Développement d'outils ou matériel d'enseignement de l'informatique

Développement d'un outil informatique démontrant visuellement la perception d'une liste pour un ordinateur

Collège du sud, Travail de maturité Version intermédiaire

Grégoire Geinoz

janv. 16, 2022

Table des matières

CHAPITRE 1

Introduction

1.1 Présentation de l'outil

L'objet de ce travail de maturité consiste en la programmation d'un outil permettant d'accompagner un professeur souhaitant développer une compréhension intuitive de la notion de liste à ses élèves. Il permet d'avoir une vision clair de ce qu'est une liste en programmation et quelles sont les manipulations qu'un ordinateur peut effectuer sur celle-ci.

Concrètement, une liste est représentée comme étant un "jeu de cartes". Chaque élément de la liste compose une carte avec sa valeur affichée d'un côté de la carte. Ainsi, tous les éléments de la liste forment à eux, le jeu de cartes au complet. Certains algorithmes requièrent des variables externes à la liste, ces variables sont créées via des appels de fonctions sur une API(à expliquer) et sont représentées dans une zone de l'écran spécifique. Cependant, des variables "spéciales", qui n'ont comme seul rôle, d'être un index qui parcours les éléments d'une liste (dans une "boucle for" par exemple), peuvent également être créées et sont représentées comme des flèches qui pointent sur l'élément de la liste qui correspond à l'index contenu dans la variable.

Le but étant de démontrer les actions opérées par un ordinateur sur une liste, les interactions entre les éléments de la liste entre eux ou avec une variable sont animées. Ainsi, pendant l'exécution d'un programme, il fera discrétement appel à l'API afin qu'une animation se déclenche et montre sur l'écran le comportement de la liste.

1.2 Intêret de l'outil

Les listes et les algorithmes sont des sujets importants à comprendre et à maîtriser dans le cadre de l'apprentissage de la programmation. Actuellement, ces concepts sont amenés aux étudiants et expliqués de manière bien trop abstraite. Il est nécessaire de comprendre ce que représente une liste pour un ordinateur et comment il peut interagir avec ; ainsi que de comprendre le fonctionnement des algorithmes, étape par étape.

Des plateformes d'enseignement de l'informatique tel que "code.org" ont un concept similaire au projet de ce travail de maturité : Développer une compréhension intuitive de concepts par l'expérimentation. Cependant, leur projet cible un public jeune pour leur inculquer des bases de programmations en posant brique par brique des blocs de code afin d'amener une cible jusqu'à son objectif. Le projet envisagé dans le cadre de ce travail de maturité cible un public plus mature capable de pousser un raisonnement plus en profondeur, en effet les listes sont une notion plus abstraite car

elles sont des conteneurs de valeurs quelconques. De plus, ce projet vise à enseigner des notions très spécifiques de la programmation : les listes.

1.3 Technologies utilisées

Le projet repose sur 2 technologies principales : Javascript et HTML5. Cependant pour alléger la quantité de travail, un framework de jeu 2D, Phaser, est utilisé pour gérer tout ce qui relève de l'affichage d'images, de la gestion de scènes et d'événements et autres qui impliquent la gestion d'objets (les cartes et les variables). Enfin, le projet étant écrit en Javascript principalement, un interpréteur de code peut être nécessaire si le langage étudié par l'outil est différent.

1.4 Configuration matérielle requise pour utiliser l'outil

Afin que l'utilisation de l'outil soit la plus accessible possible, les langages de programmation du Web ont été utilisé pour le développer. Ce qui signifie que pour utiliser l'outil, seul un accès à un ordinateur opérationnel étant doté d'un navigateur internet, et d'une connexion, suffit.

1.5 Connaissances requises pour utiliser l'outil

L'outil fait appel aux notions de variables ; éléments qui constituent la liste, ainsi qu'aux opérations de bases entre variables numérique (calculs, affectations de valeurs) et la notion de condition entre les variables utilisant des opérateurs de conditions.

1.6 Connaissances requises pour comprendre le fonctionnement de l'outil

Afin de comprendre comment l'outil fonctionne, il est nécessaire de comprendre le code qui le compose. C'est à dire qu'il faut avoir un niveau de base qui couvre tous les fondamentaux du Javascript. Le code étant largement commenté, il n'est pas forcément nécessaire de savoir utiliser le framework Phaser pour comprendre le code source.

CHAPITRE 2

Aspects pédagogiques

CHAPITRE 3

API

CHAPITRE 4

Scénarios d'utilisation

Ce chapitre vise à apporter des idées concrètes quant à l'utilisation du projet par un professeur pour ses élèves.

4.1 Introduction aux listes

4.2 Introduction aux algorithmes

CHAPITRE 5

Fonctionnement du projet

Le projet repose sur plusieurs grands aspects, qui permettent une répartition du code en différents systèmes.

5.1 L'utilisation de Phaser

Phaser se décrit comme un framework de création de jeux vidéos, il contient donc de nombreux utilitaires dans ce qui relève de la gestion d'images, de déplacements et d'effets en tous genres comme la distortion d'image ou des effets de transparence.

5.1.1 Système de scènes

L'un des plus gros avantage qu'offre Phaser est qu'il repose sur un système de scènes, qui possèdent des méthodes spécifiques permettant par exemple de pré-charger des images par la méthode « preload() » ou une méthode « update() » appelée plusieurs fois par seconde permettant d'actualiser des valeurs (très utile pour le système d'animations).

Afin de définir une scène, il faut créer une classe héritant de la scène de Phaser et d'implémenter ses méthodes :

```
class MainScene extends Phaser.Scene
{
    // Appelée lors de l'instanciation de la scène.
    constructor()
    {
        super("MainScene");
    }

    // Méthodes native aux scènes de Phaser :

    // Principalement pour charger les images.
    preload()
    {
        // ...
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }

    // Principalement pour créer des objets de Phaser.
    create()
    {
        // ...
    }

    // Appelée à interval régulier
    // pour actualiser des valeurs.
    update()
    {
        // ...
    }
}
```

Pour instancier la scène, il suffit simplement de créer une variable de configuration afin de définir la taille du « canvas » utilisé pour dessiner les images, le nombre d'actualisation par seconde de la fonction « update() » ainsi que le nom de la classe de la scène principale, et d'instancier une classe « Phaser.Game » :

```
const config = {
    // taille du canvas.
    width: 1000,
    height: 600,

    // Nombre d'actualisation par seconde.
    fps: {
        target: 60,
        forceSetTimeOut: true
    },

    // Scène principale.
    scene: [MainScene]
};



// Création du jeu avec les configurations.
const game = new Phaser.Game(config);
```

5.1.2 Système de gestion des cartes

Les scènes de Phaser permettent la création d'objets divers, comme des images, du texte, etc… qui seront affichés à l'écran. Les cartes et les variables sont en fait des objets images dont les coordonnées varient afin de les faire se déplacer :

```
class MainScene extends Phaser.Scene
{
    preload()
    {
        // Charge l'image des cartes en mémoire.
```

(suite sur la page suivante)

(suite de la page précédente)

```
        this.load.image("card", "images/card.png");
    }

    create()
    {
        // Crée une carte aux coordonnées (0; 0)
        // ayant comme image : "card".
        const card = this.add.image(0, 0, "card");
    }
}
```

5.1.3 Système d'évènements

Le système d'évènements de Phaser joue un rôle crucial dans le projet. En effet, il permet la communication entre l'API et le programme en lui-même. Pour ce faire, il est nécessaire d'instancier un objet « EventEmitter », proposer par Phaser. Cet objet permet d'émettre des événements et de les recevoir. C'est à dire qu'il est possible d'établir une connexion entre deux fichiers ou parties de code différentes en émettant un événement contenant des paramètres qui seront transmis à une autre partie du code qui appellera une fonction en y passant les paramètres spécifiés lors de l'envoi de l'événement. Cela revient en résumé à appeler une fonction qui est sensée être hors de portée :

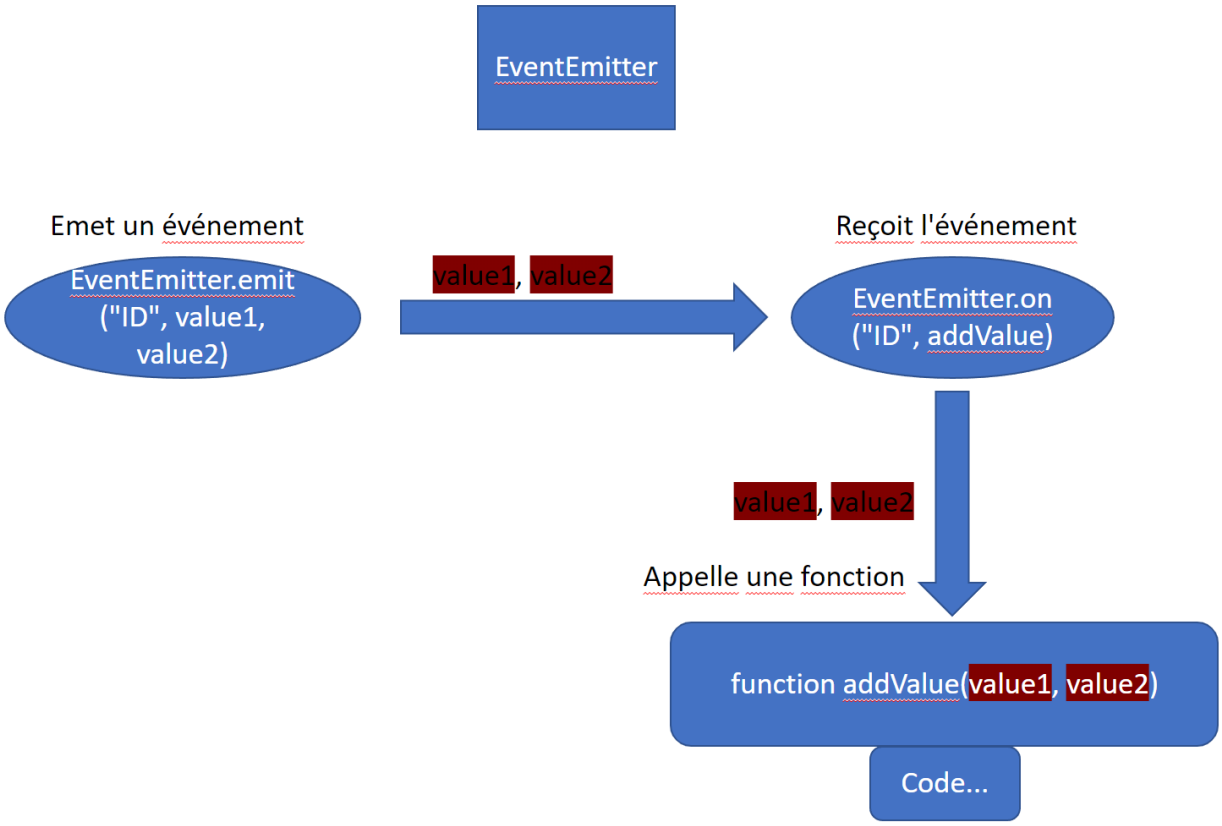

Fig. 1 – Fonctionnement du système d'événements de Phaser.

Concrètement, le code se construit de la manière suivante :

Avant tout, il faut instancier l'objet « EventEmitter » :

```
// eventEmitter.js

const eventEmitter = new Phaser.Events.EventEmitter();
```

Ensuite, le code de l'API émet les événements, par exemple :

```
// api.js

class Cards
{
    // Echange la position de deux cartes.
    swap(index1, index2)
    {
        // Emet l'événement.
        eventEmitter.emit(
            "swapCard", // Nom de l'événement.
            index1,     // Premier argument.
            index2      // Deuxième argument.
        );
    }
}
```

Enfin, le code du programme s'occupe d'intercepter l'événement :

```
class MainScene extends Phaser.Scene
{
    create()
    {
        eventEmitter.on(
            "swapCard",     // Nom de l'événement.
            this.swapCard,  // Fonction à appelée.
        );
    }

    swapCard(index1, index2)
    {
        // ...
    }
}
```

5.2 Système d'animations

Le système d'animation permet au développeur de créer des schémas d'animation. C'est à dire que, par exemple, le développeur peut aisément créer une animation qui engendre le déplacement simultané ou séquentiel d'une ou plusieurs cartes.

5.2.1 Principe fondamental

Naïvement, on pourrait penser qu'il suffit que chaque carte possède une propriété « animation » qui contient les informations nécessaires à décrire une animation, par exemple, de déplacement :

```
const card = {...};

card.animation = {
    toAnimate: true,
    type: "movement",

    // Coordonnées où la carte doit se déplacer.
    x: ...,
    y: ...
};
```

Et qu'ainsi, dans la fonction « update » gérée par Phaser, une « boucle for » parcours toutes les cartes et effectue l'animation qui lui est attachée, si celle-ci doit être animée :

```
update() // Exécutée 60 fois par seconde par Phaser.
{
    // Parcours toutes les cartes.
    for (let i = 0; i < nbCards; i++)
    {
        // Stocke la carte courante dans une variable.
        const currentCard = lstCards[i];

        if (currentCard.toAnimate)
        {
            // Gérer l'animation.
        }
    }
}
```

Or, cette manière de procédé comporte un gros désavantage. En effet, elle ne laisse au programme que la possibilité de gérer toutes les animations en même temps, ce qui signifie que si le développeur souhaite en jouer dans un certain ordre, il doit attendre que l'animation précédente soit terminer avant de configurer la suivante dans une ou plusieurs cartes. Ce n'est pas viable pour gérer une quantité importante d'animations qui s'exécutent à la suite.

C'est pour cela qu'un véritable système d'animation est nécessaire. Concrètement, la scène principale possède une propriété « animationQueue ». Il s'agit d'une liste initialement vide, qui stocke les animations les unes à la suite des autres. Ce principe simple permet de conserver l'ordre dans lequel les animations doivent être jouées ; selon l'ordre d'apparition dans la liste. Ce procédé nécessite donc également la création d'un objet « animation », qui sera l'objet stocké dans la liste animationQueue :

```
function moveCard(targetCard, ...)
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    const animation = {
        // Type de l'animation.
        type: "movement",

        // Contient un référence à la carte qui doit subir l'animation.
        card: targetCard,

        // Coordonnées où la carte doit se déplacer.
        x: ...,
        y: ...,

        // Devient vrai quand l'animation est arrivée à son terme.
        isFinished: false
    };

    // La liste d'animations relative à la scène principale.
    animationQueue.push(animation);
}
```

De cette manière, la fonction « update », gérée par Phaser, peut accéder à la liste d'animations et les jouer dans l'ordre. Ainsi, une fois que la première animation de la liste est arrivée à son terme, il suffit de la supprimer pour pouvoir traiter la suivante.

```
update() // Exécutée 60 fois par seconde par Phaser.
{
    // Si la liste d'animations n'est pas vide.
    if (animationQueue.length > 0)
    {
        // Stocke la première animation de la liste.
        const currentAnimation = animationQueue[0];

        // Identifie la type de l'animation.
        switch (currentAnimation.type)
        {
            // Type: Déplacement
            case "movement":
                // Récupère la référence de la carte concérnée par l
↪'animation.
                const card = currentAnimation.card;

                // Exécution de l'animation...
                // ...
                // ...

                if (/* Animation terminée */)
                {
                    currentAnimation.isFinished = true;
                }
                break;
            // ...
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Si l'animation courante est terminée.
        if (currentAnimation.isFinished)
        {
            // Retire la première animation de la liste.
            animationQueue.shift();
        }
    }
}
```

Cependant, comme chaque objet « animation » décrit le comportement d'une unique carte, il n'est pas possible d'exécuter plusieurs animations simultanément. Pour contourner ce problème, le système est programmé pour jouer toutes les animations à la suite qui ne sont pas interrompues par un objet « animation » de type « break » (« pause » ou « rupture » en anglais). Lorsque ce type est rencontré, le système s'assure que toutes les animations précédentes soient terminées avant de jouer le bloque d'animations suivant :

```
update() // Exécutée 60 fois par seconde par Phaser.
{
    // Parcours les animations de la liste (animationQueue).
    for (let i = 0; i < animationQueue.length; i++)
    {
        // Stocke l'animation courante de la liste.
        const currentAnimation = animationQueue[i];

        // Identifie le type de l'animation.
        if (currentAnimation.type == "movement")
        {
            // Récupère la référence de la carte concérnée par l'animation.
            const card = currentAnimation.card;

            // Exécution de l'animation...
            // ...
            // ...

            if (/* Animation terminée */)
            {
                currentAnimation.isFinished = true;
            }
        }
        // ...
        else if (currentAnimation.type == "break")
        {
            // Si le type break est le premier élément de la liste.
            if (i == 0)
            {
                // Retire le premier élément de la liste.
                animationQueue.shift();
            }

            // Met fin à l'exécution de la boucle for.
            break;
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Si l'animation courante est terminée.
        if (currentAnimation.isFinished)
        {
            // Supprime l'élément courant de la liste.
            animationQueue.splice(i, 1);
        }
    }
}
```

Bien que l'emploie du « switch » soit pratique pour identifier le type d'animation, il ne peut pas être utiliser dans le cas présent car l'instruction `break;` du type « break » doit mettre fin à la « boucle for » et non à l'instruction « switch ».

5.2.2 Déplacement des cartes

Le framework Phaser permet de déplacer ses objets par un procédé qui s'appelle le « tweening ». Ce procédé permet en effet de déplacer un objet d'un point A à un point B automatiquement et fluidement. Appliquer du tweening sur une carte exécute un code asynchrone modifiant les propriétés « x » et « y » de la carte concérnée afin de la faire se diriger vers le point souhaité :

```
    // "this" fait référence à la scène de Phaser.
    this.tweens.add({
        // Objet concerné par l'animation.
        targets: image,

        // Coordonnées de la destination.
        x: ...,
        y: ...,

        // Durée de l'animation.
        duration: ...,

        // Type d'accélération.
        ease: 'Power1',

        // Fonction appelé à la fin de l'animation
        onComplete: function() {...}
    });
```

Cependant, ce principe ne respecte pas le fondement du système d'animation développé précédement, qui consiste a stocké dans une liste toutes les animations créées, afin de pouvoir les jouer dans un ordre défini, simultanément ou non. En effet, le tweening proposé par Phaser déclenche une animation au moment-même où le tweening est créé, ou éventuellement avec un délai mesuré en microsecondes. De plus, le système de « callback » (appel d'une fonction à la fin de l'animation) rajoute une complexité supplémentaire dans la gestion des animations. Par conséquent, il est préférable que les déplacements des cartes ne relèvent pas de la responsabilité de Phaser.

Il est donc nécessaire que le programme gère ce type d'animation lui-même. Pour cela, l'objet « animation » de type « movement » doit faire appel à des notions de trigonométrie élémentaire afin de calculer l'angle en radian entre le point de départ et le point d'arrivé du déplacement :

```
const animation = {
    // Type de l'animation.
    type: "movement",
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Autres propriétés de l'animation.
    // ...

    // Angle entre le point de départ et d'arrivé.
    directionAngle: /* Angle en radian */
};
```

Pour obtenir l'angle de la direction, il suffit de calculer l'arc tangente du quotient de la différence d'ordonné sur la différence d'abscisse entre le point de départ et de d'arrivé :

$$\alpha = \arctan\left(\frac{\Delta y}{\Delta x}\right)$$

Javascript possède nativement un objet « Math » qui contient une fonction « atan2(y, x) » qui prend en paramètres les coordonnées « x » et « y » du point d'arrivé relativement au point de départ $(0\,;0)$ et retourne l'arc tangente formé par le quotient de « y » sur « x ». L'avantage de cette fonction est qu'elle gère elle-même les cas où « x » ou « y » serait égal à 0 :

```
const animation = {
    // Type de l'animation.
    type: "movement",

    // Autres propriétés de l'animation.
    // ...

    // Angle entre le point de départ et d'arrivé.
    directionAngle: Math.atan2((animation.y - card.futureY),
                               (animation.x - card.futureX));
```

Comme l'angle de la direction dans laquelle la carte doit se déplacer n'est calculé qu'une seul fois au moment de la création de l'animation, la fonction « update » de Phaser n'a plus qu'à actualiser les coordonnées de la carte en tenant compte de l'angle. Les fonctions trigonométriques Sinus/Cosinus de l'angle permettent d'obtenir le décalage horizontal et vertical approprié :

```
update() // Exécutée 60 fois par seconde par Phaser.
{
    // Parcours les animations de la liste (animationQueue).
    for (let i = 0; i < this.animationQueue.length; i++)
    {
        // Stocke l'animation courante de la liste.
        const currentAnimation = this.animationQueue[i];

        // Identifie le type de l'animation (mouvement).
        if (currentAnimation.type == "movement")
        {
            // Récupère la référence de la carte concérnée par l'animation.
            const card = currentAnimation.card;

            // Déplacement de la carte.
            card.x += Math.cos(currentAnimation.directionAngle);
            card.y += Math.sin(currentAnimation.directionAngle);
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

5.2.3 Retournement des cartes

Le retournement des cartes permet de montrer ou cacher à l'utilisateur le valeur d'une carte. Grâce à Phaser, ceci peut être géré aisément. En effet, chaque carte est une instance de la classe « Image » proposé par Phaser. Cette classe a une propriété « scaleX » et « scaleY » qui représentent respectivement l'étirement horizontal et vertical de l'image. Ainsi, modifier l'une ou l'autre de ces valeurs, modifie le rendu visuel de la carte en question. La valeur par défaut de ces propriétés est de 1, signifiant un étirement d'échelle 1 :1 ; plus la valeur est grande, plus l'image est étirée sur l'axe correspondant :

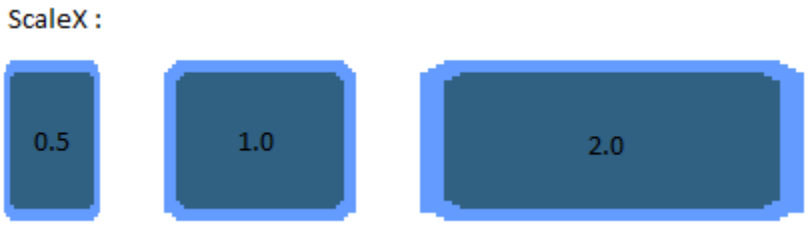

Fig. 2 – Effet d'étirement de la propriété scaleX.

Avec cette fonctionnalité, il suffit de rétrécir totalement la carte horizontalement et de l'étirer à nouveau jusqu'à sa taille originelle pour donner l'impression d'un retournement :

```
update() // Exécutée 60 fois par seconde par Phaser.
{
    // Parcours les animations de la liste (animationQueue).
    for (let i = 0; i < this.animationQueue.length; i++)
    {
        // Stocke l'animation courante de la liste.
        const currentAnimation = this.animationQueue[i];

        // Identifie le type de l'animation (retournement).
        if (currentAnimation.type == "flip")
        {
            // Récupère la référence de la carte concérnée par l'animation.
            const card = currentAnimation.card;

            card.scaleX -= 0.05;

            // Si la carte a fait un retournement complet.
            if (card.scaleX <= -1)
            {
                // Réinitialise l'étirment à sa valeur d'origine.
                card.scaleX = 1;

                // Animation terminée.
                currentAnimation.isFinished = true;
            }
```

(suite sur la page suivante)

(suite de la page précédente)

```
            }
        }
}
```