste de **TuileStatiques** et une vitesse de rafraîchissement qui détermine à quelle vitesse une animation passe d'une image à une autre.


state
Observateur
+changementEtat(e:const EvenementEtat&): void
Personnage
render
Panneau
-fond: sf::Sprite
-boutA: sf::Sprite
-boutD: sf::Sprite
-textFond: sf::Texture
-textBoutAOff: sf::Texture
-textBoutDOff: sf::Texture
-estSelect: bool
-nom: sf::Text
-paTot: sf::RectangleShape
-paRest: sf::RectangleShape
-font: sf::Font
-txtPa: sf::Text
-pvTot: sf::RectangleShape
-pvRest: sf::RectangleShape
-txtPv: sf::Text
-textBoutAOn: sf::Texture
-textBoutDOn: sf::Texture
+Panneau()
+draw(window:sf::RenderWindow*): void
+setSelectionne(etat:state::Etat*,p:state::Personnage*): void
+unsetSelectionne(): void
Scene
-couchePersonnage: Couche*
-coucheTerrain: Couche*
-etat: state::Etat*
-window: sf::RenderWindow*
-tx: float
-ty: float
-zoom: float
-camerax: int
-cameray: int
-panneau: Panneau*
+message: Message*
+Scene(etat:state::Etat*,window:sf::RenderWindow*)
+~Scene()
+changementEtat(e:state::EvenementEtat&): void
+afficher(): void
+getZoom(): int
+setZoom(z:int): void
+setCamera(x:int,y:int): void
+getCoeff(): float
+getCamerax(): int
+getCameray(): int
Tuile
#x: float
#y: float
#id: int
#trueX: int
#trueY: int
+Tuile(x:float,y:float,id:int)
+~Tuile()
+estAnime(): bool
+getID(): int const
+getX(): float const
+getY(): float const
+setX(x:float): void
+setY(y:float): void
+setID(id:int): void
+getSprite(): const sf::Sprite&
+updateSpritePosition(x:float,y:float): void
Message
-font: sf::Font
-texte: sf::Text
-active: bool
-tour: bool
-temps: clock_t
-fin: bool
+Message()
+dessiner(window:sf::RenderWindow*): void
+changerTour(): void
+finDePartie(b:bool): void
+finAffichage(): void
Couche
-tuiles: std::vector<Tuile*>
-window: sf::RenderWindow*
-brillance: std::vector<sf::RectangleShape>
-select: sf::RectangleShape*
-rouge: std::vector<sf::RectangleShape>
-degat: sf::Text
-font: sf::Font
-estDegat: bool
-mort: sf::Text
-estMort: bool
+Couche(window:sf::RenderWindow*)
+~Couche()
+getTuile(i:float,j:float,incr:float): Tuile*
+setTuile(i:int,j:int,tuile:Tuile*): void
+changementEtat(e:state::EvenementEtat&): void
+afficher(): void
+addTuile(tuile:Tuile*): void
+setSurbrillance(x:int,y:int,tx:float): void
+deplacerCamera(x:float,y:float): void
+zoomCamera(s:float): void
+unsetSurbrillance(): void
+setSelectionne(x:int,y:int,tx:float): void
+unsetSelectionne(): void
+setRouge(x:int,y:int,tx:float): void
+unsetRouge(): void
+setDegat(x:float,y:float,nbDegat:int): void
+monterDegat(): void
+stopDegat(): void
TuileStatique
-hauteur: int
-largeur: int
-texture: sf::Texture
-sprite: sf::Sprite
-id: int
-w: int
-estAffiche: bool
+TuileStatique(x:float,y:float,id:int,w:float)
+~TuileStatique()
+getHauteur(): int const
+getLargeur(): int const
+setHauteur(hauteur:int): void
+setLargeur(largeur:int): void
+getSprite(): const sf::Sprite&
+estAnime(): bool
+copy(): TuileStatique*
+getEstAffiche(): bool
+setEstAffiche(b:bool): void
+updateSpritePosition(x:float,y:float): void
+updateSpriteScale(s:float): void
TuileAnimee
-tuiles: std::vector<Tuile*>
-nbr: float
-vitesse: float
-debut: clock_t
-parent: Couche*
+TuileAnimee(x:float,y:float,id:int,vitesse:float,
couche:Couche*)
+~TuileAnimee()
+estAnime(): bool
+getPourcentage(): const float
+setPourcentage(rate:float): void
+getTuile(i:int): Tuile*
+setVitesse(speed:float): void
+getVitesse(): const float
+ajouterTuile(tuile:Tuile*): void
+setTuile(i:int,tuile:Tuile*): void
+update(time:clock_t): void
+getSprite(): const sf::Sprite&
+updateSpritePosition(x:float,y:float): void

4 Description et conception du moteur de jeu

Le moteur du jeu est destiné à enregistrer les actions qu'effectue le joueur via les périphériques usuels (claviers et souris), vérifier lesquelles sont réalisables et envoyer ces dernières à l'Etat.

4.1 Description des changements d'état

L'utilisateur peut interagir avec le jeu de 4 manières différentes :

a) **Cliquer avec la souris sur un personnage sur l'écran pour le sélectionner, ou bien cliquer sur une case vide pour le désélectionner** . Cette interaction modifie l'Etat qui enregistre quel est le personnage courant sélectionné.

b) Si un personnage est sélectionné, **cliquer avec la souris sur les boutons de l'interface graphique (bouton bleu « déplacement » et rouge « attaque ») pour changer le mode d'action du personnage.** Un personnage passé en mode attaque peut **attaquer les cibles à portée en cliquant dessus**, tandis qu'un personnage en mode déplacement peut **se déplacer vers l'endroit cliqué**. Les points de vie ou de déplacement des personnages concernés sont mis à jour dans l'État (à venir).

c) **Appuyer sur les touches directionnelles du clavier déplace la caméra en conséquence.** Cela n'a qu'un impact esthétique et influence directement le package Render.

d) **Actionner la molette de la souris modifie le zoom du rendu**. De même, cette action ne touche que le rendu.

4.2 Description logicielle

Ci-dessous se trouve le diagramme des classes du package Engine.

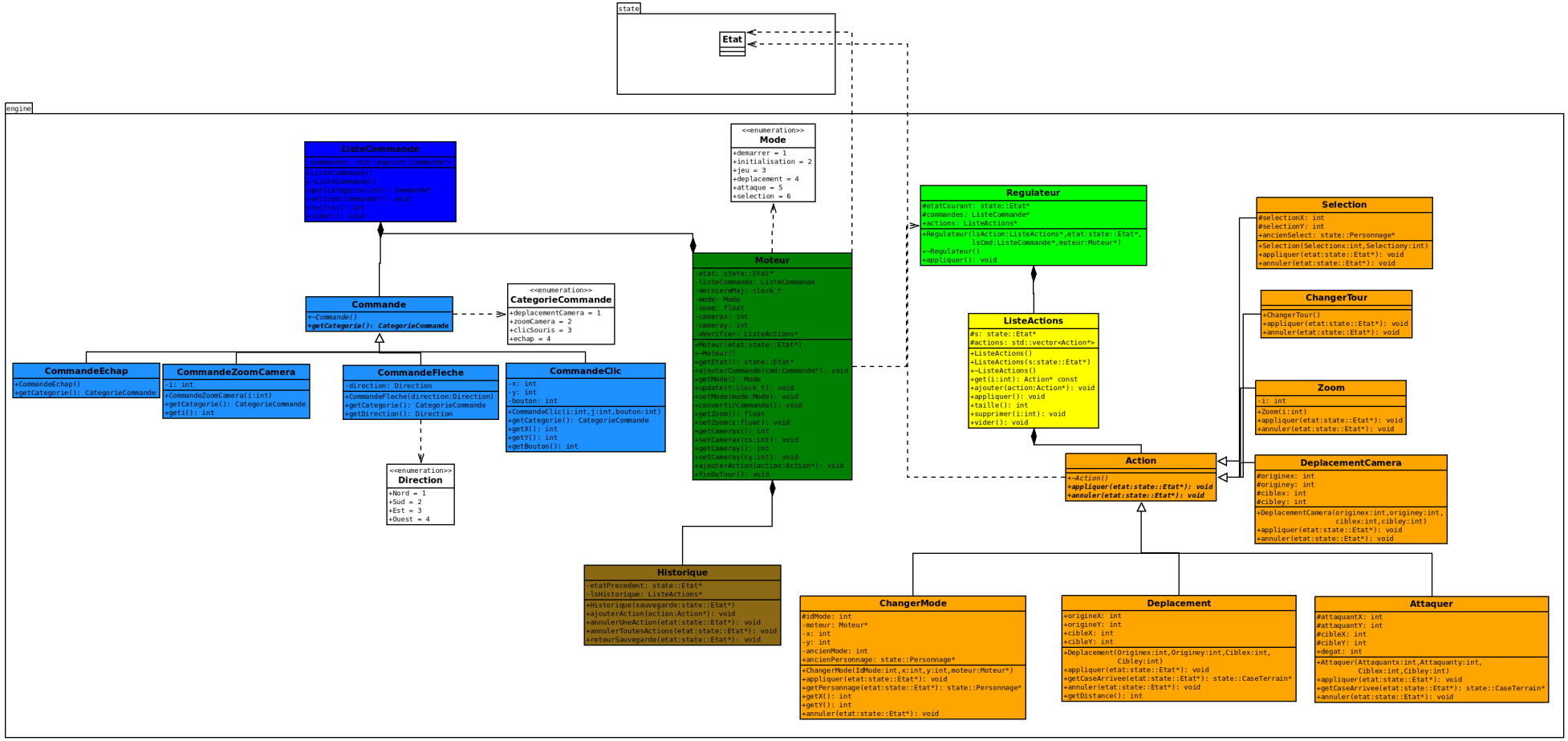
Le package Engine est principalement bâti autour d'un pattern Command :
La classe Commande et ses héritiers représente une commande extérieure : une touche clavier appuyée ou un clique de la souris. Nous facilitons le traitement des différents types de commandes grâce à un identifiant.

La classe Moteur contient une liste de Commande et convertit cette liste de commandes en une listes d'actions potentiellement réalisable. La conversion est programmée pour se faire avec un intervalle d'au moins 0,2 seconde avec la dernière conversion à avoir eu lieu.

La liste d'actions créée régulièrement par Moteur est utilisée par la classe Régulateur qui traite les actions et vérifie celles qui sont réalisable des autres.

Une fois que Régulateur a décidé des actions réalisable, Moteur les exécute.

La classe Action et ses héritiers implémentent les modifications directes de l'État.


state
Etat
engine
ListeCommande
-commandes: std::map<int,Commande*>
+ListeCommande()
+~ListeCommande()
+get(categorie:int): Commande*
+set(cmd:Commande*): void
+taille(): int
+vider(): void
<<enumeration>>
Mode
+demarrer = 1
+initialisation = 2
+jeu = 3
+deplacement = 4
+attaque = 5
+selection = 6
Regulateur
#etatCourant: state::Etat*
#commandes: ListeCommande*
#actions: ListeActions*
+Regulateur(lsAction:ListeActions*,etat:state::Etat*,
lsCmd:ListeCommande*,moteur:Moteur*)
+~Regulateur()
+appliquer(): void
Selection
#selectionX: int
#selectionY: int
+ancienSelect: state::Personnage*
+Selection(Selectionx:int,Selectiony:int)
+appliquer(etat:state::Etat*): void
+annuler(etat:state::Etat*): void
Commande
+~Commande()
+getCategorie(): CategorieCommande
<<enumeration>>
CategorieCommande
+deplacementCamera = 1
+zoomCamera = 2
+clicSouris = 3
+echap = 4
Moteur
-etat: state::Etat*
-listeCommande: ListeCommande
-derniereMaj: clock_t
-mode: Mode
-zoom: float
-camerax: int
-cameray: int
-aVerifier: ListeActions*
+Moteur(etat:state::Etat*)
+~Moteur()
+getEtat(): state::Etat*
+ajouterCommande(cmd:Commande*): void
+getMode(): Mode
+update(t:clock_t): void
+setMode(mode:Mode): void
+convertirCommande(): void
+getZoom(): float
+setZoom(z:float): void
+getCamerax(): int
+setCamerax(cx:int): void
+getCameray(): int
+setCameray(cy:int): void
+ajouterAction(action:Action*): void
+finDuTour(): void
ListeActions
#s: state::Etat*
#actions: std::vector<Action*>
+ListeActions()
+ListeActions(s:state::Etat*)
+~ListeActions()
+get(i:int): Action* const
+ajouter(action:Action*): void
+appliquer(): void
+taille(): int
+supprimer(i:int): void
+vider(): void
ChangerTour
+ChangerTour()
+appliquer(etat:state::Etat*): void
+annuler(etat:state::Etat*): void
Zoom
-i: int
+Zoom(i:int)
+appliquer(etat:state::Etat*): void
+annuler(etat:state::Etat*): void
CommandeEchap
+CommandeEchap()
+getCategorie(): CategorieCommande
CommandeZoomCamera
-i: int
+CommandeZoomCamera(i:int)
+getCategorie(): CategorieCommande
+geti(): int
CommandeFleche
-direction: Direction
+CommandeFleche(direction:Direction)
+getCategorie(): CategorieCommande
+getDirection(): Direction
CommandeClic
-x: int
-y: int
-bouton: int
+CommandeClic(i:int,j:int,bouton:int)
+getCategorie(): CategorieCommande
+getX(): int
+getY(): int
+getBouton(): int
<<enumeration>>
Direction
+Nord = 1
+Sud = 2
+Est = 3
+Ouest = 4
Action
+~Action()
+appliquer(etat:state::Etat*): void
+annuler(etat:state::Etat*): void
DeplacementCamera
#originex: int
#origineY: int
#ciblex: int
#cibley: int
+DeplacementCamera(originex:int,origineY:int,
ciblex:int,cibley:int)
+appliquer(etat:state::Etat*): void
+annuler(etat:state::Etat*): void
Historique
-etatPrecedent: state::Etat*
-lsHistorique: ListeActions*
+Historique(sauvegarde:state::Etat*)
+ajouterAction(action:Action*): void
+annulerUneAction(etat:state::Etat*): void
+annulerToutesActions(etat:state::Etat*): void
+retourSauvegarde(etat:state::Etat*): void
ChangerMode
#idMode: int
-moteur: Moteur*
-x: int
-y: int
-ancienMode: int
+ancienPersonnage: state::Personnage*
+ChangerMode(IdMode:int,x:int,y:int,moteur:Moteur*)
+appliquer(etat:state::Etat*): void
+getPersonnage(etat:state::Etat*): state::Personnage*
+getX(): int
+getY(): int
+annuler(etat:state::Etat*): void
Deplacement
+origineX: int
+origineY: int
+cibleX: int
+cibleY: int
+Deplacement(Originex:int,Origineyint,Ciblex:int,
Cibley:int)
+appliquer(etat:state::Etat*): void
+getCaseArrivee(etat:state::Etat*): state::CaseTerrain*
+annuler(etat:state::Etat*): void
+getDistance(): int
Attaquer
#attaquantX: int
#attaquantY: int
#cibleX: int
#cibleY: int
+degat: int
+Attaquer(Attaquantx:int,Attaquanty:int,
Ciblex:int,Cibley:int)
+appliquer(etat:state::Etat*): void
+getCaseArrivee(etat:state::Etat*): state::CaseTerrain*
+annuler(etat:state::Etat*): void

5 Intelligence Artificielle

5.1 IA : Stratégie

Nous créons le package AI qui ne possède qu'une classe, IA, qui applique les changements d'état successivement du joueur bleu puis du joueur rouge, via des commandes générées dans le package Engine. Il y a trois niveaux différents d'Intelligence Artificielle dans notre jeu, du plus simpliste (aléatoire) au plus sophistiqué (MinMax)

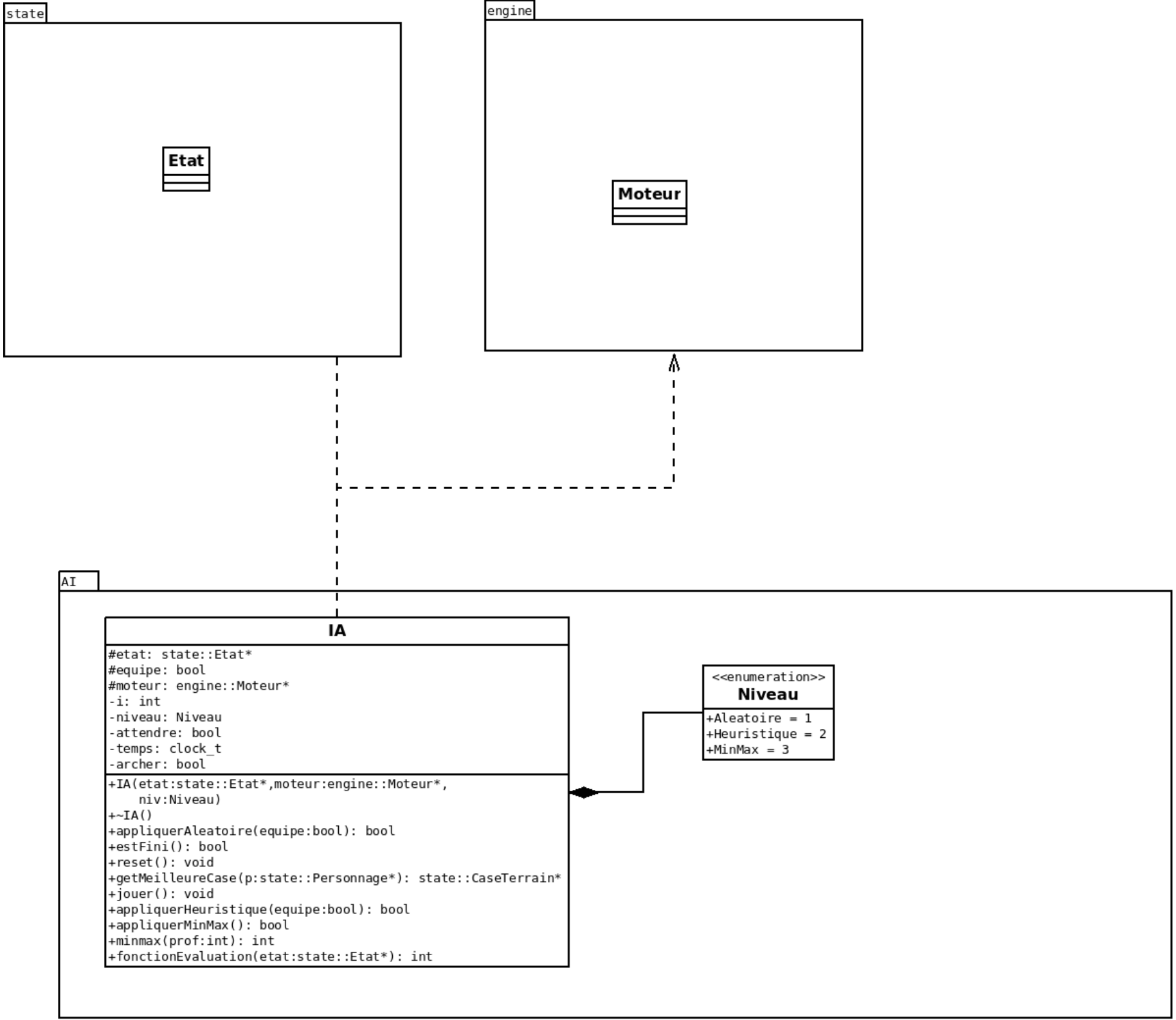
Le comportement de l'IA de niveau aléatoire est :
_ Pour tout personnage possédé par l'IA (rouge ou bleu), nous vérifions s'il peut attaquer un ennemi. Si oui, il l'attaque à volonté (s'il y a plusieurs ennemis, il attaque l'un d'entre eux au hasard). Sinon, le personnage se déplace sur une case atteignable aléatoire, c'est-à-dire l'une des cases où le personnage peut effectivement se rendre ce tour-ci.
_ Une fois que tous les personnages d'une IA ont agi, la main passe à l'autre IA qui peut à son tour agir.

Le comportement de l'IA de niveau heuristique est le suivant :
_ Pour tout personnage possédé par l'IA (rouge ou bleu), nous vérifions s'il peut attaquer un ennemi. Sinon, le personnage se déplace sur une case atteignable précise.
_ Le calcul de la case optimale est effectué comme suit : s'il existe une case à partir de laquelle il peut ou pourra attaquer au prochain tour, il s'y déplace. Sinon, il se déplace sur la case le rapprochant le plus possible du personnage ennemi le plus proche.
_ Une fois que tous les personnages d'une IA ont agi, la main passe à l'autre IA qui peut à son tour agir.

Le comportement de l'IA de niveau MinMax est le suivant :
_ Pour tout personnage possédé par l'IA (rouge ou bleu), nous vérifions (par simulation via un enregistrement dans l'engine) toutes les actions qu'il peut effectuer potentiellement.

A la fin d'une action donnée, l'algorithme réévalue la partie pour l'IA courante et donne un coefficient pour juger de la situation : plus ce coefficient est élevé, plus la situation est favorable à l'IA qui joue. Le coefficient est notamment calculé avec la quantité de personnages alliés encore présents en jeu (ils augmentent le coefficient) et la quantité de personnages ennemis (ils diminuent le coefficient). Nous calculons ainsi différents coefficients pour les différentes actions à tester.
Quand toutes les actions possibles ont été parcourues, celle associée au coefficient le plus élevé est choisie et sera appliquée.

_ Une fois que tous les personnages d'une IA ont agi, la main passe à l'autre IA qui peut à son tour agir.


state
Etat
engine
Moteur
AI
IA
#etat: state::Etat*
#equipe: bool
#moteur: engine::Moteur*
-i: int
-niveau: Niveau
-attendre: bool
-temps: clock_t
-archer: bool
+IA(etat:state::Etat*,moteur:engine::Moteur*,
niv:Niveau)
+~IA()
+appliquerAleatoire(equipe:bool): bool
+estFini(): bool
+reset(): void
+getMeilleureCase(p:state::Personnage*): state::CaseTerrain*
+jouer(): void
+appliquerHeuristique(equipe:bool): bool
+appliquerMinMax(): bool
+minmax(prof:int): int
+fonctionEvaluation(etat:state::Etat*): int
<<enumeration>>
Niveau
+Aleatoire = 1
+Heuristique = 2
+MinMax = 3

6 Définition des API Web

Nous définissons une API Web via les requêtes GET et PUT ci-dessous. Elles permettent l'échange de commandes entre le serveur et chaque joueur. Ces requêtes sont destinées à être envoyées régulièrement pour juger des modifications de l'état (GET) ou demander à le modifier (PUT).

Requête : GET /commande/<id>

Réponse :

- S'il y a des commandes en attente :

Status : 200 (HTTP OK)
Données :
type: object
properties:

```
{
        "type" : {type: string},
        "joueur" : {type: boolean},
        "ancx" : {type: number, minimum: 0, maximum: 23},
        "ancy" : {type: number, minimum: 0, maximum: 23},
        "newx" : {type: number, minimum: 0, maximum: 23},
        "newy" : {type: number, minimum: 0, maximum: 23},
        "degat" : {type: number},
        "sauvEquipe" : {type: boolean},
        "sauvType" : {type: number},
        "afficher" : {type: boolean}
}
}
```

- S'il n'y a pas de commandes en attente

Status : 404 (HTTP NOT FOUND)

Requête : PUT /commande/

Status : 200 (HTTP OK)
Données :
type: object
{
properties:
{
« epoch « : {type : number},
"type" : {type: string},
"joueur" : {type: boolean},
"ancx" : {type: number, minimum: 0, maximum: 23},
"ancy" : {type: number, minimum: 0, maximum: 23},
"newx" : {type: number, minimum: 0, maximum: 23},
"newy" : {type: number, minimum: 0, maximum: 23},
"degat" : {type: number},
"sauvEquipe" : {type: boolean},
"sauvType" : {type: number},
"afficher" : {type: boolean}
}
}
Requête : PUT/joueur/
Données :
type : « object »
{
properties
{
« equipe » : {type:boolean},
« type » : {type:string}
}
}

Requête : GET /joueurs/
Données :

type : « object » :
{
properties
{
« equipe » : {type:boolean},
« type » : {type:string}
}