Master 1 GIL -
Animation Lead -
Mise en place des animations et notification dans un jeu
Agora V3-1

Groupe 1 Agora V3-1 : Partie bibliothèque de jeux

19 mai 2024

Version	1
Date	19 mai 2024
Rédigé par	DUCROQ Yohann

Mises à jour du document

Version	Date	Modification réalisée
1	19 mai 2024	Création du document

Table des matières

1 Objet du document

Ce document a pour but de présenter la procédure de mise en place d'une animation dans un jeu sur la plateforme Agora ainsi que l'implémentation de notifications sous la forme de pop-up.

2 Terminologie

Animation : Déplacement visuel d'un élément d'un jeu lié à une ou plusieurs actions qui viennent de se produire.

Notification : Texte affiché brièvement pendant une partie afin d'informer l'utilisateur sur une opération réalisée pendant la partie en cours.

3 Animation

3.1 Prérequis

Avant de mettre en place une animation, il est nécessaire de déterminer si votre jeu en développement a besoin d'animations pour permet aux joueurs de comprendre l'état actuel d'une partie et pourquoi tel action a pu avoir lieu et comment.
Pour vous aider dans cette démarche, le plus simple est de jouer une partie réelle de votre jeu en développement et de voir les déplacements que les éléments font au cours de la partie. Si un élément fait seulement de petit mouvement d'une seconde, vous pouvez négliger l'implémentation de cette animation.

Avant de créer votre première animation, vous devez vérifier que le fichier *animationQueue.js* est toujours présent dans le dossier *assets/ressourcesGames/utils*. Ce script, utilisé par tous les jeux avec des animations, permet d'avoir une file FIFO afin d'avoir un ordre d'exécution des animations qui respecte le séquençage entre les animations : une animation est exécutée à la fin de la précédente.

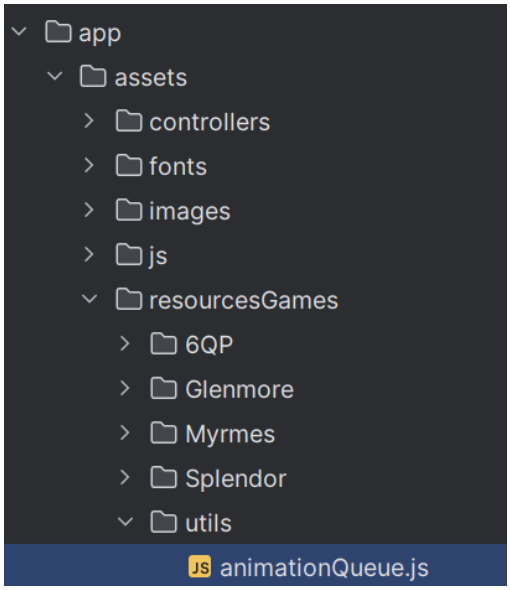

FIGURE 1 – Emplacement du script animationQueue.js

Pour chaque jeu implémentant des animations, vous devez ajouter le fichier JavaScript *animation.js* qui va contenir les fonctions réalisant chacune une animation. Vous pouvez le mettre dans le dossier de votre jeu en question situé dans *assets/ressourcesGames* :

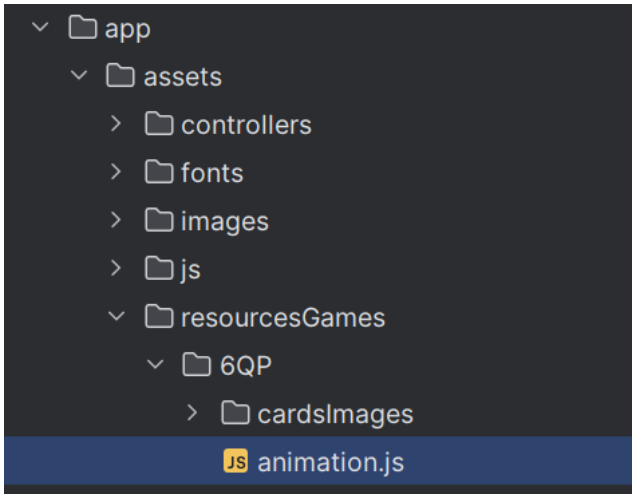

FIGURE 2 – Exemple de l'emplacement du script animation.js pour le 6 Qui Prend

Ensuite, dans le fichier *index.html.twig* de votre jeu, vous devez y intégrer les deux scripts précédents dans une balise script pour pouvoir les utiliser :

```
{# ANIMATION MANAGEMENT #}
<script type="text/javascript" src="{{ asset('resourcesGames/utils/animationQueue.js') }}"></script>
<script type="text/javascript" src="{{ asset('resourcesGames/6QP/animation.js') }}"></script>
```

FIGURE 3 – Intégration des scripts pour les animations dans index.html.twig

Pour éviter que le rafraîchissement d'un élément du jeu casse une animation, chaque événement **onmessage** reçu d'un **EventSource** doit être traité comme une animation en ajoutant son traitement dans une fonction anonyme à fournir en paramètre dans la fonction **addToQueue** de la queue d'animation, sans oublier l'appel à la fonction **executeNextInQueue** obligatoirement à la fin du code de votre fonction anonyme :

```
{% set path = path('app_game_show_sixqp', {'id': game.id}) ~ 'mainBoard' %}
const eventSourceMainBoard = new EventSource("{{ mercure(path)|escape('js') }}");

eventSourceMainBoard.onmessage = event => {
    animationQueue.addToQueue(() => {
        let mainBoard = document.getElementById('mainBoard');
        mainBoard.innerHTML = event.data;
        animationQueue.executeNextInQueue()
    })

}
```

FIGURE 4 – Example du rafraichissement du plateau principal du 6 Qui Prend

Dans ce même fichier, vous devez y ajouter l'élément suivant, au même niveau que les **include** twig :

```
<div id="animationContainer" class="hidden absolute top-0 left-0 w-full h-full bg-black
    bg-opacity-0 z-50"></div>
```

Celui-ci vous sera utile pour toute animation de déplacement d'un élément d'un point A vers un point B. Cet élément est invisible, mais sera affiché au-dessus de tous les éléments du jeu et permettra d'y cloner les éléments à déplacer le temps d'une animation uniquement.

Dernier ajout du côté du fichier *animation.js*, ajoutez-y à la fin les trois lignes suivantes :

```
let animationQueue = new AnimationQueue();
let animationContainer = document.getElementById('animationContainer');

animationQueue.executeNextInQueue();
```

La variable **animationQueue** correspondra à la queue d'animation unique pour tout jeu, et **animationContainer** servira donc pour y cloner les éléments en mouvement. La dernière ligne permet de lancer l'exécution de la queue d'animation qui attendra en arrière-plan les prochaines animations à exécuter.

3.2 Ajouter une animation

3.2.1 Partie Frontend

La création d'une animation débute dans le fichier *animation.js* de votre jeu : il vous suffit de créer une fonction portant un nom qui définit le rôle de cette animation et de fournir, si nécessaire, les éléments concernés directement par cette animation, en paramètre. Le plus commun serait de fournir leurs id pour récupérer les éléments avec la fonction **document.getElementById**.

Ensuite, vous pouvez commencer le corps de votre fonction par la création d'un objet **Promise** : le but de celui-ci est de renvoyer une promesse pour exécuter une animation et s'assurer que le code attend la fin de cette animation avant de pouvoir continuer. Le cœur de cette animation se trouvera donc dans une fonction anonyme qui sera fournie en paramètre de l'objet **Promise** créé. Les animations sont réalisées de matière native (sans framework) avec un appel à la fonction **animate** sur un objet **Element**.

L'appel à la fonction **animate** retourne un objet **Animation** qui a une propriété **finished**. Cette propriété est elle-même une **Promise** qui est résolue lorsque l'animation est terminée. Il faut ensuite utiliser la méthode **then** sur cette propriété, pour exécuter sois la suite du code, s'il y a une suite dans cette animation, sois la fin de l'animation en appelant **resolve()** qui va donc résoudre la Promise créée initialement.

L'exemple suivant concerne une animation du jeu 6 Qui Prend qui fait translater une ligne de carte en disparaissant :

```
function translateRow(rowid) :Promise<?>  {
    return new Promise(resolve => {
        let row :HTMLElement  = document.getElementById(rowid);
        row.animate(
            [
                {transform: "translateX(0px)", opacity: 1},
                {transform: "translateX(700px) scale(0.5)", opacity: 0},
            ],
            {
                duration: 2000,
                fill: "forwards",
            },
        ).finished.then(() => resolve())
    });

}
```

FIGURE 5 – Fonction translateRow pour le 6 Qui Prend

Si votre animation consiste à déplacer un élément d'un emplacement A vers un emplacement B, vous devez utiliser l'élément **animationContainer** en respectant les étapes qui suivent :

— Utilisez la fonction **getBoundingClientRect()** sur l'élément pour récupérer sa taille et sa positon exactes sur le **viewport** et sauvegarder le résultat dans une variable.

— Clonez l'élément et ajoutez-lui la classe **absolute** pour l'afficher sans contrainte sur sa position ou sa taille.

— Ajoutez-le à l'élément **animationContainer** et donnez lui la hauteur et largeur identique aux données de la variable contenant l'objet **DOMRect** retouné par la fonction **getBoundingClientRect()** .

A la fin de l'animation, n'oubliez pas de supprimer l'élément cloné dans **animationContainer**, celui-ci doit rester vide quand l'exécution d'une animation est terminée.

L'exemple ci-dessous correspond à l'animation du déplacement d'une carte d'une zone à une autre dans le jeu 6 Qui Prend :

```
function moveChosenCard(cardId) :void  {
    new Promise(resolve => {
        let cardFinalPositionElement :HTMLElement = document.getElementById('image_' + cardId);
        let cardElementInChosenCard :Element  = document.getElementById(cardId).firstElementChild;

        let chosenCardShape :DOMRect  = cardElementInChosenCard.getBoundingClientRect();
        let cardFinalPositionShape :DOMRect  = cardFinalPositionElement.getBoundingClientRect();

        let movingCardElement :Node  = cardElementInChosenCard.cloneNode(true);
        movingCardElement.id = 'movingcard_' + cardId;
        movingCardElement.classList.add('absolute');
        animationContainer.appendChild(movingCardElement);
        movingCardElement.height = chosenCardShape.height;
        movingCardElement.width = chosenCardShape.width;

        // Usefull to set a duration for the animation equal for every distance the translating movement will do
        let distance :number  = Math.sqrt((cardFinalPositionShape.x - chosenCardShape.x) ** 2 +
            (cardFinalPositionShape.y - chosenCardShape.y) ** 2);

        movingCardElement.animate(
            [
                {
                    transform: "translate(" + chosenCardShape.x + "px, " + chosenCardShape.y + "px)",
                    width: chosenCardShape.width + "px",
                    height: chosenCardShape.height + "px",
                },
                {width: chosenCardShape.width * 1.2 + "px", height: chosenCardShape.height * 1.2 + "px"},
                {
                    transform: "translate(" + cardFinalPositionShape.x + "px, " + cardFinalPositionShape.y + "px)",
                    width: cardFinalPositionShape.width + "px",
                    height: cardFinalPositionShape.height + "px"
                },
            ],
            {
                duration: distance / 0.3,
                fill: "forwards", // Reste a la positon final
            }
        ).addEventListener("finish", () :void  => {
            movingCardElement.remove();
            cardElementInChosenCard.remove();
            cardFinalPositionElement.classList.remove('invisible');
            resolve();
        });
        cardElementInChosenCard.remove()
    }).then(() :void  => animationQueue.executeNextInQueue());
}
```

FIGURE 6 – Fonction moveChosenCard pour le 6 Qui Prend

3.2.2 Partie Backend

Vos fonctions pour les animations doivent maintenant être appelées quand un message **Mercure** est réceptionné sur une route propre à cette animation.

Dans le fichier *index.html.twig* de votre jeu, ajoutez dans une balise **script** une variable Twig de type **Path** qui correspond à la route dédiée à votre animation. Ensuite, il vous suffit de créer une autre variable en JavaScript contenant un **EventSource** qui écoutera les messages provenant de la route définit juste avant, avec la fonction **onmessage**. Chaque fois que **onmessage** est appelée, on ajoute dans la queue d'animation avec **animationQueue.addToQueue**, la fonction JavaScript de l'animation créée. Si celle-ci renvoie une **Promise**, faite un appel à la fonction **then** pour exécuter **animationQueue.executeNextInQueue()** afin de passer à la prochaine animation.

Si votre animation a besoin d'information, comme l'id d'une carte à déplacer, n'oubliez d'utiliser la fonction **split** sur la propriété data de l'objet **MessageEvent** envoyé à chaque exécution de **onmessage**.

L'exemple ci-dessous correspond à la gestion des messages lié à une animation de récupération d'une carte du plateau par un joueur. Cette animation nécessite le nom de l'utilisateur et l'id d'une carte du plateau :

```
<script>
    {% set path = path('app_game_show_spl', {'id': game.id}) ~ 'animTakenCard' %}
    const eventSourceTakenCard = new EventSource("{{ mercure(path)|escape('js') }}");
    eventSourceTakenCard.onmessage = event => {
        animationQueue.addToQueue(() => {
            let username = event.data.split('__')[0];
            let devCard = event.data.split('__')[1];

            moveDevCard(devCard, username).then(() => animationQueue.executeNextInQueue());
        })
    }
</script>
```

FIGURE 7 – Gestion des événements liée à une animation.

La dernière partie se trouve dans le contrôleur de votre jeu : il faut créer une route Mercure, faire une fonction qui envoie un message sur cette route et appeler cette fonction au bon moment d'une action réalisée.

Prenons l'exemple de cette fonction qui publie un message pour réaliser l'animation de la prise d'un joker par un joueur :

```php
private function publishAnimTakenJoker(GameSPL $game, string $player): void
{
    $this->publishService->publish(
        route: $this->generateUrl( route: 'app_game_show_spl', ['id' => $game->getId()]).'animTakenTokens',
        new Response( content: $player . '__gold')
    );
}
```

FIGURE 8 – Fonction envoyant un message pour réaliser une animation.

Cette fonction utilise la méthode **generateUrl** pour construire un URL dédié à l'animation de la prise d'un joker avec l'identifiant du jeu comme paramètre. Ensuite, elle utilise le service de publication **publishService** pour envoyer un événement contenant cet URL et les informations sur le joueur qui a pris le joker.

Pour finir, vous avez plus qu'à appeler cette fonction dans votre contrôleur quand une action effectuée doit jouer cette animation. Pour garder une cohérence si l'animation se passe mal due, par exemple, à un ralentissement du serveur, une désynchronisation des éléments affichés avec la partie serveur, pensez à ajouter après une animation un appel la fonction responsable du rafraîchissement de la zone ou élément animé.

4 Notification

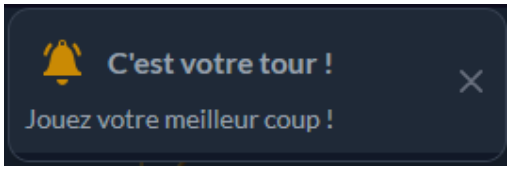

FIGURE 9 – Exemple d'une notification annonçant le tour du joueur.

4.1 Prérequis

Vous devez vérifier que le fichier *ingameNotification.js* est toujours présent dans le dossier *assets/ressourcesGames/utils*. Ce script, utilisé par tous les jeux avec des notifications, définit la classe GameNotification qui correspond à une notification.

Vous devez aussi vérifier la présence du fichier *notification.html.twig* dans le dossier *templates/Utils*. Ce fichier contient le modèle d'une notification (*notif_ template*) et les formes en SVG servant de logo aux notifications. Le contenu de ce fichier sera importé dans le fichier *index.html.twig* des jeux implémentant des notifications, en ajoutant le code suivant :

```
{% block notification %}
    <div id="notificationsContainer"
         class="absolute space-y-2 landscape:sm:w-[27%] portrait:w-[30%] landscape:lg:w-[20%]
                top-0 right-0 z-[60]">
    </div>
    {{ include('/Game/Utils/notification.html.twig') }}
{% endblock %}
```

Les notifications affichées seront dans l'élément **div notificationsContainer**.

Dans le contrôleur, ajoutez le code suivant :

```
private function publishNotification(Game $game, int $duration, string $message,
                                     string $description, string $iconId,
                                     string $loadingBarColor, string $targetedPlayer): void
{
    $dataSent = [$duration, $message, $description, $iconId, $loadingBarColor];

    $this->publishService->publish(
        $this->generateUrl('app_game_show_', ['id' => $game->getId()])
        .'notification'.$targetedPlayer,
        new Response(implode('_', $dataSent))
    );
}
```

Remplacez **Game** en paramètre par le type de votre jeu et **'app_game_show_'** par la route qui affiche votre jeu. Cette fonction envoie **$dataSent** au **$targetedPlayer** *(si vide, au spectateur de la partie)* de **$game** pour qu'il affiche la notification.

De retour dans *index.html.twig*, ajoutez le code suivant :

```
{# NOTIFICATION MANAGEMENT #}
<script>

    {% if isSpectator %}
        {% set path = path('app_game_show_', {'id': game.id}) ~ 'notification' %}
    {% else %}
        {% set path = path('app_game_show_', {'id': game.id}) ~ 'notification' ~ player.username %}
    {% endif %}
        const eventSourceNotification = new EventSource("{{ mercure(path)|escape('js') }}");
        eventSourceNotification.onmessage = event => {
            animationQueue.addToQueue(() => {
                let data = event.data.split('_');
                if (data.length >= 5 ) {
                    new GameNotification(Number(data[0]), data[1], data[2], data[3], data[4]);
                }
                animationQueue.executeNextInQueue();
            })
        }
</script>
```

La première condition détermine la valeur de la route pour afficher les notifications selon le cas où l'on est un spectateur ou un joueur : certains messages peuvent être destinés pour un seul joueur uniquement. On a ensuite la gestion de l'événement qui réceptionne les messages contenant les données pour afficher une notification : ces données sont donc fournies en paramètre de la création d'une **GameNotification**.

4.2 Ajouter une notification

L'ajout d'une nouvelle notification se fait dans le contrôleur par un appel à la fonction **publishNotification** ajouté précédemment. Les données à fournir dans l'ordre en paramètre sont :

1. **$game** → La partie concernée.
2. **$duration** → La durée d'affichage de la notification avant l'auto-suppression.
3. **$message** → Message court alertant le joueur.
4. **$description** → Description en lien avec le message, peut être plus que le message.
5. **$iconId** → Id d'une icône pour la notification. l'Id correspond à l'Id d'un élément SVG définit dans *notification.html.twig*.
6. **$loadingBarColor** → Couleur de la barre d'expiration de la notification avant auto-suppression. Correspond au mot entre **bg-** et **-500** d'une couleur dans **Background Color**.
7. **$targetedPlayer** → Si une valeur est donnée, envoie la notification au joueur désigné, ou sinon aux spectateurs.

La réception d'une notification est prise en compte comme une animation : celle-ci est mise dans la queue d'animation. Pensez bien à mettre l'appel à **publishNotification** au bon moment dans votre contrôleur.

4.3 Ajouter une icône de notification

L'ajout d'une nouvelle icône pour vos notifications s'effectue dans le fichier *notification.html.twig*. Une icône correspond à un élément SVG à la fin du fichier. En vous basant de celles déjà présentes, vous pouvez en ajouté de nouvelle, sans oublier de modifier l'id de votre icône par **"svg_"** suivit d'un nom unique aux autres icônes. Veillez à ne pas avoir une icône trop grande au risque de déformer l'affichage de la notification. Testez sur différents formats et périphérique *(ordinateur, smartphone, ...)*.