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

Visualisation interactive des opérations et algorithmes sur les listes à l'aide d'un jeu de cartes

Collège du sud, Travail de maturité Version finale

Grégoire Geinoz

mars 29, 2022

Table des matières

1 Introduction **1**
- 1.1 Motivation personnelle . . . 1
- 1.2 Présentation de l'outil . . . 1
- 1.3 Intérêt de l'outil . . . 2
- 1.4 Technologies utilisées . . . 2
- 1.5 Configuration matérielle requise pour utiliser l'outil . . . 2
- 1.6 Connaissances requises pour utiliser l'outil . . . 2
- 1.7 Connaissances requises au développement de l'outil . . . 3
- 1.8 Difficultés rencontrées lors du développement . . . 3

2 API **5**
- 2.1 Fonctionnement . . . 5

3 Scénarios d'utilisation **7**
- 3.1 Introduction aux listes . . . 7
- 3.2 Introduction aux algorithmes . . . 7

4 Aspects pédagogiques **11**
- 4.1 Constructivisme . . . 11
- 4.2 Modèle d'apprentissage de l'outil . . . 11
- 4.3 En quoi l'outil se distingue . . . 12
- 4.4 Potentiel du projet . . . 12

5 Fonctionnement de l'outil **13**
- 5.1 L'utilisation de Phaser . . . 13
- 5.2 Système d'animation . . . 17

6 Conclusion **25**
- 6.1 . . . 25

7 Tester l'outil **27**
- 7.1 Télécharger le dépôt GitHub . . . 27
- 7.2 Ouvrir un serveur HTTP local . . . 27
- 7.3 Fonctions disponibles de l'API . . . 28
- 7.4 Cartes . . . 28
- 7.5 Variables . . . 30

8 Glossaire **31**

9 Annexe **33**
9.1 Code source 33

Index **35**

CHAPITRE 1

Introduction

1.1 Motivation personnelle

Le choix du sujet de ce travail résulte de mon intérêt inlassable de toujours en apprendre plus sur les sujets qui me passionnent. Parmi ceux-ci, l'informatique ou plus précisément la programmation m'a toujours fasciné. Ainsi, si mon travail permet d'aider à appréhender et à comprendre des concepts qui semblent abstraits ou difficiles au premier abord, c'est naturellement que je m'impliquerai dans le projet. C'est donc avec le souhait de rendre un peu plus accessible le monde de la programmation que j'ai fait le choix de mon sujet.

1.2 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 claire 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 est représenté une carte avec sa valeur affichée d'un côté de la carte. Ainsi, tous les éléments de la liste forment 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 et sont représentées dans une zone de l'écran spécifique. Cependant, des variables "spéciales", qui n'ont comme seul rôle d'être un indice qui parcourt les éléments d'une liste (dans une "boucle for" par exemple), auraient également dû pouvoir être créées et auraient dû être représentées comme des flèches qui pointent sur l'élément de la liste qui correspond à l'indice 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, ce dernier fera discrètement appel à l'API, afin qu'une animation se déclenche et montre sur l'écran le comportement de la liste.

Malheureusement, toutes les fonctionnalités prévues initialement n'ont pas pu être développées avant la fin du travail. De plus, certaines fonctionnalités ont simplement été soustraites au cahier des charges, par exemple : la possibilité de déplacer les cartes avec la souris, de garder en mémoire un historique de toutes les actions effectuées afin de rejouer la série d'instructions, ou même la possibilité d'écrire du code directement dans un éditeur juxtaposé à la zone de rendu

des cartes. Bien qu'intéressantes, ces fonctionnalités ne sont pas essentielles au projet et ne sont donc envisageable que dans un second temps.

1.3 Intérêt 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 par exemple en posant brique par brique des blocs de code afin d'amener un personnage 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.

1.4 Technologies utilisées

Afin que l'utilisation de l'outil soit la plus accessible possible, les langages de programmation du Web ont été utilisés pour le développer. Le projet repose donc sur 2 technologies principales : Javascript et HTML5. Cependant pour alléger la quantité de travail, un framework [1] de jeu 2D, Phaser [2], 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.

1.5 Configuration matérielle requise pour utiliser l'outil

Comme l'outil a été développé dans un soucis de portabilité, il repose sur les technologies du Web. Ainsi, seul un accès à un ordinateur opérationnel étant doté d'un navigateur internet et d'une connexion est nécessaire pour utiliser l'outil.

1.6 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).

1. Voir glossaire
2. https ://phaser.io/

1.7 Connaissances requises au développement de l'outil

Afin de comprendre le fonctionnement de l'outil et pouvoir continuer son développement, un niveau de base qui couvre tous les fondamentaux du Javascript est nécessaire. Le code étant largement commenté, il n'est pas forcément essentiel de savoir utiliser le framework Phaser pour comprendre le code source.

1.8 Difficultés rencontrées lors du développement

A défaut d'avoir rencontré de nombreuses difficultés techniques, l'accroc principal a été la gestion du temps. En effet, l'estimation du temps pour l'ajout d'une fonctionnalité est très souvent largement inférieure à la réalité car de nombreux petits détails techniques n'apparaissent pas dans l'image globale qu'on s'imagine pour implémenter la fonctionnalité. De plus, programmer exige une grande concentration et d'être totalement impliqué afin d'être productif. Une certaine fatigue mentale survient donc rapidement et des pauses plus ou moins fréquentes sont nécessaires.

Une autre difficulté majeure rencontrée a été la gestion des animations. En effet, il s'agissait de mettre de l'ordre dans la manière dont les animations sont jouées. Par exemple, si une animation est en train d'être jouée et que, par la continuité du code, une autre animation doit être jouée, il faut attendre que la première animation soit finie avant de jouer la deuxième. Mais peut-être que l'on souhaite que les deux animations se jouent simultanément ? Il fallait permettre au développeur d'expliciter parfaitement comment les animations doivent s'enchaîner afin de prévenir tout comportement indéterminé (le code qui joue les animations est asynchrone, une animation qui dépend d'une autre n'a donc aucune connaissance de l'état de cette dernière). La manière dont cette difficulté a été gérée est expliquée en détail dans le chapitre du fonctionnement du code.

CHAPITRE 2

API

L'API proposée par le projet est l'essence même qui lui donne son sens. En effet, c'est elle qui permet l'interaction de l'utilisateur avec le programme principal.

2.1 Fonctionnement

Une API, abréviation de « Application Programming Interface », ou en français « interface de programmation d'application », désigne une façade exposant des fonctions qui peuvent être appelées depuis un autre code. Elle permet d'intégrer le programme dans une application quelconque :

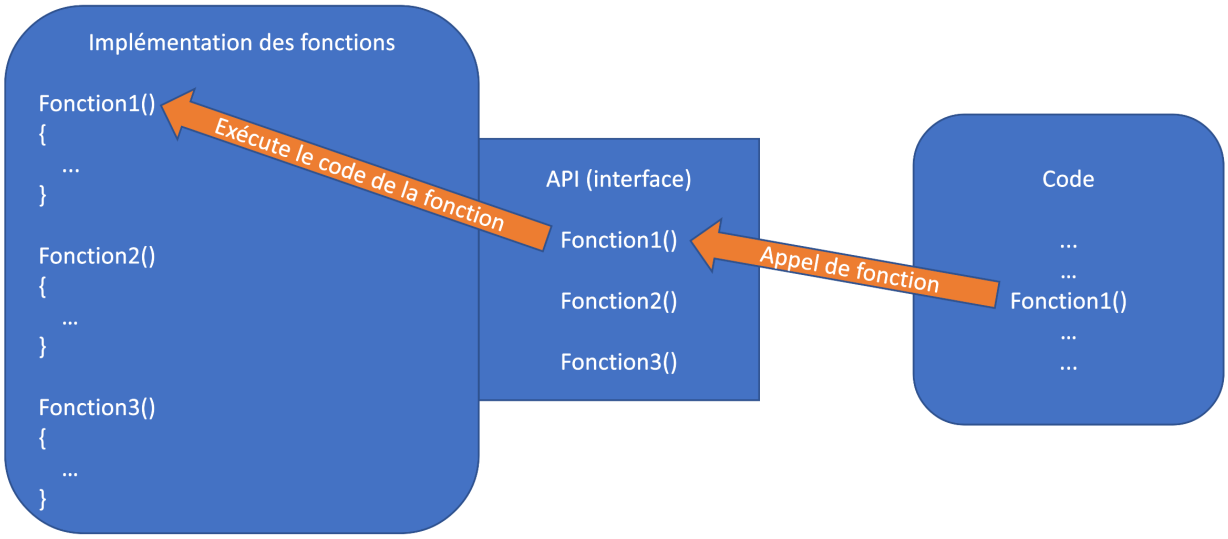

Fig. 1 – Fonctionnement de l'API.

CHAPITRE 3

Scénarios d'utilisation

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

3.1 Introduction aux listes

Un premier scénario, et pas des moindres, est de faire expérimenter aux élèves toutes les opérations qu'il est possible de réaliser avec des listes : ajouter des éléments, les supprimer, les déplacer, lire leur valeur, les comparer, etc.

Les expérimentations se font directement en écrivant du code dans un langage quelconque, tant qu'il permet d'appeler les fonctions de l'API au moment d'une opération.

3.2 Introduction aux algorithmes

Si les notions de bases de la gestion des listes sont acquises, alors l'outil devrait être en tout cas partiellement adapté à démontrer visuellement les opérations effectuées par un algorithme reposant sur l'utilisation d'une liste :

3.2.1 Sommer tous les éléments d'une liste

```
1  // Liste quelconque.
2  const liste = [1, 2, 3, 4, 5, 6, 7, 8, 9];
3
4  // La somme est nulle au départ.
5  let somme = 0;
6
7  // Parcourt la liste.
8  for (let i = 0; i < liste.length; i++)
9  {
10     somme += liste[i];
```

(suite sur la page suivante)

(suite de la page précédente)

```
11 }
12
13 // Affiche la somme dans la console.
14 console.log("La somme est : " + somme);
```

Dans cet algorithme certaines lignes peuvent faire appel à l'API afin d'animer l'exécution du programme et de mieux comprendre les opérations effectuées :

- Ligne `2` : `cards.appendList(liste)` anime la création de la liste initiale.
- Ligne `5` : `vars.create("somme", 0)` anime la création de la variable `somme` avec sa valeur de `0`.
- Ligne `10` : `cards.add(i, "somme")` ajoute la valeur de la carte à la position `i` à la valeur de la variable `somme`.

3.2.2 Trouver le minimum

Cet algorithme simple à mettre en place, permet d'avoir un avant-goût du potentiel de l'outil. Sur une liste non-triée, voici à quoi ressemblerait un algorithme qui recherche la plus petite valeur dans une liste :

```
1  // List non-triée.
2  const liste = [3, 7, 2, 9, 1, 8, 4, 6, 5];
3
4  // Le minimum est un nombre de la liste.
5  let minimum = liste[0];
6
7  // Parcourt la liste.
8  for (let i = 0; i < liste.length; i++)
9  {
10     if (liste[i] < minimum)
11     {
12         minimum = liste[i];
13     }
14 }
15
16 // Affiche le minimum dans la console.
17 console.log("Le minimum est : " + minimum);
```

- Ligne `2` : `cards.appendList(liste)` anime la création de la liste initiale.
- Ligne `5` : `vars.create("minimum")` et `cards.read(0, "minimum")` anime la création de la variable `minimum` et lui assigne la valeur de la carte à l'indice `0`.
- Ligne `10` : Il aurait été intéressant de pouvoir montrer visuellement la condition mais cette fonctionnalité ne figure pas dans la version actuelle de l'outil.
- Ligne `12` : `cards.read(i, "minimum")` assigne la valeur de la carte à l'indice `i` à la variable `minimum`.

3.2.3 Recherche dichotomique

La recherche dichotomique fait parti des algorithmes intéressants auxquels l'outil n'apporte pas encore de réels soutiens visuels qui pourraient aider à la compréhension, les fonctionnalités cruciales qu'il manquent à l'outil est la possibilité de créer des variables spéciales dont la valeur est un indice qui désigne une carte de la liste, et la visualisation des conditions :

```
1  // La liste passée en paramètre doit être triée.
2  function recherche(liste, valeur)
3  {
4      // Borne inférieure prise en compte dans la recherche.
5      let premier = 0;
6
7      // Borne supérieure prise en compte dans la recherche.
8      let dernier = liste.length - 1;
9
10     // Déclarations de variables utiles.
11     let milieu;
12     let element;
13
14     while (premier <= dernier)
15     {
16         // Milieu des bornes inférieure et supérieure.
17         milieu = Math.floor((premier + dernier) / 2);
18
19         // Element courant de la liste.
20         element = liste[milieu];
21
22         if (element == valeur)
23         {
24             // Retourne l'indice de l'élément trouvé.
25             return milieu;
26         }
27         else if (element < valeur)
28         {
29             // Décale la borne inférieure après le milieu.
30             premier = milieu + 1;
31         }
32         else
33         {
34             // Décale la borne supérieure avant le milieu.
35             dernier = milieu - 1;
36         }
37     }
38
39     // Retourne -1 si la valeur n'a pas été trouvée.
40     return -1;
41 }
42
43 // Affiche les résultats de recherches.
44 console.log(recherche([1, 2, 3, 4, 5, 6, 7, 8, 9], 5));  // 4
45 console.log(recherche([1, 2, 3, 4, 5, 6, 7, 8, 9], 10)); // -1
```

Aux lignes `5`, `8` et `11`, le mieux aurait été de pouvoir créer des variables spéciales de type « indice » qui montre

visuellement à quelles cartes elles font référence dans la liste. De plus, aux lignes `14`, `22` et `27`, l'animation des conditions ne figurent pas encore comme fonctionnalités de l'outil :

- Ligne `5` : `vars.create("premier", 0)` anime la création de la variable `premier`.
- Ligne `8` : `vars.create("dernier", liste.length - 1)` anime la création de la variable `dernier`.
- Ligne `11` : `vars.create("milieu")` anime la création de la variable `milieu` avec une valeur par défaut.
- Ligne `12` : `vars.create("element")` anime la création de la variable `element` avec une valeur par défaut.
- Ligne `17` : `vars.assign("milieu", Math.floor((premier + dernier) / 2))` anime l'assigniation de la valeur à la variable `milieu`.
- Ligne `20` : `vars.assign("element", liste[milieu])` anime l'assigniation de la valeur à la variable `element`.
- Ligne `30` : `vars.assign("premier", milieu + 1)` anime l'assigniation de la valeur à la variable `premier`.
- Ligne `35` : `vars.assign("dernier", milieu - 1)` anime l'assigniation de la valeur à la variable `dernier`.

CHAPITRE 4

Aspects pédagogiques

Ce chapitre traite un aspect des sciences de l'éducation qui tend à montrer que l'on assimile mieux les concepts en les découvrant par soi-même, et en expérimentant.

4.1 Constructivisme[1]

Le constructivisme est un modèle d'apprentissage développé par *Jean Piaget*. Il pense que toute nouvelle connaissance est basée sur des connaissances déjà acquises ; que les connaissances sont construites et adaptées à partir de ce que l'on sait déjà, d'où son nom « constructivisme ».

Selon *Piaget*, la connaissance naît des interactions d'un sujet avec un objet. Ainsi, l'environnement doit amener le sujet à se confronter à un scénario qu'il ne connaît pas et qui demande des connaissances nouvelles afin d'être compris. De cette manière, le sujet peut interagir avec l'objet du scénario et découvrir un concept par lui-même en le basant sur ce qu'il sait déjà pour finalement se créer ou affiner une représentation personnelle du concept.

4.2 Modèle d'apprentissage de l'outil

L'outil développé dans le cadre de ce travail base son modèle d'apprentissage sur la philisophie constructivisme. En effet, tous les concepts qu'il permet d'aborder nécessitent l'utilisation d'une liste, qui est représentée tel un objet réel, sous la forme d'un « jeu de cartes ». Avec cet objet de base, selon la façon dont l'API est intégrée, un étudiant peut interagir avec la liste en appelant indirectement des fonctions de l'API à travers du code, ou constater les effets qu'ont certaines lignes de code sur la liste. Ainsi, l'étudiant peut découvrir des concepts par lui-même sur la base de ce qu'il connaît déjà (les notions de variables et les opérations entre variables numériques).

1. Selon Piaget « https ://www.youtube.com/watch ?v=mcjY4KU1HWM&ab_channel=CentreJeanPiaget »

4.3 En quoi l'outil se distingue

Comme cité dans l'introduction, il existe des outils similaires à celui développer ici. Cependant, celui-ci se distingue des autres en grande partie grâce à sa portabilité, à sa personnalisation et à son large spectre d'applications. En effet, il peut être implémenter par n'importe qui dans un langage quelconque tant qu'il est possible d'appeler des fonctions de l'API discrètement pendant l'exécution du code ou après, instruction par instruction. L'API pourrait même être implémenter dans un projet plus original basé sur des suites d'instructions représentées par des briques, comme dans un des projets de "code.org" [2]. Enfin, il peut faire face à divers scénarios afin de visualiser des algorithmes ou tout simplement des instructions sur une liste afin de comprendre comment elle réagit. En somme, ce qui distingue l'outil de ce qui existe déjà dans le même registre, dans le monde de la pédagogie, est qu'il permet d'expliquer des concepts variés et qu'il peut être implémenter selon la manière d'enseigner qui convient le mieux aux besoins d'un professeur.

4.4 Potentiel du projet

Le projet possède certainement un potentiel encore faibrilement exploité qui le dépasse. En effet, il n'est encore qu'à un stade peu évolué aux niveaux des animations et de son spectre d'application qui peut être largement élargi. Le concept de l'outil pourrait être transposable pour visualiser une multitudes de notions plus ou moins complexes, en gardant l'aspect de portabilité et de personnalisation au niveau de l'intégration de l'outil. Des concepts comme la récursivité ou la gestion de la mémoire en Python (qui possède un « garbage collector » [3]) sont des notions à priori complexes qui pourraient faire l'objet d'une extension intéressante de l'outil.

---

2. Lien du projet : « https ://studio.code.org/s/aquatic/ »
3. *ramasse-miettes* en anglais

CHAPITRE 5

Fonctionnement de l'outil

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

5.1 L'utilisation de Phaser

Phaser [1] se décrit comme un ***framework*** [2] de création de jeux vidéo, il contient donc de nombreux utilitaires facilitant la gestion d'images, les déplacements et des 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 base 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.
```

(suite sur la page suivante)

1. https ://phaser.io/
2. Voir glossaire

(suite de la page précédente)

```
    preload()
    {
        // ...
    }

    // 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*** [3] 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);
```

3. Voir glossaire

5.1.2 Système de gestion des cartes

Les scènes de Phaser permettent la création d'objets divers, comme des images, du texte, ou autre, à 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.
        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'événements

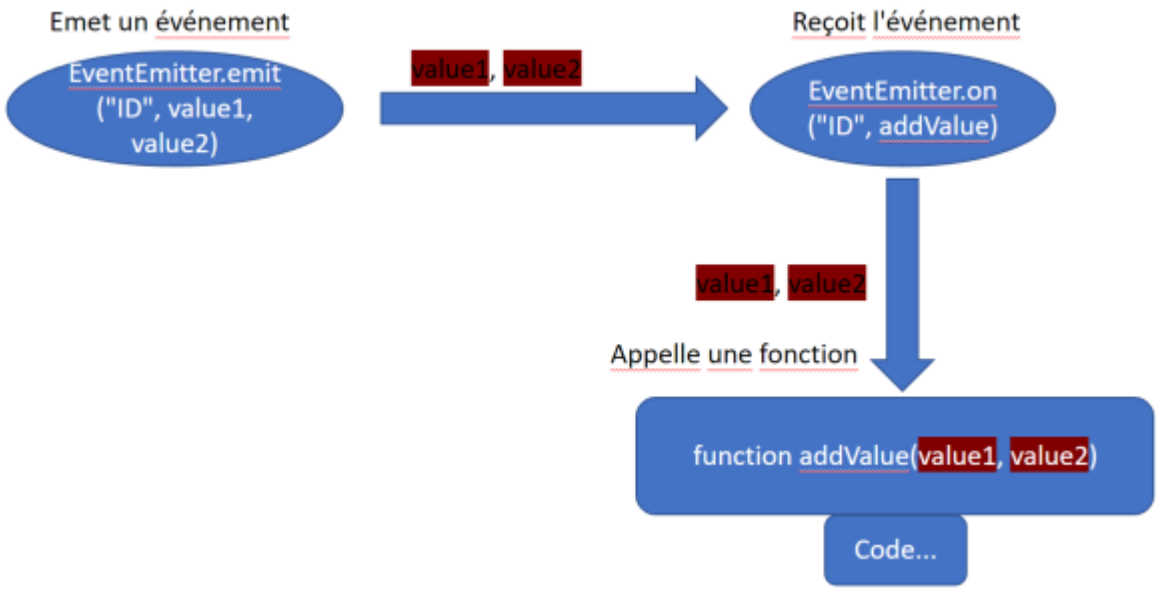

Fig. 1 – Fonctionnement du système d'événements de Phaser.

Le système d'événements 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*[4], proposé 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 plusieurs 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 lui passant les paramètres spécifiés lors de l'envoi de

4. https ://photonstorm.github.io/phaser3-docs/Phaser.Events.EventEmitter.html

l'événement. Cela revient en résumé à appeler une fonction qui est censée être hors de portée. 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 :

```
// 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 appeler.
        );
    }

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

5.2 Système d'animation

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* parcourt 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.
{
    // Parcourt 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 procéder 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 terminée 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.
                const card = currentAnimation.card;

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

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

        // Si l'animation courante est terminée.
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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* [5]. Lorsque ce type est rencontré, le système s'assure que toutes les animations précédentes sont terminées avant de jouer le bloc 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;
        }

        // Si l'animation courante est terminée.
```

(suite sur la page suivante)

5. *pause* ou *rupture* en anglais

(suite de la page précédente)

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

Bien que l'emploi du *switch* soit pratique pour identifier le type d'animation, il ne peut pas être utilisé 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* [6]. 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 concernée 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 précédemment, qui consiste à stocker 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* [7] (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 élémentaires afin de calculer l'angle en radians entre le point de départ et le point d'arrivée du déplacement :

```
const animation = {
    // Type de l'animation.
```

(suite sur la page suivante)

6. Voir glossaire
7. *Fonction de rappel* en français, voir glossaire

(suite de la page précédente)

```
    type: "movement",

    // 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'ordonnée sur la différence d'abscisse entre le point de départ et d'arrivée :

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

Javascript possède nativement un objet *Math*[8] qui contient une fonction *atan2(y, x)*[9] qui prend en paramètres les coordonnées x et y du point d'arrivée 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 seule 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);
```

(suite sur la page suivante)

8. https ://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Math
9. https ://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Math/atan2

(suite de la page précédente)

```
            card.y += Math.sin(currentAnimation.directionAngle);
        }
    }
}
```

5.2.3 Retournement des cartes

Le retournement des cartes permet de montrer ou de cacher à l'utilisateur la valeur d'une carte. Grâce à Phaser, ceci peut être géré aisément. En effet, chaque carte est une instance de la classe *Image* [10] proposée par Phaser. Cette classe possède des propriétés *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 :

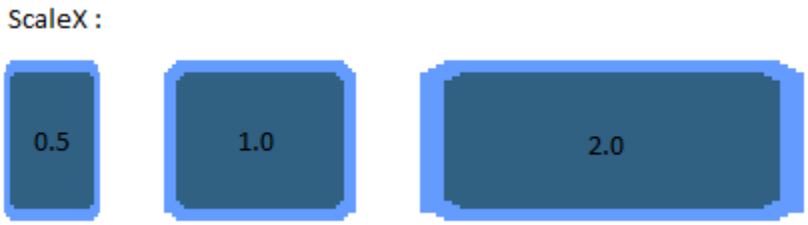

Fig. 2 – Effet d'étirement de la propriété scaleX.

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 concernée 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;
```

(suite sur la page suivante)

10. https ://photonstorm.github.io/phaser3-docs/Phaser.GameObjects.Image.html

(suite de la page précédente)

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

CHAPITRE 6

Conclusion

6.1

CHAPITRE 7

Tester l'outil

Afin de tester l'outil, il faut au préalable préparer son environnement :

7.1 Télécharger le dépôt GitHub

Premièrement, il faut se procurer le code du projet. Pour cela, télécharger le dépôt suivant : `https://github.com/GeinozG/TM-2021`.

7.2 Ouvrir un serveur HTTP local

Il faut à présent configurer un *serveur http* local. Plusieurs manières d'ouvrir un serveur sont possibles mais la plus simple est probablement en utilisant Python. Après avoir téléchargé une version récente de Python, il suffit d'ouvrir un invite de commandes dans le dossier du dépôt GitHub précédemment téléchargé et d'entrer la commande suivante : `python -m http.server 8000`. Il ne reste plus qu'à ouvrir un navigateur Web et d'entrer l'adresse suivante : `http://localhost:8000`.

Une zone noire doit normalement être visible sur l'écran. Afin de créer des cartes ou des variables, il faut ouvrir le terminal d'outil de développement (en appuyant sur F12). Depuis cette fenêtre, allez sur l'onglet console et entrez ces commandes afin d'initialiser l'API :

```
const cards = new Cards();

const vars = new Variables();
```

7.3 Fonctions disponibles de l'API

7.4 Cartes

Pour utiliser ces fonctions, il suffit d'écrire `cards.` suivit du nom de la fonction et de ses paramètres.

7.4.1 Append

Ajoute un certain nombre de cartes avec des valeurs similaire à la fin de la liste.

```
append(value = 0, nbCards = 1)
```

7.4.2 Append list

Ajoute une liste de cartes avec des valeurs variables à la fin de la liste.

```
appendList(lstValues)
```

7.4.3 Insert

Insère un carte ayant une valeur à un index donné dans la liste.

```
insert(value, index)
```

7.4.4 Pop

Retire le dernier élément de la liste.

```
pop()
```

7.4.5 Swap

Echange la position de deux cartes.

```
swap(index1, index2)
```

7.4.6 Move

Change la position d'une carte par rapport à son index courant et l'index de destination.

```
move(indexCard, indexDestination)
```

7.4.7 Read

Lit la valeur d'une carte et la stocke dans une variable.

```
read(index, outVarName)
```

7.4.8 Assign

Assigne une valeur à une carte de la liste selon son index.

```
assign(index, value)
```

7.4.9 Add

Additione la valeur de deux cartes et la stocke dans une variable.

```
add(index1, index2, outVarName)
```

7.4.10 Multiply

Multiplie la valeur de deux cartes et la stocke dans une variable.

```
multiply(index1, index2, outVarName)
```

7.4.11 Substract

Soustrait la valeur de deux cartes et la stocke dans une variable.

```
substract(index1, index2, outVarName)
```

7.4.12 Divide

Divise la valeur de deux cartes et la stocke dans une variable.

```
divide(index1, index2, outVarName)
```

7.4.13 Show index

Montre ou non l'indice de chaque carte de la liste (vrai par défaut).

```
showIndex(shouldShow)
```

7.5 Variables

Pour utiliser ces fonctions, il suffit d'écrire `vars.` suivit du nom de la fonction et de ses paramètres.

7.5.1 Create

Crée une nouvelle variable.

```
create(name, defaultValue)
```

7.5.2 Delete

Supprime une variable.

```
delete(name)
```

7.5.3 Assign

Assigne une valeur à une variable.

```
assign(name, value)
```

CHAPITRE 8

Glossaire

Framework : « Désigne un ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou partie d'un logiciel. » [1]

Canvas : « Ajouté en HTML5, l'élément **canvas** est un nouvel élément qui peut être utilisé pour dessiner des graphismes via des scripts JavaScript. » [2]

Tweening : « Le tweening (ou interpolation) est un procédé dans le film d'animation qui permet de créer des images intermédiaires successives de telle sorte qu'une image s'enchaîne agréablement et de façon fluide avec la suivante. » [3]

Fonction de rappel : « En informatique, une fonction de rappel (callback en anglais) ou fonction de post-traitement est une fonction qui est passée en argument à une autre fonction. Cette dernière peut alors faire usage de cette fonction de rappel comme de n'importe quelle autre fonction, alors qu'elle ne la connaît pas par avance. » [4]

1. https ://fr.wikipedia.org/wiki/Framework
2. https ://developer.mozilla.org/fr/docs/Web/API/Canvas_API
3. https ://fr.wikipedia.org/wiki/Tweening
4. https ://fr.wikipedia.org/wiki/Fonction_de_rappel

CHAPITRE 9

Annexe

9.1 Code source

Index

C

Canvas:, 31

F

Fonction de rappel:, 31
Framework:, 31

T

Tweening:, 31