Projet Développement Web Avancé n.3: Curve Fever (Agario like)

PANCHALINGAMOORTHY Gajenthran, MOTAMED Amin

Lundi 7 Janvier 2019

Table des matières

1 Introduction

Notre groupe constitué de PANCHALINGAMOORTHY Gajenthran et de MOTAMED Amin a pour but de réaliser le jeu Curve Fever (Agario-like). Crée en 1995 par Filip Oščádal et Kamil Doležal, le jeu Curve Fever (également connu sous le "Achtung, die Kurve") est un jeu de stratégie en multijoueur qui peut être joué jusqu'à 8 joueurs (source : Wikipedia). La seule règle de ce jeu est d'être le dernier en vie. Pour cela, il ne faut pas rentrer en collision avec un autre serpent (ou soi-même). Nous avons légèrement modifié les règles de ce jeu. En effet, désormais le but du jeu est soit d'avoir le plus de points possibles (les pommes rapportent des points par exemple) ou soit d'être le dernier en vie. Attention, il y a également des pièges dans le jeu qui risquent de vous tuer. Le serpent a la possibilité d'avancer soit à droite, soit à gauche, il ne peut pas revenir en arrière et il est constamment en déplacement.
Nous avons également décidé de modifier certains comportement d'un Curve Fever classique (nous expliquerons plus en détail dans les prochaines parties).

2 Architecture du code

Au delà de la réalisation du jeu, un de nos objectifs était de bien organiser notre code et d'avoir une bonne architecture pour notre code afin de pouvoir plus facilement le maintenir dans le temps. Nous espérons avoir accompli cette objectif.
Les informations/fonctionnalités côté client sont bien évidemment séparées des informations/fonctionnalités provenant du serveur. Il faut bien dissocier les deux pour obtenir une architecture cohérente.
La communication entre le client et le serveur se déroulera de telle manière :

- le client enverra les entrées saisies par l'utilisateur et se contentera d'afficher le jeu à l'aide des données transmis par le serveur.
- Le serveur s'occupera de la gestion du jeu et des éléments du jeu (personnes, objets...). Il ajoutera, modifiera et/ou supprimera des éléments à l'aide (ou non) des entrées saisies par l'utilisateur.

Par exemple dans notre jeu :

- le client envoie les entrées claviers au serveur et, à l'aide des données des serpents et des objets dans le jeu, il se charge d'afficher les éléments à l'écran.
- Les entrées claviers envoyées par le client permettront au serveur de modifier les coordonnées du serpent (ou pour être plus précis la direction du serpent) et auront donc une influence sur le déplacement de

notre serpent (et qui peut avoir également des répercussions sur les autres éléments du jeu, gérées par le serveur). Une fois les différents éléments du jeu mis à jour, on transmet les données du serpent et des objets du jeu au client.

L'idéal est d'envoyer le moins de contenu inutile possible entre le client et le serveur afin de fluidifier les transactions.

3 Répartition des fichiers

Comme nous l'avons dit précédemment, les fichiers sont répartis de telle sorte que les fonctionnalités du client ne soient pas mélangées directement avec celles du serveur.
Dans un premier temps, nous retrouverons un dossier *public* qui comme son nom l'indique est « publique » et donc visible par tout le monde : il concerne donc le client. Dans celui-ci, nous aurons :

- le dossier *css* pour la feuille de style afin de s'occuper de la mise en forme de notre page (mais nous n'aurons pas réellement besoin du dossier *css*, car nous allons nous occuper de la mise en forme directement avec Javascript : l'idée est de tout réaliser avec Javascript).
- Le dossier *img* qui regroupe l'ensemble des images nécessaires pour l'affichage de notre jeu.
- *index.html* concerne notre page HTML
- Et le dossier JS qui rassemble tous les fichiers JS côté client (ça sera le dossier qu'on va principalement modifier)

Puis dans un second temps, nous regrouperons dans le dossier *src* l'ensemble des fonctionnalités côté serveur et donc de la gestion du jeu : ils s'agit seulement de fichier JS.
Et enfin pour faire la relation entre les deux dossiers, nous avons *server.js*. Le fichier *server.js* va nous permettre de créer une relation entre le dossier *public* (côté client) et le dossier *src* (côté serveur) à l'aide de plusieurs outils que nous verrons dans d'autres sections.
Remarque : Il existe également d'autres fichiers, moins important pour l'architecture de notre code comme le fichier *README.md* qui permet d'offrir une description du jeu (règles du jeu) et d'autres détails qui peuvent sembler nécessaire pour les utilisateurs (commande pour lancer le jeu, par exemple). Et le *package.json* qui permet d'offrir une description de notre code (les auteurs, le nom ...), la manière dont on lance notre programme et les dépendances/outils nécessaires pour notre programme.

4 Détails du code

Dans cette section, nous allons essayer de vous détailler notre code : la manière dont nous l'avons construit, les différentes fonctions à savoir, les outils utiles pour le bon déroulement du code...
Pour cela, nous allons dans un premier temps parler du fichier *server.js*, puis des fichiers côté client et enfin décrire les fichiers présents dans le dossier *public*.
Avant de passer aux sections suivantes, nous tenons à vous rappeler que pour pouvoir lancer notre programme, il suffit de lancer la commande suivante :
`node server.js`

4.1 Server.js

Comme nous l'avons dit précédemment, le fichier *server.js* va nous permettre de créer la relation entre le client et le serveur. Pour cela, nous avons besoin de plusieurs outils afin de pouvoir créer cette communication. Pour créer notre serveur nous utiliserons le framework express afin de nous faciliter la tâche (conseillé par la majorité des gens pour sa puissance et sa facilité). La communication entre le client et le serveur se fera à l'aide de Socket.IO qui nous permettra de faire de la communication en temps réel (communication synchrone), nécessaire pour notre jeu. Donc d'abord, nous allons commencer par importer les outils nécessaires pour la création du serveur et la communication (à l'aide de `require`), puis nous initialiserons ces outils comme sur la figure suivante.

En plus de lui dire le port sur lequel il doit écouter (ici port 8080), nous devons également lui servir les fichiers statiques qui se trouvent dans le dossier public et définir le routage afin de répondre aux demandes du client.

Une fois, le serveur géré, il faut maintenant s'occuper de la communication à l'aide des WebSocket. Pour établir une connexion entre le client et le serveur, il faut que le client se connecte au serveur à l'aide de `io.connect`. Cela va permet de créer également un objet Socket qui va pouvoir différencier les clients les uns des autres (notamment avec le `socket.id`). Une fois la demande du client réalisée, le serveur va intercepter cette demande avec la fonction `on` et va pouvoir ainsi écouter les différentes interventions/actions du client en ayant toujours en paramètre le Socket du client. Dans notre programme, le serveur va écouter que trois interventions provenant du client : la création d'un nouveau joueur, les entrées clavier et la déconnexion d'un joueur.
Pour cela nous avons donc créé une fonction `listen` qui prend en paramètre le socket du client (pour savoir à qui appliquer les modifications), le mes-

sage envoyé, et la fonction à appliquer pour ce message. Il faut savoir qu'il y a deux façons d'échanger des informations entre le client et le serveur : la fonction `emit` (qui permet d'envoyer des données) et l'autre partie répond à l'envoi de données à l'aide de la commande `on` en appelant une fonction fournie en paramètre. La fonction `listen` se contente seulement d'écouter le client et donc de répondre à l'envoi de données avec `on` (en réalité il reçoit seulement les entrées claviers comme données).
Ainsi, nous avons réussi à créer un serveur et à établir la connexion avec un client. Cependant, il manque un élément essentiel dans notre code : le jeu. En effet, les données reçues par le client vont nous permettre de réaliser des modifications dans notre jeu côté serveur. De plus, le jeu va également transmettre ces modifications au client afin qu'il puisse dessiner cela sur son écran. Nous avons donc décider d'initialiser la classe `Game` (que nous détaillerons dans la prochaine section) qui regroupera l'ensemble des fonctionnalités gérant la création, la modification et la suppression des éléments du jeu à savoir les items et les joueurs), et à chaque intervalle de temps réguliers, nous mettrons à jour les éléments du jeu et nous enverrons ces éléments aux clients. Nous allons donc parler dans la section suivante de quoi est constitué notre jeu côté serveur.

4.2 Serveur

Les actions côté serveur sont dans le dossier *src*. Le dossier *src* contient l'ensemble des fonctionnalités s'occupant de la gestion du jeu à savoir le plateau de jeu, les joueurs ou encore les items/objets (et plus généralement les éléments présents dans notre jeu). Il va s'occuper d'ajouter, de retirer ou de modifier les éléments du jeu et leurs comportements. La classe `Game` (situé dans le fichier *game.js*) englobe l'ensemble des classes présents dans *src* : le `World` (*world.js*) qui correspond au plateau de jeu, les `Snake` (*snake.js*) qui correspondent aux serpents, les `Player` (*player.js*) qui correspondent aux joueurs et les `Item` (*item.js*) qui correspondent aux items/objets du jeu.

4.2.1 Game (game.js)

La classe `Game` rassemblera les éléments de notre jeu, elle contiendra le plateau de jeu, les joueurs et les items du jeu. En initialisant le jeu (dans la fonction `init`), nous créons également le plateau de jeu et nous fixons un timer afin de pouvoir faire apparaître des objets à des intervalles de temps réguliers.
Le `Game` s'occupe de réaliser 4 tâches essentielles : l'ajout d'un élément, la mise à jour d'un élément, la suppression d'un élément et la transmission des

éléments au(x) client(s).
Pour ajouter des éléments, on utilise les fonctions :

- `addNewItem` qui permet de créer un nouvel item aléatoire avec des coordonnées également aléatoires dans le jeu et de le stocker dans la liste des objets du jeu. On pense également à l'insérer dans le plateau de jeu avec `inserItem`.
- `addNewPlayer`, il s'agit une des fonctions provenant de `io.on` qui se déclenche lorsque le client lance une demande au client. Ici, c'est pour créer un nouveau joueur dans le jeu (qui représentera le client ayant fait la demande). On va donc créer un paquet comprenant le nouveau joueur créé avec des coordonnées aléatoires, l'ensemble des ennemis présents dans le jeu (ou plutôt les autres joueurs) et l'ensemble des items présents dans le jeu. Ce paquet va ensuite être envoyé au client ayant fait la demande en utilisant la fonction `emit` (on sait que nous envoyons bien au client ayant fait la requête car on obtient en paramètre de la fonction l'objet Socket du client, donc lorsqu'on va `emit`, on utilisera ce Socket). Une fois transmis, nous stockons, comme les objets, le joueur dans la liste des joueurs. Ici les joueurs correspondent à des `Map` où la clé est le `socket.id` du joueur/client et la valeur est la classe Joueur et donc le serpent en quelque sorte) afin de bien différencier chaque client reçoit un socket ID unique "normalement" qui va lui permettre de communiquer avec le serveur et vice versa.

Pour mettre à jour des éléments, on utilise les fonctions suivantes :

- `updatePlayerInput` comme `addNewPlayer`, il s'agit d'une fonction qui est écoutée par le serveur. La fonction permet de mettre à jour la direction du serpent à l'aide des entrées claviers passées en paramètre de la fonction (provenant du client) à l'aide de la méthode `update` de classe `Player`, en choisissant le bon joueur qui incarne le serpent (c'est pour cela que nous avons choisi de créer un `Map` pour les joueurs pour les reconnaître plus facilement avec les méthodes `has` pour savoir si le joueur existe et `get` pour récupérer le joueur grâce à sa clé à savoir le `socket.id` passé en paramètre de la fonction)
- `update` se contente d'appeler la méthode `updatePlayers` et `updateItems` de la classe `Game`.
- `updatePlayers` qui permet de mettre à jour les joueurs. On va donc bouger les serpents en modifiant leurs coordonnées (à l'aide des entrées claviers passés par le client dans la fonction `updatePlayerInput`), vérifier si les serpents sont en collision avec d'autres serpents ou d'autres éléments du jeu, les placer dans notre plateau de jeu et vérifier si ils sont toujours en vie (si un serpent n'est pas envie on le retire du jeu

avec la fonction `removePlayer`).

- `updateItems` pour mettre à jour tous les items. La mise à jour consiste à retirer tous les objets déjà consommés et faire apparaître de nouveaux objets si le timer est à 0.

Pour retirer un élément du jeu (en réalité seulement les joueurs vu que les items nous les gérons directement dans `updateItems`), on utilise les fonctions suivantes :

- `removePlayer` qui retire un joueur du jeu selon le socket ID passé en paramètre (fonction appelé lorsque le joueur se déconnecte). Pour le retirer du jeu, nous le retirons de la liste des joueurs et nous effaçons son serpent du plateau de jeu (`clearSnake`).
- `removeAllPlayer` qui retire tous les joueurs du jeu. Pour l'instant pas forcément utiliser. Mais vu que nous n'avons pas terminé encore le projet et qu'il peut facilement être améliorable, nous pouvons pensé à la création d'un administrateur qui efface tous les joueurs du jeu. Il suffit simplement d'utiliser la méthode `clear` de la classe Map pour cela.

Et enfin une fois le jeu mis à jour, on peut envoyer ses éléments au(x) client(s) avec la méthode `emitValuesToClient` (qui veut simplement dire émettre les valeurs au client). La méthode `emitValuesToClient` envoie un paquet (qui est un `Objet` clé/valeur) contenant trois propriétés : le joueur, les ennemis et les items présents dans le jeu de la même forme que le paquet envoyé dans `addNewPlayer`. Ici, on parcourt la liste des joueurs et envoie pour chaque joueur, les informations, nécessaires pour le client, du joueur (représentant le client), des ennemis (on récupère les ennemis à l'aide de la fonction `getEnemies` qui crée un tableau avec la liste des informations de tous les joueurs sauf le joueur passé en paramètre), et des items.

Ainsi, la classe `Game` comporte l'ensemble des éléments de notre jeu en utilisant les classes que nous allons voir dans les prochaines sections (`World`, `Item` et `Player`). Il va permette de communiquer avec le jeu côté client (*game.js* côté client) à l'aide des fonctions `emit` et `on` (par l'intermédiaire de *server.js*). Il s'agit du moteur de jeu.

4.2.2 World (world.js)

La classe `World` correspond à notre plateau de jeu où chaque case correspond à un élément. Un élément peut soit être une case vide, un item ou une

partie du corps d'un serpent (représenté par `World.TILES_ID` où on associe le nom de l'élément à un identifiant). Ainsi, cela va nous permettre de détecter plus facilement les collisions entre les éléments de notre jeu (et d'éviter de passer par plusieurs itérations). Pour cela on doit faire des fonctions qui insère un élément dans la cases du plateau et qui vérifie si la case est occupée par un élément.
Pour insérer le serpent (ou pour être plus précis la tête du serpent vu que le corps est déjà inséré) ou un objet, on remplace seulement la case actuelle par l'identifiant du serpent ou de l'objet en question (qui se trouve dans `World.TILES_ID`). Pour le serpent, on pense à retirer la queue du serpent du plateau de jeu si le serpent a bougé et si le serpent n'est pas "UNLIMITED" (nous définirons cela lorsqu'on évoquera la classe Snake). Et pour l'objet, il faut aussi penser à ajouter en plus de l'identifiant, le nom de l'Item afin de pouvoir appliquer le bonus/malus en question au serpent. Les fonctions gérants l'insertion des éléments sont `insertSnake` pour l'insertion d'une partie du corps d'un serpent et `insertItem` pour un item.
Concernant la détection de collision, il s'agit surtout de vérifier si la tête d'un serpent ne touche pas un élément du jeu (serpent ou un item). Il suffit donc de vérifier si la case de la tête est déjà occupé par une autre case que vide. Si ce n'est pas le cas, on retourne faux sinon cela dépendra de l'élément en question. Par exemple pour `checkSnakeTile` qui vérifie si la tête du serpent est en collision avec une partie d'un serpent, la fonction retourne true si c'est le cas . Cependant pour `checkItemTile` qui vérifie si la tête du serpent est en collision avec un item, la fonction return un objet Item (l'objet Item qui est en collision) si c'est le cas.
C'est ici qu'on va pouvoir également connaître les dimensions de notre jeu, et donc agrandir (ou non) notre jeu mais également donner de manière aléatoire les coordonnées d'un serpent ou d'un item lors de leur apparition avec `spawnSnake` et `spawnItem`.

4.2.3 Snake (snake.js)

La classe `Snake` correspond à l'entité que nous contrôlons qui représente un serpent. Le serpent possèdera plusieurs propriétés : des coordonnées x, y, une direction, une valeur booléenne afin de savoir si il est en vie, un score, une taille, et un corps. Dans cette classe, on s'occupe seulement du déplacement et de la détection de collision du serpent. La méthode `move` gère le déplacement du serpent : sa position change en fonction des valeurs x et y de la direction, on replace le serpent dans le plateau de jeu si il dépasse la limite du World, et on s'occupe de modifier les coordonnées de la tête du serpent. En réalité, ici les coordonnées x, y correspondent à la position de la tête du

serpent et le corps (`body`) correspond à la tête et le corps du serpent (ainsi si nous écrivons `body[0]` cela revient à écrire la tête du serpent, mais trouvons plus simple d'écrire en plus des coordonnées x et y pour la tête afin que ça soit plus lisible).
De plus, si on décide d'avoir un corps illimité et donc de grandir à chaque déplacement, nous pouvons mettre la variable `Snake.UNLIMITED_BODY` à true (qui permettra d'augmenter la taille du serpent indéfiniment).
La méthode `collision` permet de détecter les collisions avec les autres éléments du jeu. Ainsi il appellera les fonctions `checkSnakeTile` (si il y a détection, alors le serpent meurt) et `checkItemTile` (si il y a détection, alors on applique l'effet de l'objet sur le serpent) que nous avons vu précédemment.

4.2.4 Player (player.js)

La classe représente un joueur dans le jeu. Le joueur incarne un serpent et possède également d'autres attributs comme le socket (ou même un nom même si nous n'avons pas eu le faire, ou plutôt d'exploiter de manière efficace cet attribut). C'est les valeurs de ces objets là que nous allons transmettre au client (pas toutes, mais les plus essentielles), mais c'est également cette classe qui va pouvoir exploiter les entrées des clients. En effet, la méthode `update` va nous permettre de modifier la direction du serpent à l'aide des entrées du client (`keyboardState` donné en argument de la fonction). Dans cette méthode, on verra d'abord, si les entrées ne sont pas incohérents (par exemple si l'utilisateur entre à la fois flèche du haut et flèche du bas ou encore si le serpent se déplace vers le haut et que l'entrée souhaite qu'on se déplace vers le bas) à l'aide de la méthode `impossibleMove` qui retourne vrai si le mouvement est incohérent, faux sinon. Si la méthode retourne vrai alors on ne met pas à jour la direction du serpent.
Pour la modification de la direction, au lieu de faire plein de conditions pour trouver l'entrée du client, nous avons décidé de créer un objet clé/valeur `Player.DIRECTIONS` qui correspond aux différentes directions possibles du joueur. Cette objet aura les mêmes propriétés que le `keyboardState`. Du coup, il suffit de parcourir l'objet `keyboardState`, de vérifier quelles sont les touches qui sont entrées par le client, de prendre la valeur de `Player.DIRECTIONS` avec la clé de `keyboardState`.
Et enfin, nous avons également la méthode `getDir`, utile pour l'affichage du serpent et donc pour le client. Elle retourne la direction actuelle du serpent (chaîne de caractère) en déduisant des coordonnées x, y de la direction. Par exemple, si nous nous déplaçons de -1 en x et de 0 en y, cela retournera "left" car il se déplace à gauche. Cela nous permettra de gérer l'animation de la tête de notre serpent.

4.2.5 Item (item.js)

La classe `Item` représente un item/objet dans le jeu. Cela peut correspondre à un bonus ou un malus pour le joueur. Chaque item aura des coordonnées x et y, un nom pour savoir quelle(s) effet(s) appliqué(s) sur le serpent, et une valeur booléenne qui permet de savoir si l'objet a été consommé ou pas. Si l'objet a été consommé alors on retire l'objet du jeu (un peu dans la même que le serpent et la valeur booléenne `alive`). On a défini également plusieurs variables globales : pour les noms des objets, la valeur des objets, le temps d'apparition des objets, et le nombre d'objets maximum dans le jeu. Nous avons également une liste de tous les noms d'objets `ITEMS_NAME`. Concernant les méthodes, à chaque fois que le joueur va rentrer en collision avec un objet, il appellera la méthode `apply` qui elle-même une autre méthode selon le nom de l'objet (si le serpent mange une pomme rouge `APPLE_ITEM` alors on appellera la méthode `growSnake` qui incrémentera la taille du serpent, si il mange une pomme empoisonnée `POISON_ITEM` alors le serpent meurt et enfin si il traverse des pièces alors son score augmentera) : on applique ainsi un effet selon l'objet.
Puis nous avons également deux autres fonctions statiques permettant de générer de nouveau item dans notre jeu. `chooseRandomItem` choisit un nom d'item au hasard dans la liste `ITEMS_NAME` qui va ensuite être en relation avec l'autre fonction statiques `endOfSpawnTime`. `endOfSpawnTime` retourne vrai si le compte à rebours pour l'apparition s'est terminée et si le nombre d'objet maximum n'est pas atteint (et relance le compte à rebours) sinon retourne faux. Cela va nous permettre de créer un nouvel item dans la classe Game à des intervalles de temps réguliers.

Voici tout le contenu de notre serveur. Ainsi nous remarquons qu'il réalise une grande partie du travail du programme car désormais il nous reste plus qu'à voir d'où vient les entrées claviers du client (Controller) et l'affichage de nos éléments (View). Pour cela, nous verrons dans la section suivante les fichiers côté client.
Remarque : Nous avons également dans le dossier global, tous les utilitaires, c'est-à-dire les fonctions des fonctions qui ne sont pas directement en relation avec notre projet mais qui sont utile à certains moments dans le dossier src. Elle possède des fonctions telles que `getRandomFloor` (qui permet de récupérer un entier au hasard), `isBound` (vérifier si un point appartient bien à la surface d'un rectangle) ou encore `getRandomColor` (choisir une couleur au hasard...)

4.3 Client

Les actions côté client sont dans le dossier *public*. Le dossier public contient l'ensemble des fonctionnalités s'occupant de l'affichage de nos éléments du jeu transmis par le serveur et de l'envoi des entrées du client au serveur. Il comprend le fichier `index.html`, un dossier *img* qui regroupera l'ensemble des images du jeu, un dossier *css* pour la mise en forme de notre page mais surtout le dossier *js* qui rassemble tous nos scripts. Le client commence déjà par lire le contenu de la fonction `main` du fichier *client.js*. Le `main` va d'abord se connecter avec le serveur à l'aide `io.connect` afin que les deux parties communiquent (comme nous l'avons dit précédemment, le `io.connect` renvoie un objet Socket qui permet d'identifier et de différencier le client des autres). Une fois connectée, on initialise le jeu en instanciant un objet `Game` (cette fois-ci côté client donc pas les mêmes fonctionnalités) et lancer la fonction `handleEvents` du fichier *events*.js qui va gérer les entrées. On va donc pour cela créer une classe `Game` mais cette fois-ci côté client qui se chargera de récupérer les informations du serveur et de réaliser les actions côté client. Mais nous avons également une classe `Player` s'occupant d'actualiser les données du joueur pour l'affichage et une classe `Display` qui contiendra toutes les informations et fonctions d'affichage et de mise en forme. Et enfin, le fichier *events.js* qui n'a pas de classe mais qui permet de récupérer les entrées du client.

4.3.1 Game (js/game.js))

La classe `Game` possède le socket du client permettant de l'identifier, un élément qui servira d'affichage, le joueur représentant le client en question, les ennemis, les objets du jeu et le frameId qui permettra d'arrêter la `requestAnimationFrame`. Lorsqu'on initialise le jeu (fait lors du `main`) avec la méthode `init` de Game, on instancie un objet Display et on l'initialise. En plus de ça, on envoie un message au serveur pour qu'il rajoutait un nouveau au jeu côté serveur mais également recevoir un paquet de la part du serveur pour initialiser les valeurs du joueur, des ennemis et des objets.
Pour générer les éléments du jeu, nous utilisons la méthode `initGameValues` (appelé dès que le joueur est créé) qui comme son nom l'indique initialise la valeur du jeu (côté client, nous le rappelons). Comme nous l'avons vu la fonction `emitValuesToClient` du `Game` côté serveur, le serveur transmet un bloc de données composé de 3 éléments : le joueur contrôlé par le client (propriété : `"player"`), les ennemis (propriété : `"enemies"`) et les objets (propriété : `"items"`) présents dans le jeu. Le serveur transmet seulement :

- le corps (affichage du serpent sur le plateau), le score (affichage du

score sur le tableau de scores) et la direction (animation de la tête du serpent sur le plateau) pour les ennemis.
- seulement la tête du serpent vu qu'il vient d'apparaître, la taille du serpent et la direction. Et cette fois-ci on instancie un objet Player qui s'occuper de tous algorithmes pour permettre un affichage cohérent.
- une liste de tous les items présents sur le plateau de jeu

Une fois le jeu initialisé, on peut désormais lancé la boucle principale du jeu côté client avec la méthode `start` qui elle-même `run`. `run` va utiliser `requestAnimationFrame` qui permet d'exécuter du code à chaque rendu d'image du navigateur.
Ainsi à chaque rendu :

- nous enverrons au serveur les entrées clavier contenus dans la variable `KEYBOARD_STATE` (gérée dans events.js) du client
- nous mettrons à jour les éléments de notre jeu avec les informations données par notre serveur.
- La méthode `setGameValues` s'occupe justement de mettre à jour les éléments à l'aide des données reçus par le serveur (passées en paramètre de la fonction). Ici, le serveur nous donne seulement la tête du joueur car nous avons déjà le corps de ce dernier donc nous pouvons très bien dessiner le serpent en ayant seulement sa tête car la classe Player s'occupe de mettre à jour le corps et la tête du joueur. Cependant, nous n'avons pas eu le temps de réaliser la même chose pour les ennemis, du coup nous envoyons tous le corps à chaque fois, ce qui est assez lourd (l'idéal aurait été de faire la même chose que pour le joueur client). Cette méthode nous donne également la liste des items.
- nous afficherons les différents éléments du jeu reçus par le serveur à l'aide de la méthode `render`. Cette méthode se contente d'appeler les différentes méthodes de la classe `Display` du fichier display.js. Elle nettoie la zone d'affichage, affiche le fond d'écran du jeu, affiche les serpents, le tableau des scores et les objets du jeu (nous verrons les méthodes utilisées lorsque nous allons traitées la classe `Display`).

4.3.2 Player (js/player.js)

La classe représente les serpents, elle est différente de la classe `Player` côté serveur car il possède beaucoup moins de méthodes et d'attributs. En effet, cette classe se contente seulement de récupérer les valeurs du joueur transmis par le serveur. Elle possède donc seulement les coordonnées x, y du serpent (la tête), la direction du serpent, le score du joueur et la taille du serpent. La méthode `update` va se contenter de mettre à jour les valeurs

des attributs de la classe Player avec les données passées en paramètres (qui viennent du serveur). Ainsi, nous n'avons pas besoin d'envoyer tout le corps au client, il suffit seulement d'envoyer les coordonnées de la tête car on stocke les anciennes coordonnées de la tête (qui font désormais parties du corps). Nous pouvons également rajouter d'autres algorithmes dans l'avenir pour faciliter l'affichage et éviter de faire transiter trop de données entre le client et le serveur.

4.3.3 Display (js/display.js)

Comme son nom l'indique la classe `Display` s'occupe de l'affichage des différents éléments du jeu. Il comporte un canvas (élément HTML permettant de dessiner), un tableau des scores, une image et l'ensemble des images. Tout d'abord, on se charge d'initialiser l'objet `Display` à l'aide de la méthode `init` qui va permettre de créer un élément HTML qui est un "canvas" pour notre zone de dessin, de lui attribuer un ID, une longueur et une largeur, un élément HTML qui est "ul" (une liste non ordonnée) pour notre tableau des scores et enfin on charge toutes les images de notre jeu (situé dans *img*).
Une fois initialisée, nous devons nous occuper de l'affichage de 4 éléments :

- le fond d'écran qui va juste être affiché sur tout l'écran.
- l'item qui est situé dans le fichier *img/items.png*. Il s'agit de plusieurs mis côte à côté donc il faudra utiliser plus de paramètres pour la fonction `drawImage` du canvas afin de récupérer seulement la zone de l'image qui nous intéresse.
- De même pour les serpents, nous aurons besoin de la direction du serpent afin de connaître la zone de l'image (*img/snakes.png*) à choisir. L'image de la tête du serpent est différente selon la direction du serpent, or le corps possèdent toujours la même image. De plus, pour différencier le joueur client des ennemis, nous avons choisi de représenter le serpent client en vert et les ennemis en jaune.
- le tableau des scores avec comme premier élément de la liste, le score du joueur et le reste le score des ennemis dans l'ordre décroissant. Pour cela, on compare les différents score des ennemis et on place seulement les N premiers scores les plus élévés (N étant défini comme variable globale : `TOP_SCORERS`)

4.3.4 Events (js/events.js)

Le fichier *events.js* regroupe l'ensemble des entrées claviers du joueur. Ces dernières sont tous situées dans la fonction `handleEvents` qui se charge de vérifier si les évènements "keydown" (touche de clavier pressée) et "keyup"

(touche de clavier relachée) sont déclenchés. Si l'évènement "keydown" est déclenché, on appelle la fonction `onKeydown` qui mettra à jour l'objet `KEYBOARD_STATE` en remplissant ses valeurs par true. Par exemple si le joueur appuie sur la flèche du haut ou sur "x", nous aurons pour la propriété "up", la valeur true. Si l'évènement "keyup" est déclenché, on appelle la fonction `onKeyup` qui mettra à jour l'objet `KEYBOARD_STATE` en remplissant ses valeurs par false. Par exemple si le joueur appuie sur la flèche du haut ou sur "x", nous aurons pour la propriété "up", la valeur false. Nous traitons 4 états du clavier, la gauche ("ArrowLeft" ou "q"), la droite ("ArrowRight" ou "d"), le haut ("ArrowUp" ou "z") et le bas ("ArrowDown" ou "x").

5 Améliorations possibles

Malheureusement nous n'avons pas eu totalement le temps de nous concentrer sur ce projet vu que nous avons eu un autre projet à réaliser en parallèle. Du coup, nous pensons qu'il y a plusieurs éléments que nous pouvons améliorer ou rajouter. Voici une liste non exhaustive des améliorations possibles :

- L'agrandissement des dimensions de notre plateau de jeu (côté serveur) pour avoir un repère plus grand. Ce qui aura du coup des répercussions sur le mouvement du serpent et également sur l'affichage côté client. On peut même penser à créer un plateau de jeu dynamique qui s'agrandirait au fur et à mesure qu'un serpent se rapproche du bord.
- Le nom du joueur et également une base de données (Model) pour pouvoir placer les meilleurs sur une base de données avec le nom des joueurs. Mais cela requiert un peu de temps que prévu donc nous nous sommes pas focalisés sur ça.
- la couleur de notre serpent. Comme nous vous l'avions dit précédemment, nous utilisons deux images pour le serpent. Une image du serpent en vert (pour le client) et une en jaune (pour les autres clients). L'idée est de donnée une couleur à notre serpent et de pouvoir appliquer des modifications sur l'image du serpent à l'aide de la couleur du serpent en jouant sur les propriétés de l'image (comme le TP2). Comme ça, nous aurons des serpents de différentes couleurs mais seulement une seule image.
- Nous avons optimisé les échanges de données pour le joueur client mais nous devons également la même chose pour les ennemis afin de faire circuler le moins de données possibles. L'idée est de faire la même étape pour les ennemis que pour le joueur.

6 Conclusion

Ce projet nous a permis d'acquérir de nouvelles connaissances et d'être plus rigoureux. En effet, il nous a aidé à bien organiser notre code, à bien le présenter (à l'aide de commentaire) et donc à le rendre plus maintenable au fil des années.