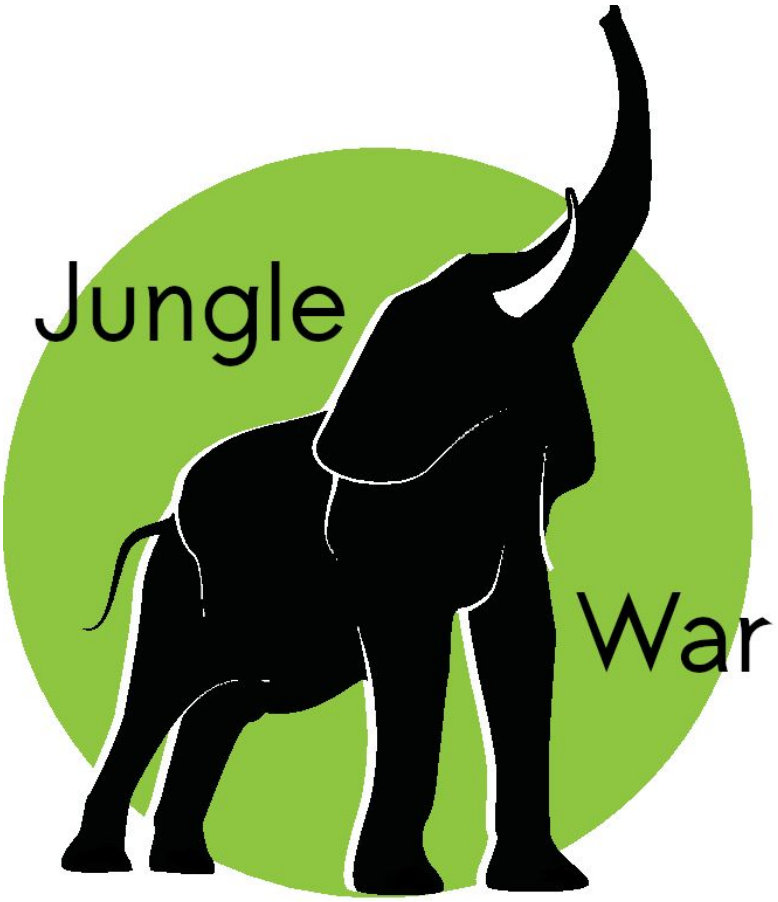
Rapport projet IS

Baptiste LANOUE et Robin SICSIC

25/09/2019 Jalon 1.1
16/10/2019 Jalon 1.final
31/10/2019 Jalon 2.1
12/11/2019 Jalon 2.2
20/11/2019 Jalon 2.final
27/11/2019 Jalon 3.1
13/12/2019 Jalon 3.final

Jungle War



Description du jeu

Archétype du jeu : Stratego

Stratego est un jeu stratégique où 2 joueurs s'affrontent et dont les mécanismes de jeu sont essentiellement le bluff et l'affrontement.
Le but du jeu est d'atteindre le drapeau adverse. Pour cet objectif, les joueurs disposent de 40 pièces qui ont des valeurs de puissances différentes et croissantes. Ces pièces sont face cachés. Pour découvrir la valeur d'une pièce, le combat est nécessaire.

Stratego est inspiré d'un ancien jeu chinois "Jeu du combat des animaux" qui est plus simple que le Stratego et que nous allons remettre au goût du jour.

Jeu du combat des animaux

Ce jeu se joue avec 8 pièces par joueur qui sont placées sur les carreaux. Les 2 camps sont le blanc (ou le rouge) et le noir (ou le bleu).

Chaque joueur dispose de huit animaux différents en tant que pièce de jeu. Les animaux ont leur propre valeur de puissance et certains ont des déplacements spécifiques.
Chaque pièce peut capturer une pièce adverse de rang égal ou inférieur. Cependant un rat (qui a la valeur la plus faible) peut capturer un éléphant (qui a la valeur la plus forte).

Fonctionnement du jeu

Deux joueurs s'affrontent sur un plateau avec des animaux. Il doivent prendre le contrôle de la tanière de l'adversaire ou capturer toutes les pièces de l'adversaire Chaque pièce a un score de puissance qui permet de manger les autres pièce inférieures ou égale et peut avoir des mouvements spéciaux. Les pièces ne sont pas face cachés.

Liste des pièces (classe Animal)

Ci-joint le tableau, des pièces du jeu avec la valeur de puissance et spécialité.

Pièce	Puissance	Spécialité
ELEPHANT	8	Perd contre le RAT uniquement si le RAT attaque par une case EARTH
TIGER	7	Si TIGER est sur une case SHORE, il peut se déplacer vers la case SHORE associé en face de sa case initiale.. Cependant si le RAT allié ou ennemis est sur sa trajectoire via une case WATER, TIGER ne peut pas se déplacer.
LION	6	Si LION est sur une case SHORE, il peut se déplacer vers la case SHORE associé en face de sa case initiale.. Cependant si le RAT allié ou ennemis est sur sa trajectoire via une case WATER, LION ne peut pas se déplacer.
LEOPARD	5	Si LEOPARD est sur une case SHORE, il peut se déplacer vers la case SHORE associé en face de sa case initiale.. Cependant si le RAT allié ou ennemis est sur sa trajectoire via une case WATER, LEOPARD ne peut pas se déplacer.
DOG	4	Pas de spécialité.
WOLF	3	Pas de spécialité.
CAT	2	Pas de spécialité.
RAT	1	Peut marcher dans les cases WATER Tue ELEPHANT uniquement si il se déplace depuis une case EARTH.

Priorité à l'attaquant : Si une pièce avance sur une case dont l'occupant est une pièce adverse de même valeur. C'est l'attaquant qui remporte.

Chaque joueur a un animal de chaque type, rangé par puissance croissante dans un dictionnaire de type "unordered_map".

Liste des cases (classe Square)

Type de Case (énumération)	Spécialité
EARTH	Tout animal peut avancer sur une case EARTH.
WATER	Le RAT peut se déplacer dans les cases WATER. TIGER, LION, LEOPARD peuvent sauter horizontalement et verticalement haut dessus des cases WATER depuis une case SHORE si le RAT allié ou ennemis n'est pas sur la trajectoire.
SHORE	TIGER, LION, LEOPARD peuvent sauter horizontalement et verticalement haut dessus des cases WATER depuis une case SHORE si le RAT allié ou ennemis n'est pas sur la trajectoire.
TRAPJ1	Si un animal de J2 est sur la case TRAPJ1 de l'opposant , il passe dans l'état TRAPPED (voir description des états au jalon 1.final) , ce qui signifie que sa puissance est réduite à 0.
TRAPJ2	Si un animal de J1 est sur la case TRAPJ2 de l'opposant , l'animal passe dans l'état TRAPPED, ce qui signifie que sa puissance est réduite à 0.
THRONEJ1	Si un animal de J2 est sur la case THRONEJ1 de l'opposant, l'animal passe dans l'état VICTORIOUS et le joueur qui possède l'animal gagne la partie.
THRONEJ2	Si un animal de J1 est sur la case THRONEJ2 de l'opposant, l'animal passe dans l'état VICTORIOUS et le joueur qui possède l'animal gagne la partie.

Ressources

Ressource du terrain : plateau de la jungle

D'après l'auteur, utilisation et modification libre.

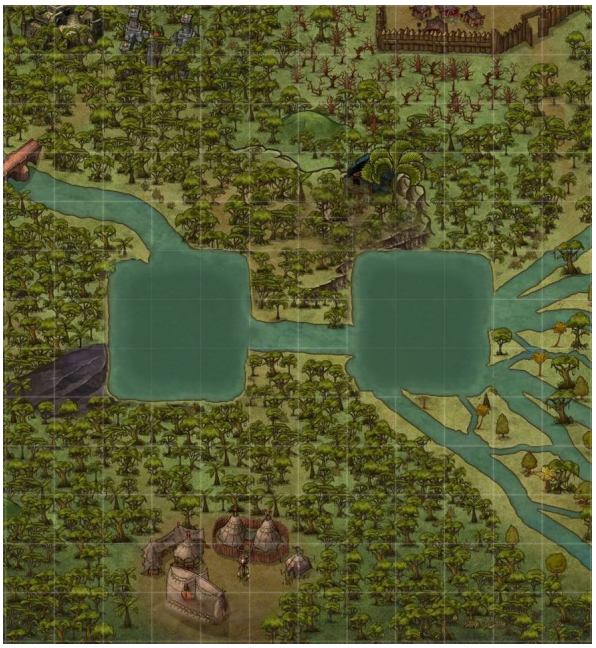


Figure 1 : Plateau de jeu

Version utilisée pour la programmation avec les coordonnées :

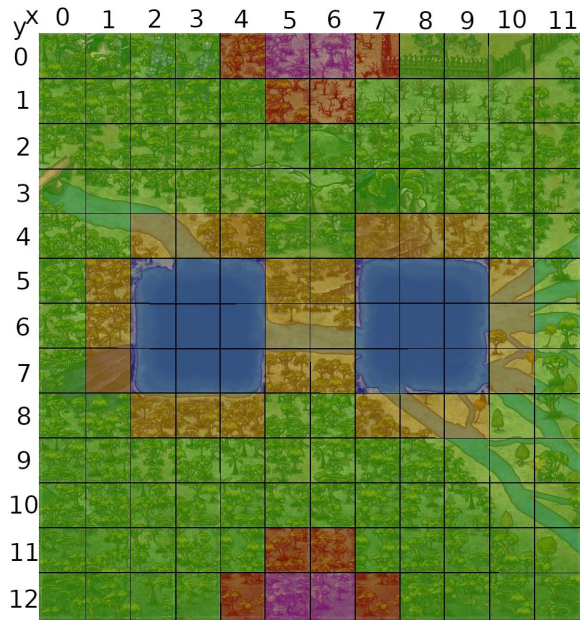


Figure 2 : Plateau de jeu version programmation

Vert : EARTH
Orange : SHORE
Bleu : WATER
Rouge : TRAP (J1 ou J2)
Rose : THRONE (J1 ou J2)

Ressources des pièces : animaux

Pour le choix des animaux, nous avions décidé de récupérer des création un peu partout sur internet. Nous les avons retravaillés pour qu'elle soient plus cohérentes en elles et dans le même thème. Elles sont en HD mais ne sont cependant pas libres de droits. Nous avons donc opté pour de nouvelles ressources, cette fois-ci libre de droit en utilisation et modification. Nous avons trouvé les ressources mais elle sont en cours de modification pour être adaptée au jeu. Elle sont stockée dans le rossier *res*.

Environnement de développement

Changement d'environnement depuis le dernier rapport. Nous sommes tous les deux passé sur un linux ubuntu 18.04 en dual boot.

Une clé SSH a été générée avec Gitkraken et enregistrée sur github. On peut ainsi administrer le git depuis ce logiciel et régler les conflits.
On peut voir ici quelques modifications (push) du programme main avec le "hello world".

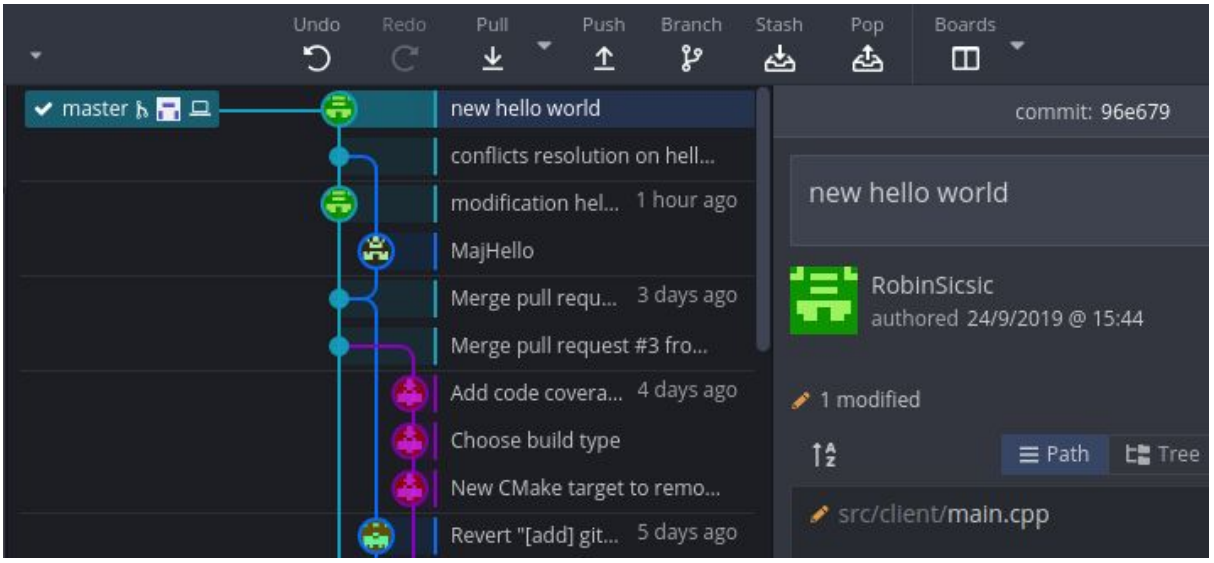


Figure 3 : Capture d'écran de GitKraken

Pour l'édition de code nous utilisons Atom et pour le diagramme de classe nous utilisons Dia. Pour les diagrammes d'état et de pseudo-algorithme, nous utilisons le site draw.io relié à google drive.

Description et Conception des états

Description des classes

State

Un état du jeu (State) est formée par une grille (grid) de cases (Square) ainsi que deux joueurs (player1 et player2). L'attribut *turn* permet de savoir à quel tour de jeu en est la partie. Il y a les attributs game over et winner qui renvoie un 1 en cas de fin du jeu et le joueur gagnant. Finalement, l'attribut highlight qui est une liste de pair de Coord-IDaction qui donne des informations au sujet des actions possibles à ces coordonnées utilisé dans l'engine et dans le rendu. La grille est un vecteur de vecteurs de cases (Square), il représente le terrain. Elle est ainsi initialisée par le constructeur State() et State(string nom1, string nom2) de la classe State. Elle ne sera pas modifiée au cours du jeu.
Le state possède des méthodes permettant de renvoyer un Square de la grid en fonction d'un objet Coord (getSquare) et de renvoyer une paire d'animal et sa couleur aussi en fonction d'un objet Coord (getSelection). Ces méthodes seront essentielles pour le fonctionnement de l'Engine qui imposer les règles du jeu.
Le state contrôle aussi le menu avec un attribut propre et une enumeration de menu.

Player

Les joueurs possèdent eux même une liste d'animaux (animals), un nom, une couleur. Le constructeur de Player va initialiser la position des animaux sur la map.

La liste d'animaux de chaque joueur est un vector.

Coord

Classe qui permet d'organiser un système de coordonnée avec un attribut X et Y. Chaque animal aura son propre attribut Coord

Animal

Chaque animal a un statut variable (AnimalStatus) compris entre 0 et 1, ainsi que des coordonnées (x et y) sur le plateau.

Chaque animal a un id pour déterminer son type de 0 à 7 et donc sa puissance.
Description des états des animaux:

Etat	Description
NORMAL	L'animal est dans son état normal, sur une case EARTH, vivant et de puissance non modifiée.
DEAD	L'animal est mort, n'est plus représenté sur le plateau de jeu mais toujours présent dans la liste des animaux du joueur.

Square

Chaque case du plateau a un identifiant “id” invariable (type SquareID) qui représente son type (EARTH, WATER, etc.) ainsi qu’une caractéristique booléenne d’être occupée ou non. On connaît les coordonnées d’une case de part sa position dans le vecteur “grid”.

Observeur

L’observateur de State notifie le rendu en cas de changement, avec un code d'événement pour que le rendu n’ai pas à tout redessiner, seulement ce qui a changé.
Les événements sont :

- ALL_CHANGED L’état doit totalement être redessiné.
- ANIMALSJ1_CHANGED Les animaux de J1 doivent être redessinés.
- ANIMALSJ2_CHANGED Les animaux de J2 doivent être redessinés.
- TURN_CHANGED Les informations liées au tour de jeu doivent être actualisées.
- HIGHLIGHT_CHANGED Les cases en surbrillances ont changées (surement suite à la sélection d’un animal) et doit être redessinées.

Diagramme de classe

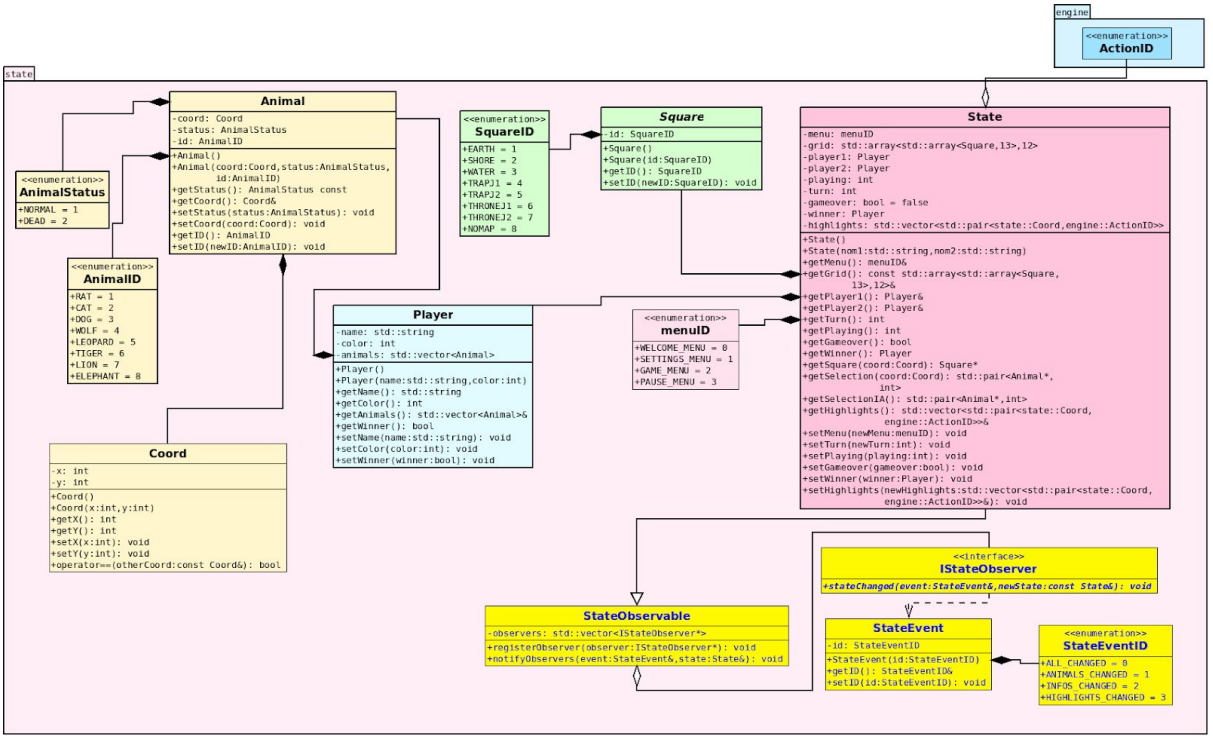


Figure 4 : Diagramme de classe de state

Description et Conception du Rendu

Stratégie de rendu d’un état

Pour le rendu d’un état, nous avons fait le choix d’un rendu par tuile à l’aide de la bibliothèque SFML.

Nous divisons la scène à afficher en couches : une couche pour le terrain, une pour les jetons et une pour le descriptif des informations des joueurs. Nous rajouterons des couches afin par exemple d'afficher les cases sélectionnées.

La grille de terrain est créée à partir d'une seule image transformée en Sprite pour simplifier l'édition.

Les jetons sont des sprites qui sont placés sur la la grille du Terrain via les coordonnées initialisées dans le constructeur de la Player: *Player::Player(string name,int color,bool playing)* (src/shared/state/Player.cpp).

Ainsi lorsqu'on applique la commande ./bin/client renderTest2, la map est initialisée avec des jetons placés grâce à la méthode de RenderLayer *draw:(sf::RenderWindow& window)* dans notre main.cpp. Il prend un objet **state** et un objet **window**, la fenetre dans laquelle on veut afficher notre jeu.

Afin de tester l'initialisation, il est possible de modifier l'emplacement des jetons dans le constructeur de Player. Ce qui modifiera le rendu à l'écran.

Conception Logiciel

Classe RenderLayer : C'est la classe principale de notre rendu et permet l'affichage des différents éléments de l'état à afficher. Elle devrait être une classe observateur lié à la classe State. Elle possède comme attribut l'état du Jeu **renderingState,** les animaux et leur sprites : **animalsSpriteJ1** et **animalsSpriteJ2,** la Window **window.** Les textures du terrain et des animaux respectivement **textureGrid** et **textureAnimals.** Un vecteur de pointeur TileSet **tileSets** qui sera utilisé plus tard car pour le moment, les images d'animaux peuvent être chargée directement en utilisant un seul fichier animalsTile.png **.**
Pour initialiser, la map on utilise *Draw:(sf::RenderWindow& window).*
Pour obtenir nos sprites à afficher à partir de la liste des animaux des joueurs, on utilise la méthode *mapToSprites(vector<Animal> animalsMap, int color).* Elle modifie l'apparence l'emplacement des sprites en fonction du statut des animaux et de leur coordonnées.
Nous avons aussi rajouté les sprites d'infos et de highlight (**spritesInfos et spritesHighLights**)

Diagramme de classes de rendu

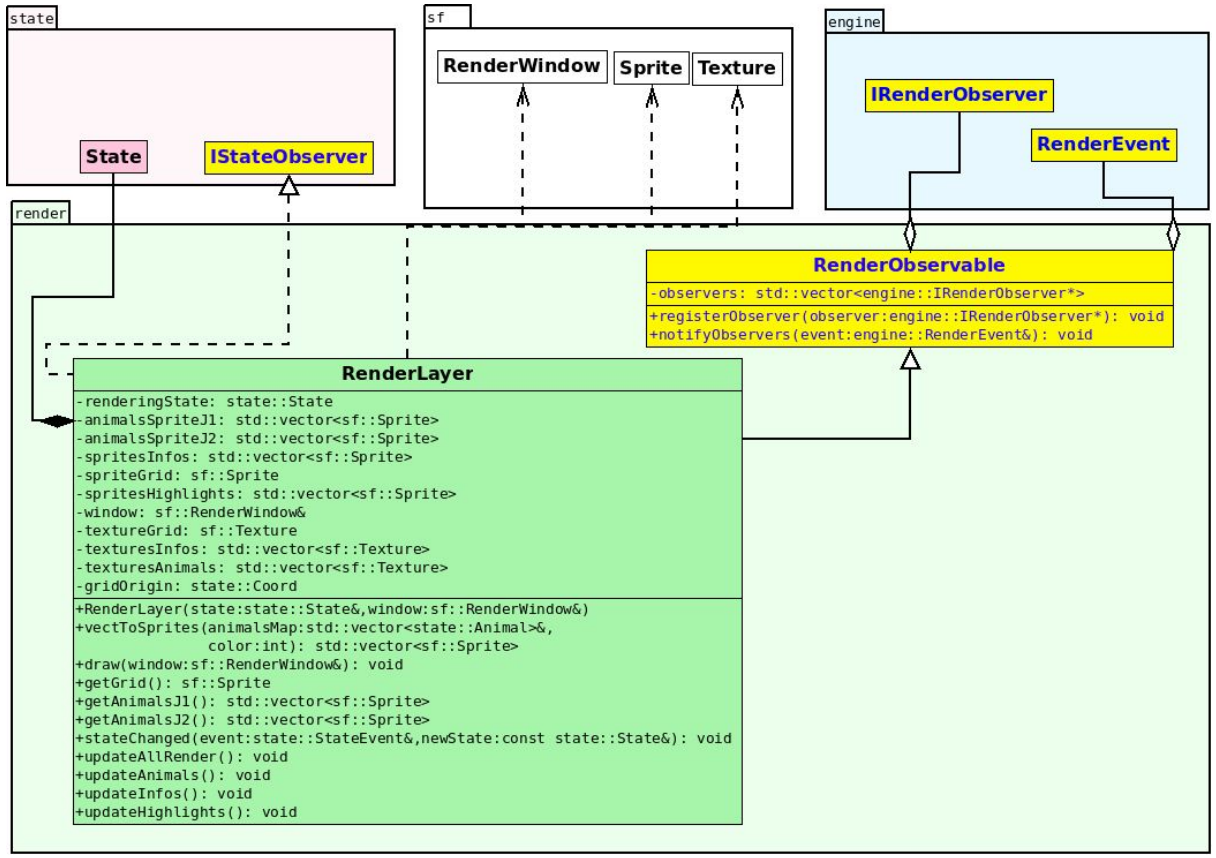


Figure 5 : Diagramme de classe de render

Règle de changement d’états et moteur de jeu

Suite à un événement provenant du rendu tel qu’un clic de souris, le moteur de jeu interprète la commande et effectue une modification de state. Une fois le state modifié, le rendu est notifié et l’affichage est actualisé.

En début de tour, le joueur peut sélectionner une pièce et puis la déplacer. La pièce peut se déplacer en fonction des règles.

Un animal ne peut être déplacé que d’une case par une case, en fonction de ses caractéristiques.

Un personnage ne peut attaquer un autre personnage que lorsque celui ci respecte les contraintes cités dans la description des règles et appartient au camp adverse (les attaques alliées ne sont pas autorisées).

Le tour de jeu d’un joueur est terminé lorsque qu’il a effectué un déplacement ou une attaque. C’est alors le tour du joueur adverse.

Lorsqu’un animal est attaqué par un ennemi, si l’animal a une valeur plus petite ou égale, il est passé dans le statut DEAD.

Si tous les animaux d’un joueur meurent, la partie est terminée et le joueur adverse gagne.

Description des classes

Classe Engine :. Elle permet de stocker les commandes dans une std::map avec clef entière (avec « addOrder »). Ce mode de stockage permet d'introduire une notion de priorité en anticipation à la version du jeu en lien avec un serveur : on traite les commandes dans l'ordre de leur clef (de la plus petite à la plus grande). Lorsque la méthode « update » est appelée, le moteur appelle la méthode « execute » de chaque commande puis supprime toutes les commandes (Order) une fois exécutées.

La méthode **authorisedActions()** va gérer les règles de notre jeu. Cette méthode va renvoyer une liste de paire<Coord,IDAction> qui correspond aux actions potentiels autour de la case ciblée en entrée. L'ensemble du codage de la méthode est expliquée dans Gestion des règles du jeu. Elle appelle de nombreuses méthodes de State notamment getSquare() et getSelect()

Classe Order :

La classe Move, héritant de la classe Order, possèdent chacune une méthode « execute » qui fait effectuer à la pièce un déplacement. Move a 2 attributs **targetAnimal** qui correspond à l'animal qui va agir et **targetCoord** qui correspond à la destination de l'animal

La méthode "execute" comme son nom indique execute l'action. Elle va utiliser la méthode authorisedAction() d'Engine afin de savoir l'IDAction du Move. En fonction de l'IDAction du Move, il y aura des modifications de l'état du jeu.

NONE : aucun impact, impossible d'execute

SHIFT : changement de coordonnée de **targetAnimal** par **targetCoord**

ATTACK : changement de coordonnée de **targetAnimal** par **targetCoord** et changement de status de NORAML en DEAD de l'animal attaqué situé en **targetCoord**

JUMP : changement de coordonnée de **targetAnimal** par **targetCoord**

SHIFT_TRAPPED : changement de coordonnée de **targetAnimal** par **targetCoord**.

SHIFT_VICTORY. :changement de coordonnée de **targetAnimal** par **targetCoord**, renvoie le joueur gagnant au State et la valeur 1 à l'attribut **Gameover** de State

La Classe Select va permettre que lorsque le joueur clique sur un pion (**targetAnimal)** les 4 cases autour de celui ci sont coloriés en fonctions des actions possibles : NONE, SHIFT, ATTACK, JUMP, SHIFT_TRAPPED et SHIFT_VICTORY.

Observeur

L'observateur de ENGINE notifie le rendu en cas de changement, avec un code d'événement pour que le rendu n'ai pas à tout redessiner, seulement ce qui a changé.
Les événements sont :

- PLAY_MOUSE_SELECT indique la méthode execute de Select

- PLAY_MOUSE_MOVE indique la méthode execute de Move

Diagramme de classe du moteur de jeu

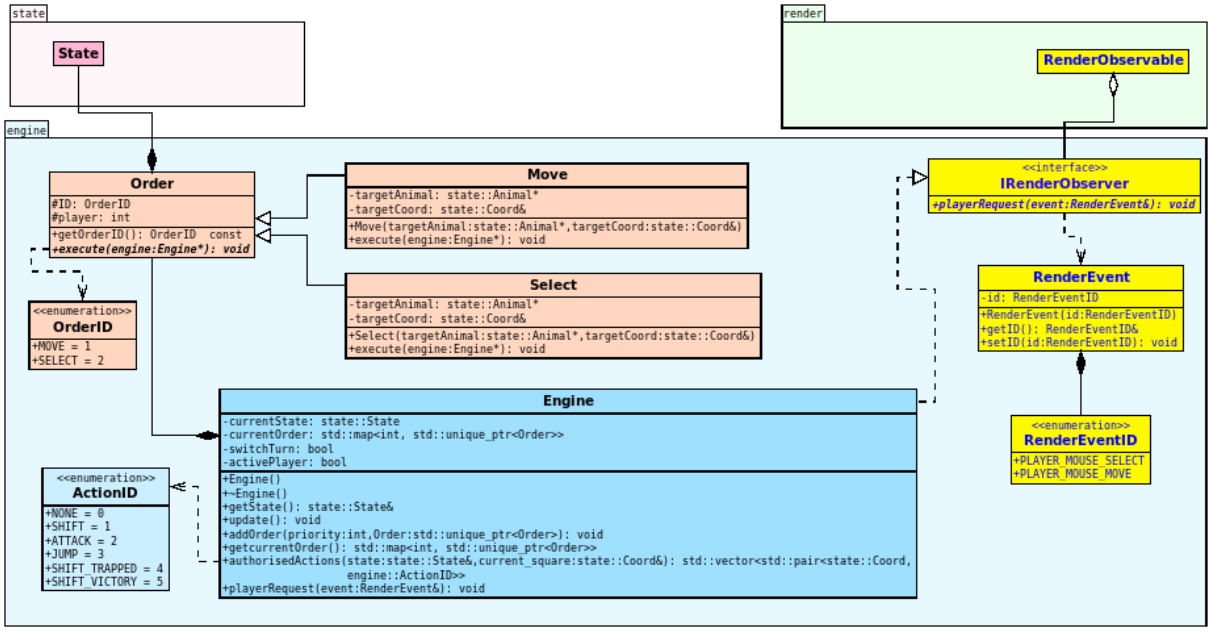


Figure 6 : Diagramme de classe de engine

Le diagramme des classes pour le moteur de jeu (« ***engine.dia*** ») est présenté ci-dessus.
Le moteur de jeu repose sur le Design Pattern Command.

Gestion des règles du jeu

Lorsque que c'est son tour, un joueur peut cliquer sur une pièce pour afficher les déplacements possibles. Suite à cela, le joueur clique sur la case pour choisir son action parmis MOVE et ATTACK qui sont des **ActionID**. Il est donc nécessaire de définir un algorithme évaluant les coups autorisés pour un animal donné. Dans un soucis d'efficacité, nous avons choisis de tester les cases adjacentes à l'animal et d'en déterminer les actions possibles, plutôt que d'évaluer chaque case du plateau indépendamment.
L'algorithme est différents en fonction des pièces sélectionnées. Il a trois schémas, un pour le RAT, un pour les pièces CAT DOG WOLF ELEPHANT et un pour les pièces LEOPARD TIGER LION.

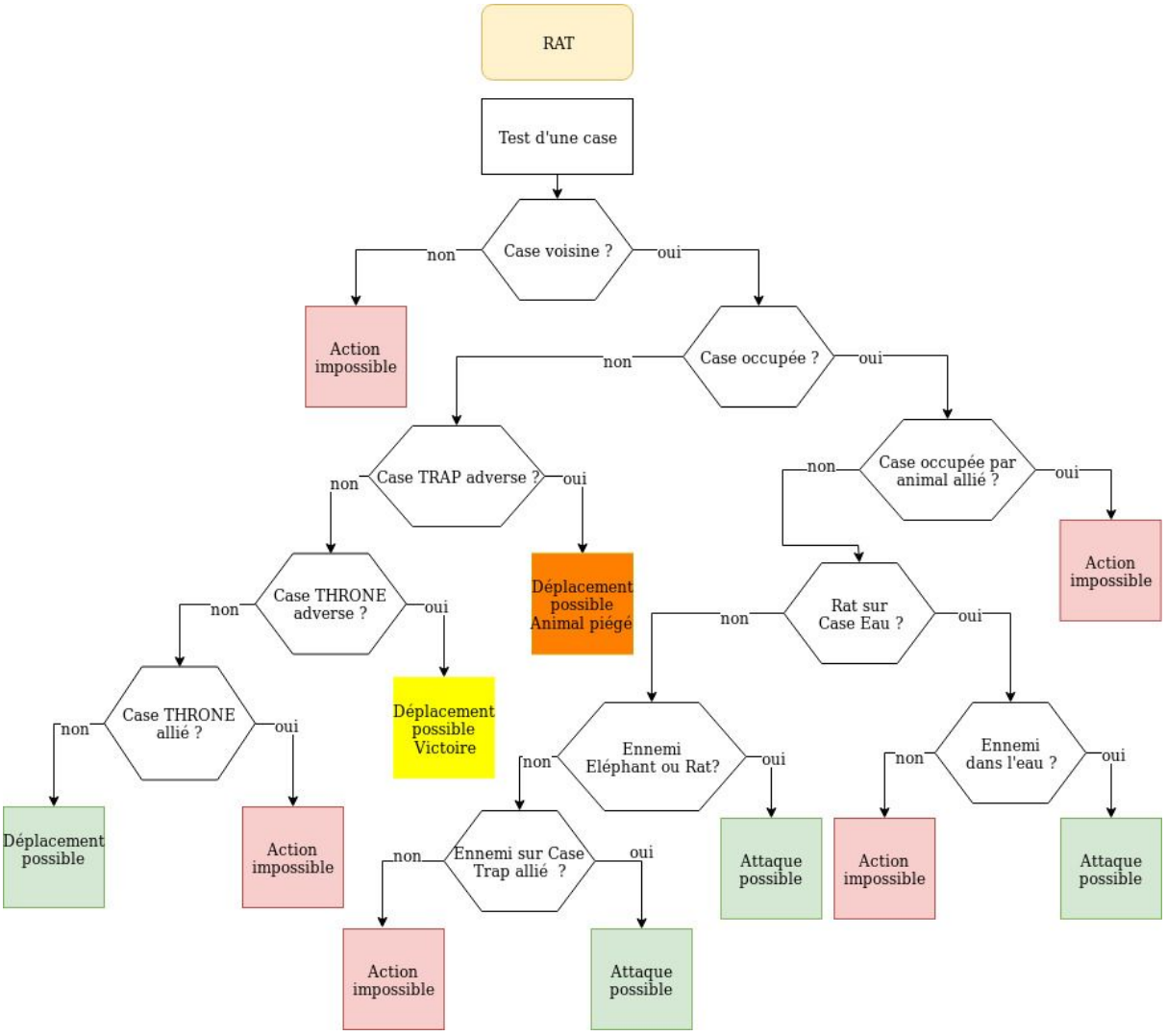


Figure 7 : Diagramme d'état de la méthode d'évaluation des coups possible authorisedActions, dans le cas d'un déplacement de RAT

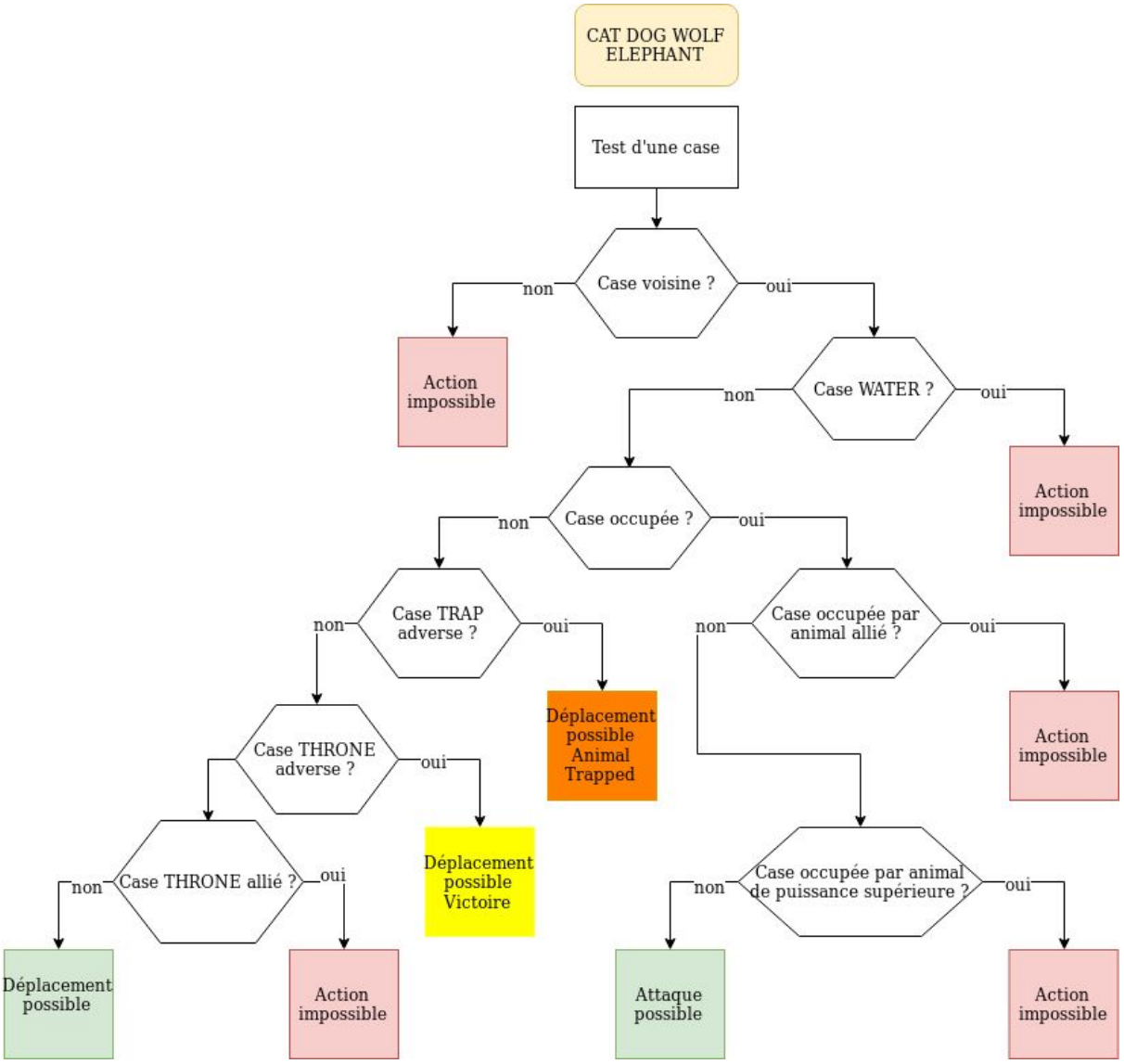


Figure 8 : Diagramme d'état de la méthode d'évaluation des coups possible authorisedActions, dans le cas d'un déplacement de CAT, DOG, WOLF ou ELEPHANT

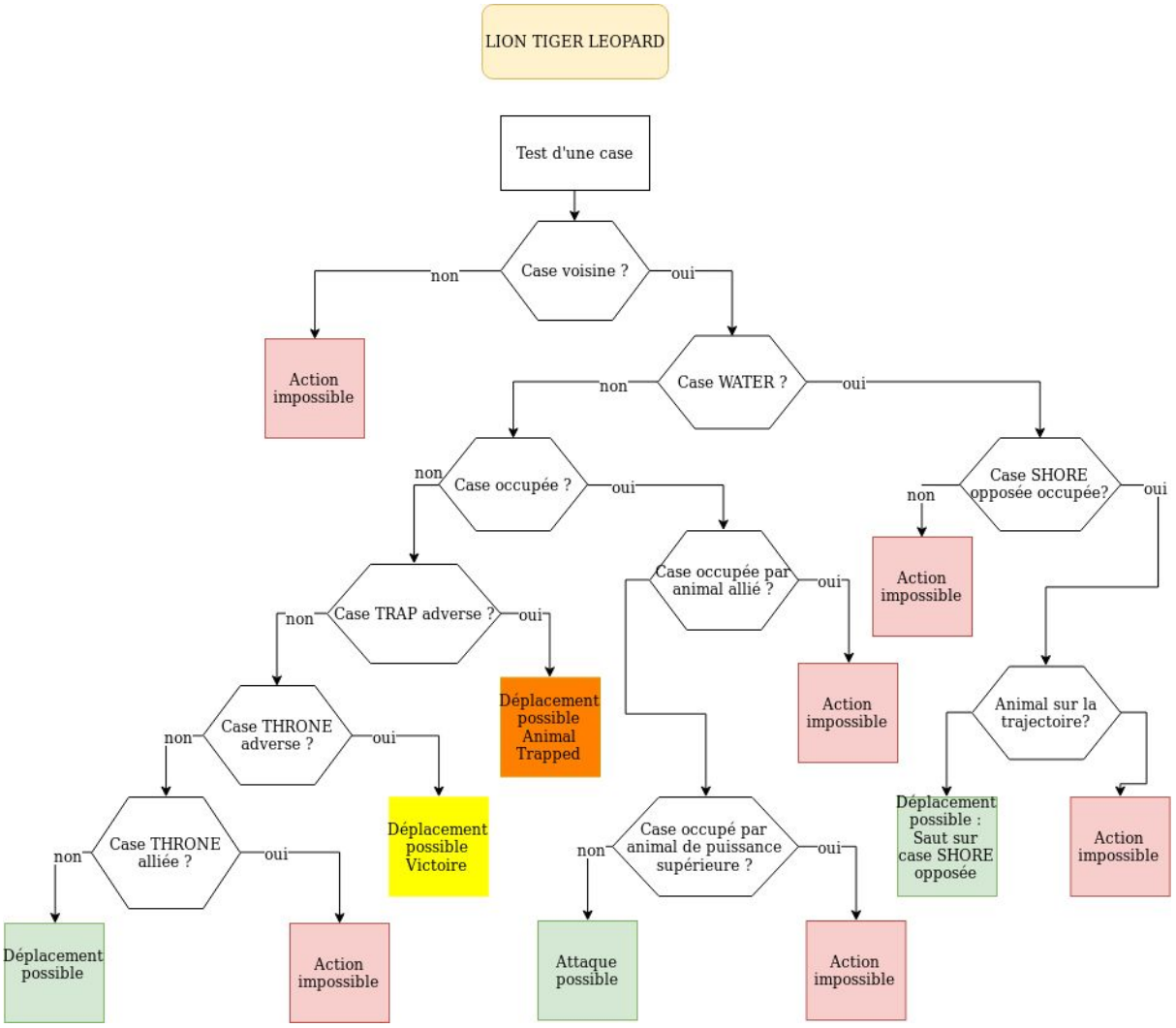


Figure 9 : Diagramme d'état de la méthode d'évaluation des coups possible authorisedActions, dans le cas d'un déplacement de LION, TIGER et LEOPARD

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence Aléatoire

L'IA contrôle une liste de pion. Il peut en déplacer un par tour. Il choisi un pion au hasard et choisi au hasard parmi les actions proposés par le moteur du jeu. Le tour se termine lorsque le pion du l'IA a fait son action

5.1.2 Intelligence Heuristique

Nous avons programmé une IA qui a pour objectif d'atteindre le trône adverse coûte que coûte en jouant avec sa meilleure pièce. Il attaquera toute pièce adverse à sa portée.

Pour procéder, l'IA calcul les distance entre toutes ses pièces et le trône adverse. Elle choisit de rapprocher ses pièces et donc choisit de préférence la pièce la plus loin. Elle peut aussi choisir sa pièce la plus forte.
Parmis les actions possibles avec cette pièce, elle choisit une action permettant naïvement de rapprocher la pièce du trône.

Une version améliorée de l'intelligence heuristique consiste à calculer un score associé à chaque case atteignable par un animal. **L'IA choisit dans un premier temps un animal au hasard et le déplace de façon à avoir un score maximal.** Vous l'aurez compris, le comportement de l'IA est donc entièrement déterminé par sa capacité à calculer le score des coordonnées voisines.

Pour être pertinent, le **calcul du score des actions de l'animal** doit prendre en compte certains paramètres :

1. La distance des prédateurs ennemis (animaux ennemis susceptibles de le manger)
2. La distance des proies ennemis (animaux ennemis susceptibles d'être mangés)
3. La distance aux objectifs (THRONES)
4. La distance des animaux alliés susceptibles de le protéger
5. La géographie du terrain

La liste n'est pas exhaustive et doit être complétée afin d'obtenir un comportement plus fin et anticipatif. Les distance sont exprimées en nombre de coups à jouer.

Notre première version prend en compte les paramètres 1, 2 et 3 comme expliqué ci-après.

Calcul du score

$$score = preyScore + predatorScore + objectifScore$$

$preyScore =$ $\exp(-\frac{x}{3} + 6.4) \cdot \frac{1}{d}$

avec x = la distance à la proie
et d = la distance de la proie à son objectif

$predatorScore =$ $(-200)^* \exp(-\frac{x}{2})$

avec x = la distance au prédateur

$objectifScore =$ $\exp(-\frac{x}{6} + 4.9)$

avec x = la distance à l'objectif

On peut voir graphiquement l'importance d'un ennemi dans le score :
preyScore=f(distance, d=4), predatorScore=f(distance), objectifScore=f(distance)

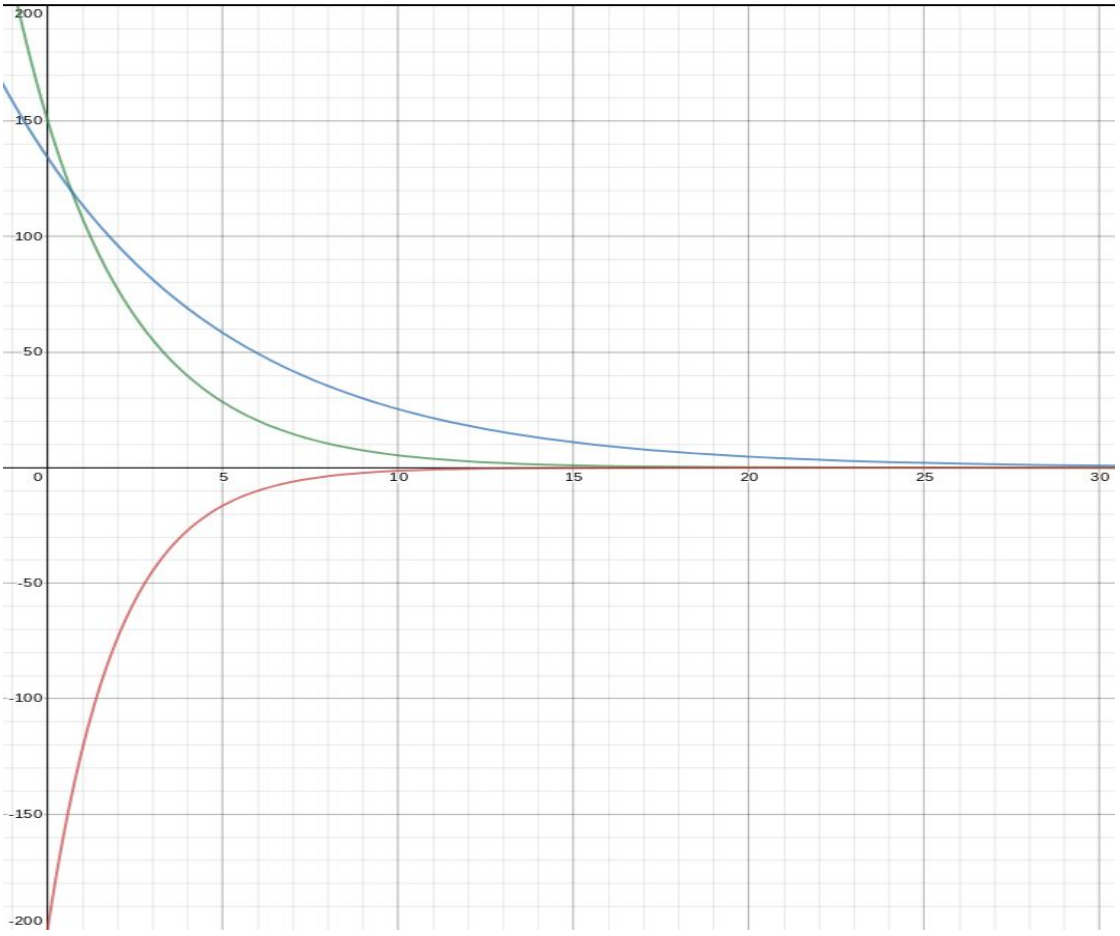


Figure 10 : Fonctions de scores de l'IA Heuristique

Nous avons aussi envisager de prendre en compte le nombre de pièce de chaque joueur ou le nombre de coup disponibles après l'action.

5.1.3 Intelligence Avancée

Pour réaliser notre intelligence avancée, nous avons envisagé l'algorithme Min-Max. Il consiste à donner un score non pas à une action mais à un état donné. Ainsi, l'ordinateur va pouvoir sélectionner l'état qui le favorise tout en défavorisant l'adversaire, et ce en anticipant *un maximum de tours* possibles (appelé *profondeur*).

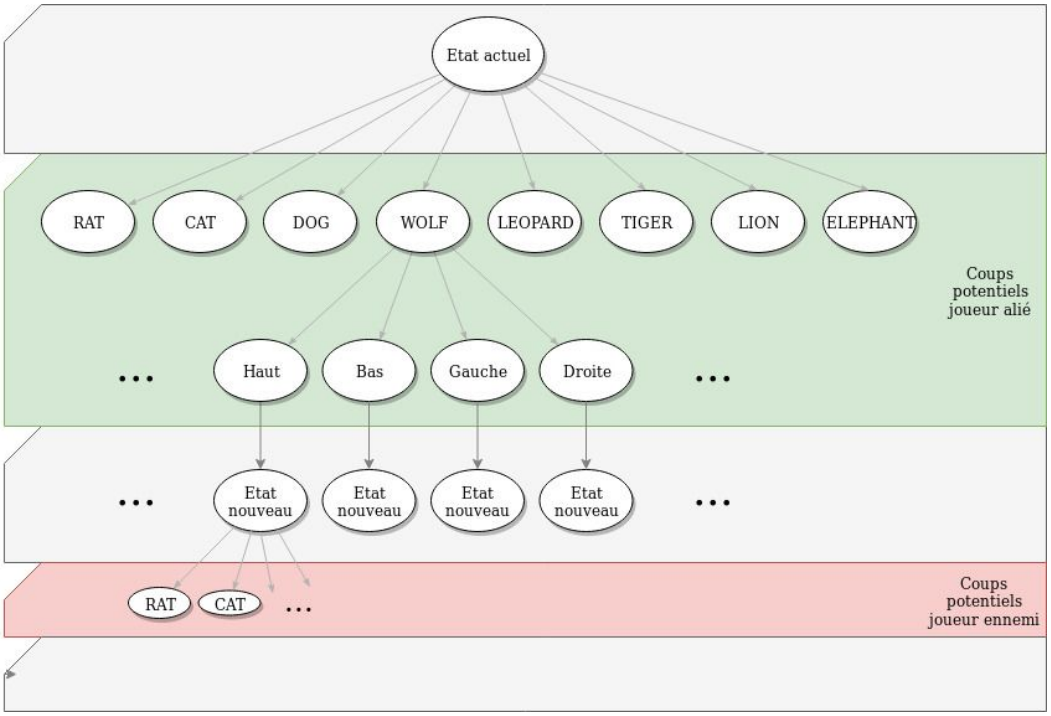


Figure 11 : Diagramme de l'algorithme Min-Max appliqué à Jungle War

Pseudo code général de l'algorithme minmax

```
fonction minimax(noeud, profondeur, maximizingPlayer) is
    if profondeur = 0 or noeud est un noeud terminal then
        return la valeur de noeud
    if maximizingPlayer then
        valeur := −∞
        for each fils de noeud do
            valeur := max(valeur, minimax(fils, profondeur − 1, FALSE))
        return valeur
    else (* min *)
        valeur := +∞
        for each fils de noeud do
            valeur := min(valeur, minimax(fils, profondeur − 1, TRUE))
        return valeur

(* Appel initial *)
```

minimax(origine, profondeur, TRUE)

5.2 Conception Logicielle

Le diagramme des classes pour l'intelligence artificielle est présenté ci-dessous.

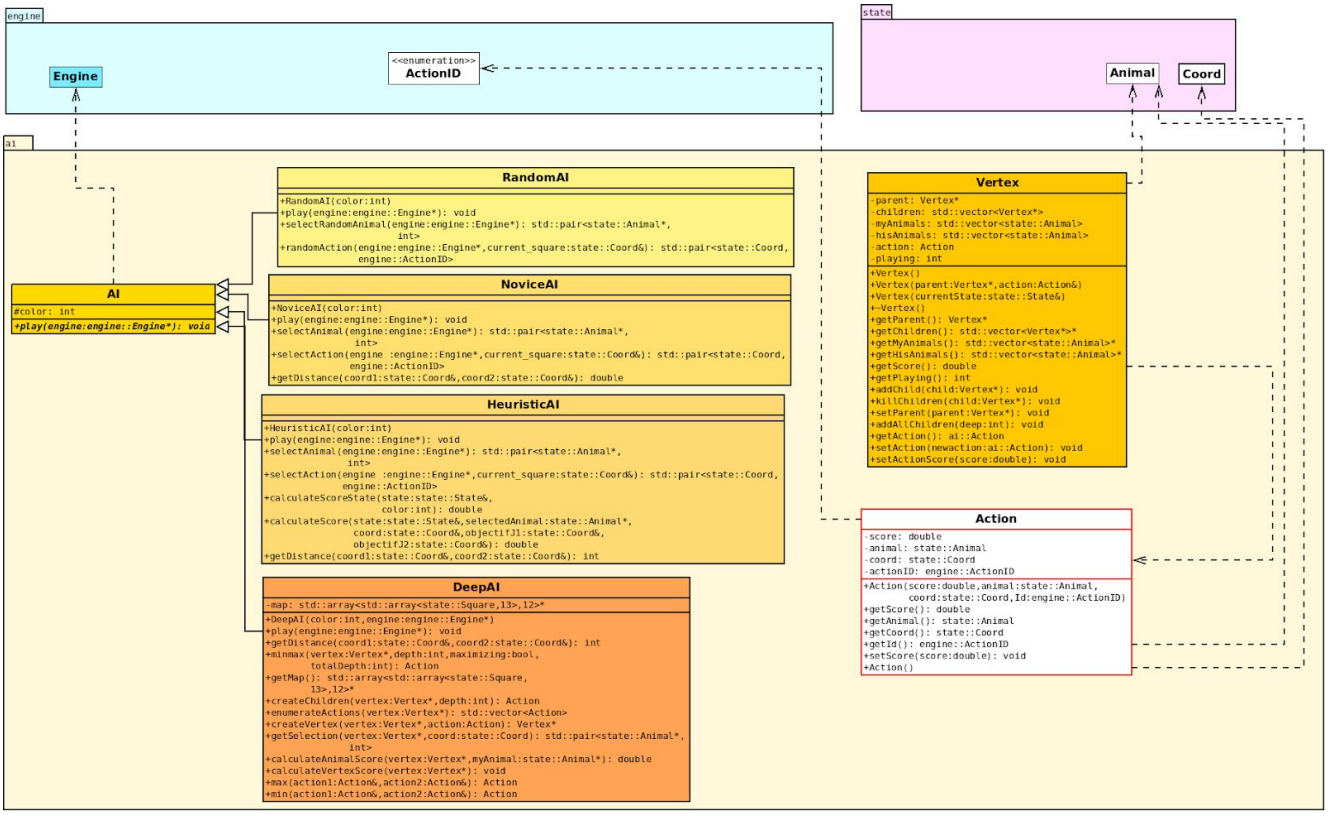


Figure 12 : Diagramme de classes de AI

Classe IA : Les classes filles de la classe IA implémentent différentes stratégies d'IA. Elle possèdent un camp et une fonction « run ».
Classe RandomAI : Classe qui implémente l'IA aléatoire.
Classe NoviceAI: Classe qui implémente l'IA Novice(qui détermine un coup favorable selon une stratégie basique). Elle possède par exemple une fonction permettant de calculer la distance euclidienne entre deux coordonnées, ce qui permet de déterminer une action à jouer.
Classe HeuristicAI : Classe qui implémente l'IA Heuristic. La méthode calculateScore() est pour l'instant utilisée par l'IA heuristique pour calculer le score des actions des animaux.
Classe DeepAI : Classe qui implémente l'IA Deep IA. La Deep IA est une engine amélioré avec sa propre authorisedAction : enumerateAction() et sa propre getSelection. enumerateAction() permet d'énumérer les Actions possibles de chaque pièce au prochain tour. Il va être utiliser pour créer l'arbre des choix. Il a pour attribut la map de State. Il utilise essentiellement les objets de la classe vertex et action pour permettre de créer un arbre des choix et appliquer l'algo min-max mais aussi le calcul de score.
Classe Vertex: Classe qui implémente la structure qui permet de créer l'arbre des choix et appliquer l'algo min-max. Nous l'avons imaginé comme un State light avec le minimum d'attribut possible pour calculer le score du Vertex et modifier l'état du jeu. Ainsi nous avons comme attribut la liste des animaux des deux joueurs, l'attribut playing. Un vecteur des Vertex fils et un attribut qui pointe son parent. Il a aussi un attribut de type Action Il y a deux

constructeurs qui permettent de créer des noeuds. Le constructeur qui prend pour input une référence de State permettant de créer le noeud du state light et l'autre constructeur qui va permettre de créer des fils en prenant les données de son père et modifiant son attribut Action.

Classe Action: Classe qui implémente la structure qui va remonter l'arbre de des choix lors de l'algorithme min-max. Il a pour attribut le Score d'un potentiel état, l'Animal qui va modifier l'état, la coordonnée de destination de l'animal et le type d'Action.

Implémentation de l'algorithme Min-Max:
Dans notre cas, nous souhaitons qu'avec un seul Vertex nous créons ses fils puis ensuite appliquer l'algorithme Min-Max. Nous souhaitons remonter choisir l'Action avec le meilleure score aux yeux de l'AI. Ainsi nous "remontons" le meilleur score puis au premier fils du Vertex nous remontons la meilleure Action. Pour créer l'arbre nous listons à chaque appel de la fonction les actions jouables du prochaine tour ensuite on crée tout les fils qui possède les informations de son père qui seront modifier par l'Action en input.

Pseudo code de l'algorihtme minmax adapté à notre jeu

```
ACTION fonction minimax(Vertex, profondeur, maximizingPlayer, profondeur initiale) is
  Liste des Actions = Calculer Liste d'Action potentielles à partir du Vertex ->
enumerateActions(Vertex);
 Vertex.Action =  Action_Vertex;
 if profondeur = 0  then
       calculer le score de Action_Vertex
       return l'Action_Vertex;
 if profondeur = profondeur initiale  then
      Action_Vertex.Score:= −∞
       for each Action de Liste des Actions
          Création d'un Fils du Vertex(Son Père, Action);
          Action_Vertex := max(Action_Vertex, minimax(fils, profondeur − 1, FALSE,
profondeur initiale))
         return Action_Vertex
  if maximizingPlayer then
      Action_Vertex.Score:= −∞
      for each Action de Liste des Actions do
          Création d'un Fils du Vertex(Son Père, Action);
          Action_Vertex.Score:= max(Action_Vertex.score, minimax(fils, profondeur − 1,
FALSE, profondeur initiale ).score)
       return Action_Vertex;
   else (* min *)
       Action_Vertex.Score := +∞
        for each Action de Liste des Actions do
          Action_Vertex.Score:= max(Action_Vertex.score, minimax(fils, profondeur − 1,TRUE,
profondeur initiale).score)
       return Action_Vertex;
```

Sources

Description stratego :
https://fr.wikipedia.org/wiki/Stratego

Description jeu des animaux :
https://fr.wikipedia.org/wiki/Jeu_du_combat_des_animaux

Elephant logo libre de droits :
https://publicdomainvectors.org/en/free-clipart/Elephant-silhouette-clip-art/79178.html

Plateau libre de droits :
https://www.drivethrurpg.com/product/273176/Jungle-Delta-Jungle-Map

Pseudo code minmax initial :
https://fr.wikipedia.org/wiki/Algorithme_minimax