Ecole Nationale Supérieure de l'Electronique et de ses Applications

Projet Latéral Transversal

Tales of Kornwal

OUAZZAGHTI Reda
et
ZOUHDI Zakaria

Projet de troisième année supervisé par
Prof. Gosselin

29 Septembre 2016

Version 1.1 du 29/09/16 8 décembre 2016

Table des matières

1 Objectif **2**
- 1.1 Présentation générale . . . 2
- 1.2 Règles du jeu . . . 3

2 Description et conception des états du jeu **4**
- 2.1 Description des états . . . 4
- 2.2 Conception logicielle . . . 5
- 2.3 Lien avec le rendu . . . 5

3 Stratégie et conception de rendu **7**
- 3.1 Stratégie de rendu . . . 7

4 Règles de Changements d'Etats **9**
- 4.1 Moteur de rendu . . . 9
- 4.2 Pattern Command . . . 9
- 4.3 Changements extérieur . . . 10

5 Stratégie d'intelligence artificielle **11**
- 5.1 Stratégie de rendu . . . 11
  - 5.1.1 Stratégie d'IA simple : le zombie . . . 11
  - 5.1.2 Stratégie d'IA moyenne : le "groupe" . . . 12
  - 5.1.3 Stratégie d'IA Forte : Pathfinding . . . 12

6 Modularisation **14**
- 6.1 Parallélisation du moteur de jeu . . . 14

7 API web **15**
- 7.1 Services Commandes . . . 15
  - 7.1.1 Méthode PUT . . . 15
  - 7.1.2 Méthode GET . . . 16
- 7.2 Services Connexion Utilisateur . . . 16
  - 7.2.1 Méthode PUT . . . 17

Chapitre 1

Objectif

1.1 Présentation générale

By David Revoy / Blender Foundation - Own work, CC BY 3.0

Tales of Kornwal est un jeu vidéo basé sur les mêmes règles de Fallout Tactics, *i.e* un jeu d'aventure doté d'un système de combat en tour par tour, permettant aux joueur de progresser et interagir avec un univers à l'allure originale à mi-chemin entre le médiéval-fantastique et le post-apocalyptique.

Les interactions seront basées sur un système de gestion d'inventaire et d'au moins une caractéristique, qui feront office de modificateurs lors d'actions enterprises par le personnage (*e.g* : la caractéristique "Force" influera grandement sur les dégâts infligés par un ennemi ou par le héros, ainsi que l'utilisation de telle ou telle arme).

1.2 Règles du jeu

Le jeu pourra posséder plusieurs aspects dépendant de l'étude du cahier des charges :

— Déplacement d'un personnage sur une "zone" de la mappemonde, accédant aux différentes cases nord-sud-est-ouest de la map en cliquant sur l'une des extrémités de l'écran.

— Système de combat tour par tour : lors d'une rencontre avec un ennemi, la map se vide de tous les sprites autres que le personnage joueur et ses adversaires, laissant donc place au duel entre le héros et l'ennemi. Le joueur commencera en premier (sauf modification) et disposera de deux choix possibles : se déplacer d'une case dans la zone, ou attaquer l'ennemi, faisant baisser son capital de points de vie. Le nombre de points de vie retirés dépendra de la caractéristique FORCE du personnage, ainsi que de son ARME équipée. Le tour se termine après que l'une des actions suivante a été effectuée, laissant le tour à l'ennemi (qui se déroulera de la même manière).

— Système d'interaction avec les personnages joueur ou non-joueur (discussions, interface d'échange d'objets)

Chapitre 2

Description et conception des états du jeu

2.1 Description des états

Un état du jeu est défini par les éléments suivants :

— **Les élements fixes** (classe StaticElement). Elle est composée de deux classes filles : d'une part, une classe "Static", correspondant aux éléments immobiles permettant de définir les zones où les personnages peuvent se déplacer (*e.g* herbe, sable, neige...). Ces élements peuvent être de type "CombatZone" si le personnage se trouve dans une zone d'influence ennemie, "NoCombatZone" s'il ne se trouve pas dans une telle zone, ou de type "ChangeMap" : lorsque le personnage se trouve dans cet espace, il bascule dans une nouvelle map, différente de celle où il était. D'autre part, une classe "Wall", définissant les zones infranchissables où le personnage ne peut se déplacer. IL peut s'agir d'un obstacles (arbre) ou tout simplement d'un bord de la map. (*e.g* mur).

— **Les personnages** (classe Character), éléments mobiles se déplaçant sur la grille (ou plus exactement sur les cases définies par les classes Space). Ces éléments peuvent être contrôlés soit par l'humain (Personnage joueur), soit par les IA (Personnages non joueurs). Ils possèdent tous une position définie par ses coordonnées X et Y, ainsi qu'une direction. Chaque personnage possède un certain nombre de données : niveau, expérience, force, points de vie etc .. Les personnages possèdent trois status possibles :

  — YOURTURN : Le tour du personnage. Il aura le droit de dépenser des points de mouvements pour se déplacer sur la map, ou des

points d'actions pour endommager les adversaires.

— HISTURN : Le tour des autres personnages. Le personnage doit rester immobile et subir les actions des autres personnages jusqu'à ce que son tour arrive.

— DEAD : Le personnage ne peut plus bouger, son tour n'arrivant jamais : il est mort.

2.2 Conception logicielle

L'architecture du logiciel est composé des élements suivants :

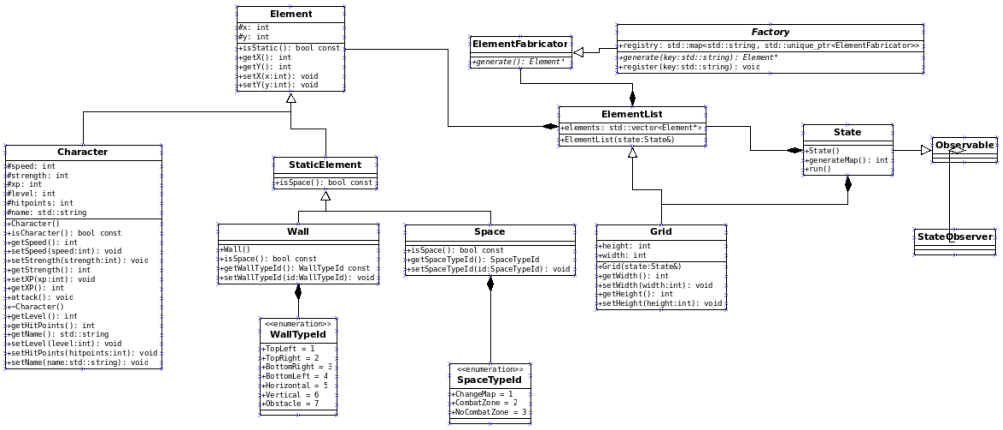
— une classe "Elements" : les classes "Character" et "Space" héritent toutes les deux d'une classe "Elements". La méthode de type *bool isStatic* permet de distinguer la classe d'élements statiques de celle des éléments mobiles en renvoyant *true* si l'objet est de type "StaticElement", false sinon.

— des conteneurs d'éléments : la classe "ElementLists" contient une liste d'éléments fixes et mobiles. La classe "Grid" hérite de la classe "ElementList" afin de disposer ces éléments sur une map, ou "grille". Le tout correspond à un état donné, qui est un objet de la classe "State".

— une "Factory" ou fabrique d'éléments : il s'agit d'une interface permettant d'instancier une liste d'éléments sans avoir à spécifier la famille d'objets.

2.3 Lien avec le rendu

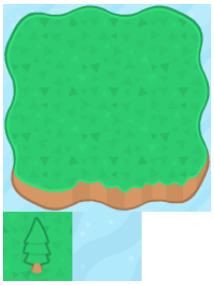
Il se fait via la classe "StateObserver", qui suit le pattern design "Observer". Cette classe "observe" les changements d'états et en avertit le client via le rendu. A terme, il faudra rajouter une interface afin de distinguer les différents évènements (changement de map etc ..).


Element
#x: int
#y: int
+isStatic(): bool const
+getX(): int
+getY(): int
+setX(x:int): void
+setY(y:int): void
ElementFabricator
+generate(): Element*
Factory
+registry: std::map<std::string, std::unique_ptr<ElementFabricator>>
+generate(key:std::string): Element*
+register(key:std::string): voic
ElementList
+elements: std::vector<Element*>
+ElementList(state:State&)
State
+State()
+generateMap(): int
+run()
Observable
Character
#speed: int
#strength: int
#xp: int
#level: int
#hitpoints: int
#name: std::string
+Character()
+isCharacter(): bool const
+getSpeed(): int
+setSpeed(speed:int): void
+setStrength(strength:int): void
+getStrength(): int
+setXP(xp:int): void
+getXP(): int
+attack(): voic
+~Character()
+getLevel(): int
+getHitPoints(): int
+getName(): std::string
+setLevel(level:int): void
+setHitPoints(hitpoints:int): void
+setName(name:std::string): void
StaticElement
+isSpace(): bool const
Wall
+Wall()
+isSpace(): bool const
+getWallTypeId(): WallTypeId const
+setWallTypeId(id:WallTypeId): void
Space
+isSpace(): bool const
+getSpaceTypeId(): SpaceTypeId
+setSpaceTypeId(id:SpaceTypeId): void
Grid
+height: int
+width: int
+Grid(state:State&)
+getWidth(): int
+setWidth(width:int): void
+getHeight(): int
+setHeight(height:int): void
StateObserver
<<enumeration>>
WallTypeId
+TopLeft = 1
+TopRight = 2
+BottomRight = 3
+BottomLeft = 4
+Horizontal = 5
+Vertical = 6
+Obstacle = 7
<<enumeration>>
SpaceTypeId
+ChangeMap = 1
+CombatZone = 2
+NoCombatZone = 3

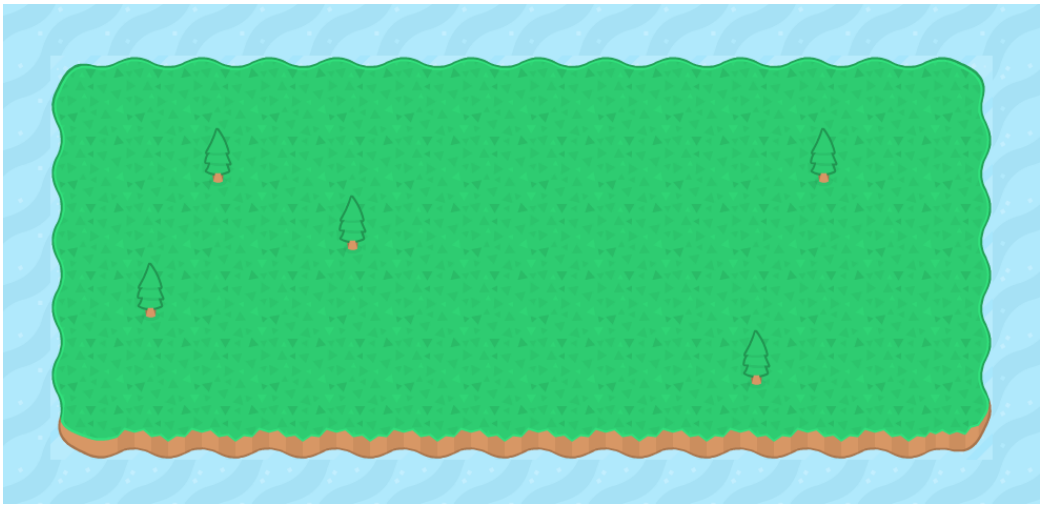
Chapitre 3

Stratégie et conception de rendu

3.1 Stratégie de rendu

Le premier objectif a été de pouvoir générer une map à partir d'une tileset. Il s'agit d'une image constitué d'unités élémentaires de texture (tuiles), qui serviront de base pour la map. Nous écrivons ensuite un code qui attribue à chaque tuile un identifiant entier, puis qui stocke dans un tableau le numéro de chaque tuile que l'on veut voir apparaître. La position de la tuile dans le tableau détermine sa position dans la future map. Voici un exemple de tileset : 

Voici une map généré grâce à ce TileSet, obtenu en lançant le main :



On crée une classe Tile dans un package "rendu" ; deux classes héritent de cette classe Tile, les classes Map et CharacterMap (exploitée plus tard). De la même manière que le code mentionné avant, la classe Map stocke les coordonnées d'une tuile et la classe Tile stocke l'ensemble des tuiles pour un état donné. Il faudra donc associer à chaque instance d'élément une tuile, puis associer une liste d'élements et une "Grid" à un état, puis générer la map correspondant à cet état (pour l'instant statiques).

Chapitre 4

Règles de Changements d'Etats

4.1 Moteur de rendu

Les Changements d'états se fait avec le moteur de rendu. Lors d'un mouvement (commandé par une touche du clavier), il y a un ordre géré par le moteur (package Engine) qui met à jour l'état instancié, qui à son tour met à jour la position des personnages, et le personnage est alors déplacé vers la zone indiquée. Le personnage peut se déplacer à droite, à gauche, vers le haut ou vers le bas. Pour cela, chaque objet"Character" dispose d'une énumération "Direction". Pour permettre le déplacement, la classe Engine dispose d'une méthode "moveCharacter()", qui prend en arguments un pointeur de Character et une direction. Selon la direction entrée, la fonction modifie les coordonnées X et Y via par exemple les méthodes "getX()" (récupération de la position X actuelle) et "setX" de l'élément pour la coordonnée X. Afin que le changement de position du personnage soit aussi appliqué sur le sprite, on met en place un observable "PositionObs", qui dispose de deux méthodes "notifyPosX()" et "notifyPosY()" pour synchroniser le changement de position du personnage à celui du sprite. On n'a plus qu'à appliquer la méthode moveCharacter dans le main, dans la boucle pollEvent. Enfin, on évite les sorties de map via de simples conditions if.

4.2 Pattern Command

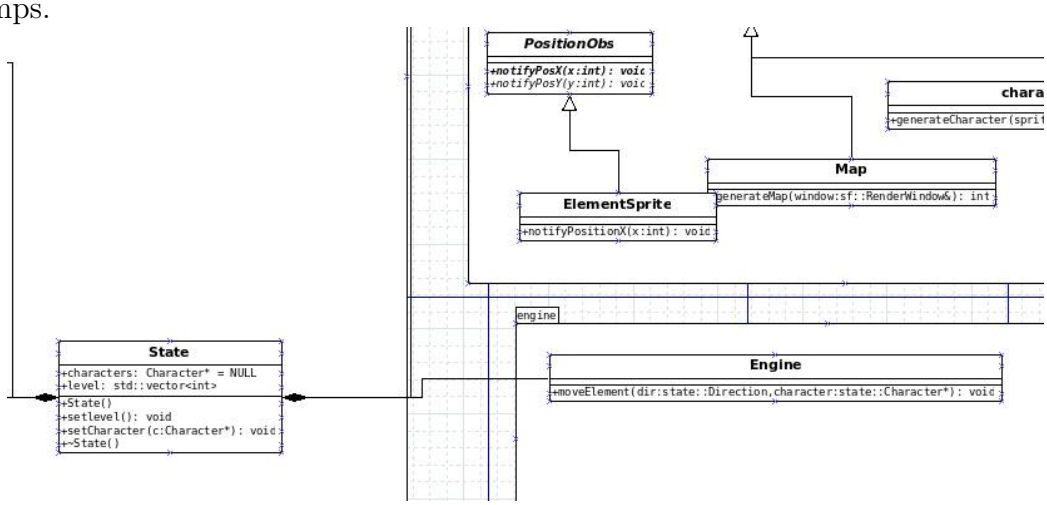
On utilise un pattern de type Command pour gérer les commandes et mettre à jour en conséquence l'état du jeu. On dispose pour cela : D'une classe abstraite mère « Command », contenant la méthode abstraite run( ) : c'est la méthode qui lance la commande en question. De deux classes filles, une pour chaque type de commande : une classe MoveCommand pour gérer

le déplacement et une classe AttackCommand pour la gestion des attaques des personnages (diminution des points de vie de l'adversaire)

Nous ajoutons dans la classe Engine deux listes de commandes : une liste d'attente « waitingCommands » où sont stockées provisoirement les commandes reçues et une liste active « activeCommands » qui contient les commandes destinées à être exécutées lors de la mise à jour. Celle-ci se fait via la méthode update( ), qui copie les commandes de la liste d'attente dans la liste active, vide la liste d'attente, et lance les commandes de activeCommands. Enfin, update() supprime les commandes de la liste active. On dispose également d'une méthode « addCommands » pour stocker les commandes dans waitingCommands.

4.3 Changements extérieur

Le constructeur de State, contenue dans le package du même nom, lui permet d'initialiser l'attribut level de la classe, chargeant ainsi un niveau. Le second attribut de type Element* permet d'acceder directement à l'élément et par conséquent, à le modifier. Une amélioration permettra d'accéder à une liste d'élements, afin d'instancier et mettre à jour plusieurs sprites en même temps.



Chapitre 5

Stratégie d'intelligence artificielle

5.1 Stratégie de rendu

Dans le cadre d'un jeu de rôle, les différents types personnages adversaires suivent chacun un comportement distinct, en fonction de leur "intelligence" au sein de l'univers. Aussi, un zombie aura le comportement prévisible d'errer sur la map jusqu'à croiser un personnage, et de l'attaquer jusqu'à ce que mort s'en suive. Un gobelin, sensé être frêle et sournois, aura tendance à adopter un comportement plus complexe en commençant d'abord par chercher les gobelins les plus proches, s'unir, et se déplacer au hasard en bande... Avant de lâchement battre en retraite s'il se retrouvent en situation de 1 contre 1. Enfin, un orque rodé au méthodes d'embuscades tentera de traquer le personnage en prenant le chemin le plus court menant à lui, mais en ayant pour priorité de joindre ses forces aux autres orques de la map avant d'attaquer le personnage.

En terme d'implantation, toutes les IA sont capables de jouer le jeu et les différentes entités. Il sera même également possible d'implanter des personnages à des niveaux d'intelligence variés, l'entrée des méthodes de la classe IA comprenant un argument "State", donnant accès à la liste des personnages.

L'ajout d'un sleep (fonction de la bibliothèque unistd) pour une durée d'une seconde permet d'observer le comportement du personnage en détail.

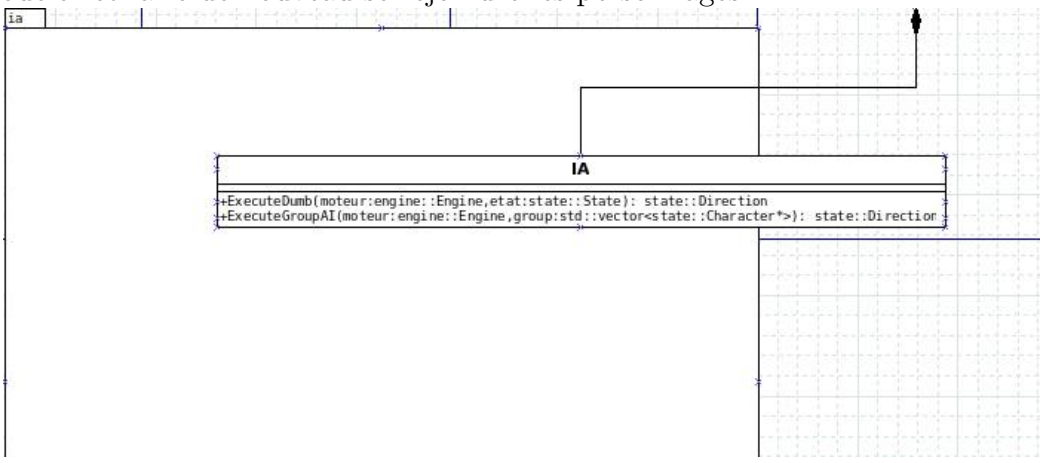
5.1.1 Stratégie d'IA simple : le zombie

L'intelligence artificielle des zombies est analogue à celle des canons fantastiques : peu intelligent, vagabondant de manière aléatoire jusqu'à croiser

une proie. Le zombie ayant 4 possibilités de déplacement sur la map, il en choisit aléatoirement une des 4, jusqu'à croiser le héros. En terme de code, les directions sont choisies à l'aide d'un entier généré aléatoirement, exécutant la commande de déplacement pour chacune des directions aléatoires choisies. Tout ceci se fait dans la méthode "Execute()" de la classe IA.

5.1.2 Stratégie d'IA moyenne : le "groupe"

L'intelligence artificielle moyenne repose sur le principe de l'union faisant la force, tentant de regrouper les personnages au plus vite, et à se déplacer alors en groupe mais au hasard, en respectant une distance maximale de deux cases (plus ou moins 100 pixels). Ayant en entrée un groupe d'objets "Characters", l'IA vérifie à chaque itération que tous les personnages présent sur la carte sont à moins de 100 pixels de distance que ce soit sur l'axe X ou Y, faute de quoi l'AI ordonne au personnage de se déplacer sur le même axe, vers la direction des autres personnages. Une fois la condition de proximité respectée, l'intelligence artificielle fait déplacer les éléments dans des directions aléatoires jusqu'à ne plus respecter ladite condition, avant de reboucler et faire de nouveau se rejoindre les personnages.



5.1.3 Stratégie d'IA Forte : Pathfinding

L'objectif est de ramener le personnage soldat IA vers le personnage héros le plus rapidement possible et en évitant les obstacles éventuels. On utilise un algorithme s'inspirant celui de l'algorithme « A* ». C'est un algorithme itératif : à chaque itération, on tente de se rapprocher de la destination. On dispose pour cela le deux listes de nœuds : une liste ouverte, qui contient tous les nœuds déjà visités, et une liste fermée, contenant les nœuds faisant partie du chemin solution. Pour calculer le poids de chaque nœud, on calcule

la distance entre le point étudié et le dernier jugé comme bon, ainsi que la distance entre le point étudié et le point de destination. La somme de ces deux distances nous donne la qualité du noeud étudié. Plus un noeud a une qualité faible, meilleur il est. Voici les étapes de l'algorithme :

— On commence par le noeud de départ, c'est le noeud courant

— On regarde tous ses noeuds voisins

— si un noeud voisin est un obstacle ou qu'il est dans la liste fermée, on l'oublie

— si un noeud voisin est déjà dans la liste ouverte, on met à jour la liste ouverte si le noeud dans la liste ouverte a une moins bonne qualité (et on n'oublie pas de mettre à jour son parent)

— sinon, on ajoute le noeud voisin dans la liste ouverte avec comme parent le noeud courant

— On cherche le meilleur noeud de toute la liste ouverte. Si la liste ouverte est vide, il n'y a pas de solution, fin de l'algorithme

— On le met dans la liste fermée et on le retire de la liste ouverte

— On réitère avec ce noeud comme noeud courant jusqu'à ce que le noeud courant soit le noeud de destination.

Pour cela on dispose de la classe Node ainsi que d'une classe PathFinding. La classe Node possède comme attributs les coordonnées x, et y ainsi qu'un poids. La classe PathFinding possède un certain nombre de méthodes permettant d'appliquer l'algorithme :

— Une méthode "distance" pour calculer la distance "à vol d'oiseau" entre 2 points (à partir des attributs x et y des objets Characters).

— Une méthode "ajouterCase" pour ajouter un noeud voisin à la liste.

— Une méthode "bestNode" pour déterminer le meilleur noeud de la liste.

— Enfin, une méthode "findPath" qui sert à déterminer le chemin une fois la destination atteinte.

Chapitre 6

Modularisation

6.1 Parallélisation du moteur de jeu

L'implantation récente du Pattern Command (cf p.9) permet de simplifier la mise en place d'un thread permettant de gérer entièrement les mises à jour et l'execution des commandes. Afin d'éviter que des commandes se perdent entre l'exécution et la mise à jour du jeu, un double tampon de commandes sera utilisé, waitingCommands et activeCommands. Lors d'un update (= exécution des commandes appelées par les touches de clavier, chargée dans waitingCommands), waiting charge toutes les commandes dans la liste active, qui s'occupera d'effectuer les changements de l'état du jeu. Ainsi, waitingCommands sera toujours disponible pour un ajout de commandes, et ce même durant l'exécution de commandes antérieures.

Chapitre 7

API web

7.1 Services Commandes

Pour les commandes, on utilise une méthode PUT pour déposer les commandes vers le serveur, et une méthode GET pour obtenir l'information sur l'époque du jeu.

7.1.1 Méthode PUT

(i) **Requête** :

(a) Méthode HTTP : PUT

(b) URI : PUT /COMMANDES/

(c) Données :

```
type: "object",
properties: {
    "command": { type:string, pattern: "(d|at|in)" }, //si command =d
    "epoch": { type:int},
    "dir": { type:int},
    "character": { type : string },

},
required: [ "commande", "epoque" ]
```

(ii) **Réponse** :

Cas où la commande est validée :

(a) Status : 200

(b) Données :

```
type: "object",
properties: {
    "epoch": { type:int},
    "dir": { type:int},
    "character": { type : string },

},
required: [ "epoch" ]
```

Cas : problème interne :

(a) Status : 500

7.1.2 Méthode GET

(i) **Requête** : GET / Commande / ¡epoch¿.

(ii) **Réponse** :

Cas où l'époque existe :

(a) Status : 200 (=ok)

(b) Données :

```
type: "object",
properties: {
    "command": { type:string, pattern: "(d|at|in)" },
    "epoch": { type:int},
    "dir": { type:int},
    "character": { type : string },

},
required: [ "command","epoch" ]
```

Cas où l'époque n'existe pas :

— Status : 404

— Pas de données .

Cas erreur interne :

— Status : 500

7.2 Services Connexion Utilisateur

Dans ce service, on a besoin d'une méthode PUT pour ajouter un utilisateur au jeu en cours, et une méthode DELETE pour supprimer un utilisateur.

7.2.1 Méthode PUT

(i) **Requête** : GET / Commande / ¡epoch¿.

(a) Méthode HTTP : PUT

(b) URI : PUT /LOGIN/

(c) Données :

type: "object",
properties: {
"character": { type : string },

},
required: ["character"]

(i) **Réponse** :

Cas où la connexion s'est bien déroulée :

(a) Status : 200 (=ok)

(b) Données :

type: "object",
properties: {

"id_perso": { type:int},

},

Cas où l'époque n'existe pas :

— Status : 403. Pas de données.

Cas erreur interne :

— Status : 500