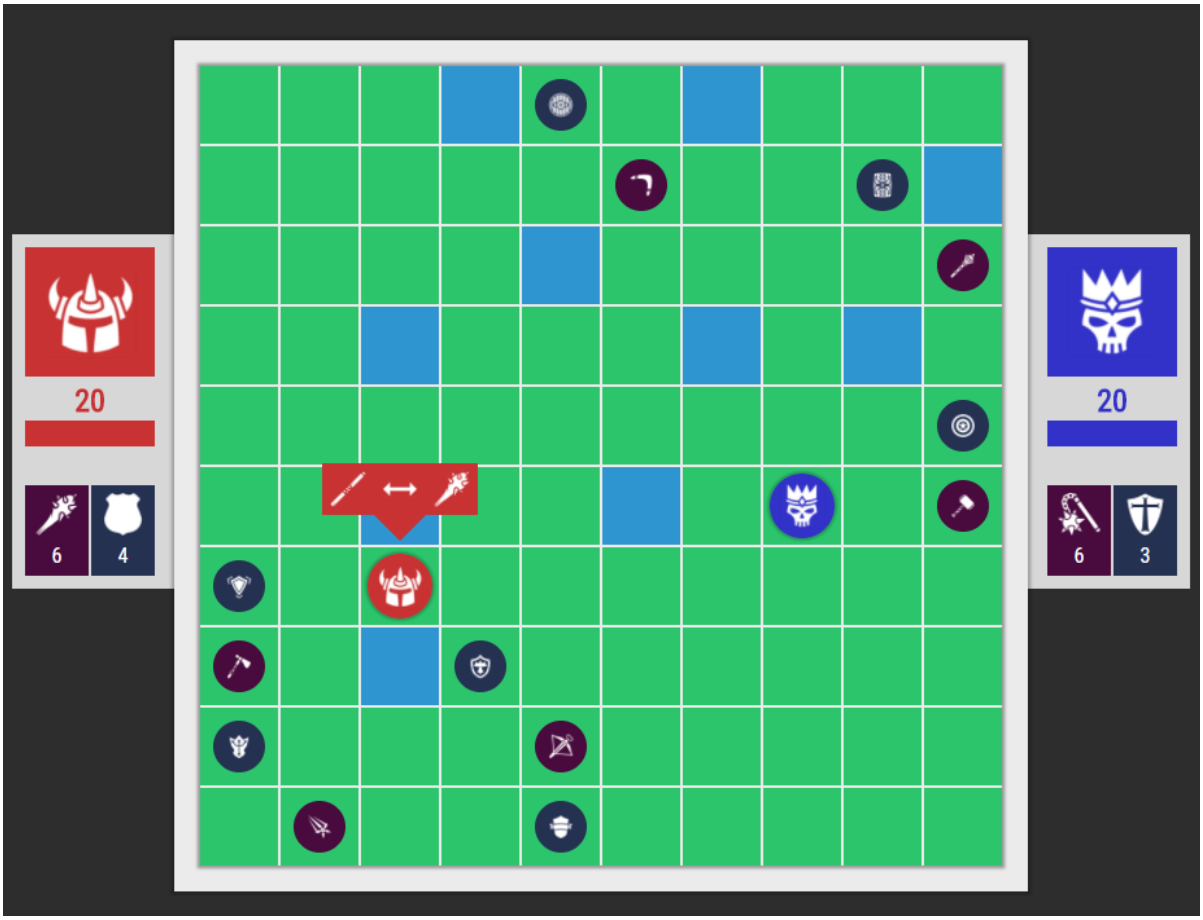
Boardgame

L'objectif de ce travail est de mettre en pratique les connaissances de bases apprises sur Javascript (via OpenClassroom, Codecademy, SoloLearn).



Création du plateau de jeu

Pour créer le plateau, j'ai ajouté un conteneur HTML en *display: grid* et réglé la propriété *grid-template-columns* de manière à créer un tableau de 10x10 cases.

En JS, j'ai créé un objet *Square* pour les cases avec comme propriétés une id, une position, et 4 autres propriétés pour définir si la case est occupée (1) ou non (0) par un joueur, un obstacle, une arme ou un bouclier. La fonction d'initialisation automatise les deux premières propriétés, les autres étant à 0 par défaut. J'ai ensuite utilisé une boucle pour générer les 100 objets.

J'ai créé un objet *Player* pour les joueurs avec comme propriétés une id, une position, l'attaque (*attack*) et la défense (*defense*), les points de vie (*hp*) et les armes (*weapon*) et boucliers (*shield*) équipés. La fonction d'initialisation automatise les deux premières propriétés, les autres étant prédéfinis. La position des joueurs étant définis, j'ai créé un tableau à deux valeurs correspondant aux positions initiales des deux joueurs. J'ai utilisé une boucle pour générer les deux objets en définissant la propriété de position à partir du tableau.

Génération des éléments et positionnements aléatoires

Il s'agissait alors de positionner aléatoirement sur le plateau de jeu des éléments : les obstacles (*obstacles*) qui bloquent le chemin d'un joueur et les équipements (armes et boucliers) que peuvent ramasser les joueurs. J'ai donc créé un système de positionnement aléatoire. Celui-ci devait respecter certaines règles :

- deux éléments ne peuvent occuper la même case ;
- deux éléments de même type ne peuvent être sur des cases adjacentes ;
- les éléments doivent être à au moins deux cases de distance de la position initiale des joueurs et les équipements ne peuvent pas être accessibles dès le premier tour.

Pour réaliser cela, j'ai créé trois tableaux qui répertorient les cases restantes attribuables aléatoirement : *availablePositionsForObstacles*, *...ForWeapons* et *...ForShields*.

J'ai ensuite créé une boucle pour générer chaque objet qui intègre des fonctions de positionnement. Ci-dessous un exemple avec les obstacles.

Une boucle *for* est mise en place.

Un objet *Obstacle* est créé.

La fonction *elementsPosition*(*availablePositionsForObstacles*) est utilisée pour choisir une valeur aléatoire dans le tableau entré en paramètre.

La fonction *removeFromAvailablePositions*(*position*, *availablePositionsForObstacles*, 1) est utilisée pour supprimer des trois tableaux *availablePositionsFor...* la position (définie précédemment) et supprimer du tableau entré en deuxième paramètre les positions à une (troisième paramètre) case de distance.

> La fonction *findSurroundingPositions*(*position*) est utilisée au sein de la fonction précédente pour identifier les cases entourant la position entrée en paramètre. La fonction teste les 8 positions autour d'une case (y compris les diagonales) et indique soit la position de la case trouvée, soit *null* s'il n'y a pas de case (hors du plateau). Les cases sont analysées en partant de celle se situant au-dessus à gauche puis dans le sens de lecture et la fonction retourne un tableau de huit nombres. Par exemple :

Position :

37

Analyse :

26	27	28
36	37	38
46	47	48

Valeurs retournées :

```
[26, 27, 28, 36, 38, 46, 47, 48]
```

La fonction d'initialisation d'objet *initObstacle*(*"OB" + i*, *position*) est utilisée pour générer les propriétés : la première correspond à l'id (i étant la variable d'incrémentation de la boucle) et la seconde à la position.

L'obstacle est intégré au tableau des objets (*obstacles*).

La propriété *obstacle* de l'objet *square* correspondant à la position de l'obstacle prend la valeur 1 pour indiquer qu'un obstacle s'y trouve.

La même boucle est utilisée pour les armes et les boucliers avec en plus la propriété de nom (*name*) de l'équipement et les propriétés d'attaque ou de défense (aléatoire).

Intégration des objets dans le DOM

J'ai créé une fonction *draw*(array, object, delay) pour générer les éléments du DOM. Elle prend en paramètre le tableau d'objets (par exemple *weapons*), l'objet (*weapon*) et le délai. Avec une boucle *for*, les cases sont intégrées dans la *grid* tandis que les autres éléments sont intégrés dans les cases (en position *absolute*). Une id est appliquée à chaque élément correspondant à l'id de l'objet. Si le paramètre *delay* est à 1, un délai d'attente est ajouté entre la création de chaque élément grâce à une fonction *sleep*(*ms*).

Déplacement des joueurs

Le déplacement des joueurs se fait en deux temps. D'abord les cases accessibles sont calculées et affichées. Une fonction est ajoutée au clic sur ces cases et permet au joueur de se déplacer.

Identification des cases accessibles

J'ai créé une fonction *findReachablePositions*(*player*, *position*) qui permet de définir les cases accessibles d'un joueur à partir d'une position.

Pour chaque direction (haut, bas, gauche et droite), une boucle *while* est utilisée et s'appuie sur la précédente fonction *findSurroundingPositions*(*position*). Le programme est le suivant.

Pour définir les cases accessibles vers le haut, si les propriétés *player* ou *obstacle* de l'objet *square* dont la position est celle de la deuxième valeur du tableau renvoyée par la fonction *findSurroundingPositions*(*position*) sont à 0, cela signifie que la case au-dessus du joueur n'est pas occupée par un autre joueur ou un obstacle et qu'il peut s'y déplacer. Le test est alors réalisé sur cette case accessible, pour voir si celle encore au-dessus est accessible. La boucle s'arrête quand une case n'est pas accessible ou quand trois cases ont été testées.

À chaque case accessible, un objet *reach* est créé et prend pour propriété *range* le nombre d'itérations (permettant de savoir si la case est à 1, 2 ou 3 cases de la position du joueur).

J'ai créé une fonction *isLeadingToFight*(player, position) qui va tester, à chaque fois qu'une case accessible est trouvée, si elle se trouve à côté d'un autre joueur, ce qui signifie une case enclenchant un combat si le joueur s'y déplace. Cette fonction s'appuie sur la précédente fonction *findSurroundingPosition*(*position*) pour tester si les cases correspondant aux valeurs renvoyées par cette fonction (les valeurs au position 2, 4, 5 et 7 du tableau soit haut, gauche, droite, bas) ont en propriété *player* la valeur 1 (donc qu'un joueur est sur la case). Si c'est le cas, la propriété de combat de l'objet (*fight*) identifiant la case accessible prend la valeur 1.

Les objets correspondant aux cases accessibles sont ensuite intégrés dans le DOM, prenant une classe *reach* ou *reach-fight* selon la valeur de la propriété *fight*.

Déplacement

Lorsque le joueur clique sur une des cases accessibles, la fonction *moveToPosition*() s'enclenche. L'id de l'objet *reach* correspondant à l'élément dans le DOM est retrouvée grâce à la promiscuité entre les deux id. Une boucle *while* est utilisée. Le joueur se déplace case par case jusqu'à atteindre la valeur *range* de l'objet *reach* correspondant. La fonction *sleep*(*ms*) est utilisée pour créer une pause entre chaque déplacement d'une case.

Sur chaque case par lequel passe le joueur, un test est réalisé pour savoir si cette case contient une arme ou un bouclier, ce qui lance la fonction *switchItem*() (cf. infra).

Si la case d'arrivée est une case de combat, la fonction *startFight*() (cf. infra) est lancée. Sinon, le joueur suivant joue.

Changement d'équipement

Le changement d'équipement se fait lorsqu'un joueur passe par une case contenant une arme ou un bouclier. Un test est réalisé pour trouver dans le tableau des armes ou boucliers l'objet correspondant. L'équipement du joueur (propriété *weapon* ou *shield*) est modifié et le DOM est mise à jour en fonction.

Des éléments graphiques sont ajoutés pour indiquer le changement d'équipement à l'utilisateur. La présentation dépend de si le joueur n'avait pas d'équipement avant (auquel cas une icône du nouvelle équipement est affichée) ou s'il en avait (auquel cas les icônes de l'ancien et du nouvel équipement sont affichées).

La propriété *position* de l'équipement trouvé est définie à -1 pour le retirer du plateau. Celle de l'ancien équipement, le cas échéant, prend la valeur de la position actuelle (le joueur prend le nouvel équipement et pose l'ancien).

Les fonctions *showItemsLevel*() et *hideItemsLevel*() permettent de voir/cacher le niveau d'un équipement en passant le curseur au-dessus.

Combats

Lorsqu'un joueur arrive sur une case de combat, le combat s'enclenche. La fonction *startFight*() gère la début du combat, c'est-à-dire les propositions d'actions pour le joueur engagé. Les éléments graphiques correspondant aux actions (contre-attaquer ou parer) sont affichés. Un test est réalisé pour savoir d'où l'attaquant arrive et placer les icônes en fonction. Par exemple, si l'attaquant arrive de la gauche, les icônes d'actions du défenseur seront affichées à droite du défenseur. Si l'attaquant arrive par le dessus, les icônes seront affichées sous le défenseur.

La fonction *chooseAction*(*action*) démarre quand le joueur choisit de contre-attaquer ou de parer. Si le défenseur contre-attaque, les dégâts de chaque joueur sont appliqués.

S'il se défend, les dégâts sont calculés (dégâts de l'arme réduits des défenses du bouclier, minimum zéro). Les équipements sont abîmés, ce qui réduit l'attaque des armes et la défense des boucliers. Si un équipement arrive à zéro, il est détruit et le joueur repart avec l'équipement initial (aucun équipement, ce qui donne des statistiques de 1 en attaque et en défense).

Des icônes sont affichées pendant quelques secondes pour montrer les dégâts subis. Un test est réalisé en amont. En effet, le joueur qui arrive sur la case de combat attaque en premier. S'il tue l'autre joueur, celui-ci ne peut contre-attaquer, le joueur attaquant ne subit donc pas de dégâts. En fonction des points de vie restant, un deuxième round commence ou, si un des joueurs n'a plus de vie, l'écran de fin désigne le vainqueur.